Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Configurable Stochastic Analysis Framework for Asynchronous Systems

Scientific Students' Associations Report

Authors:

Attila Klenik
Kristóf Marussy

Supervisors:

dr. Miklós Telek
Vince Molnár
András Vörös

2015.

Contents

Összefoglaló A kritikus rendszerek – biztonságkritikus, elosztott és felhőalkalmazások – helyességének biztosításához szükséges a funkcionális és nemfunkcionális követelmények matematikai igényességű ellenőrzése. Számos, szolgáltatásbiztonsággal és teljesítményvizsgálattal kapcsolatos tipikus kérdés általában sztochasztikus analízis segítségével válaszolható meg.

A kritikus rendszerek elosztott és aszinkron tulajdonságai az *állapottér robbanás* jelenségéhez vezetnek. Emiatt méretük és komplexitásuk gyakran megakadályozza a sikeres sztochasztikus analízist, melynek számításigénye nagyban függ a lehetséges viselkedések számától. A modellek komponenseinek jellegzetes időbeli viselkedése a számításigény további jelentős növekedését okozhatja.

A szolgáltatásbiztonsági és teljesítményjellemzők kiszámítása markovi modellek állandósult állapotbeli és tranziens megoldását igényli. Számos eljárás ismert ezen problémák kezelésére, melyek eltérő reprezentációkat és numerikus algoritmusokat alkalmaznak; ám a modellek változatos tulajdonságai miatt nem választható ki olyan eljárás, mely minden esetben hatékony lenne.

A markovi analízishez szükséges a modell lehetséges viselkedéseinek, azaz állapotterének felderítése, illetve tárolása, mely szimbolikus módszerekkel hatékonyan végezhető el. Ezzel szemben a sztochasztikus algoritmusokban használt vektor- és indexműveletek szimbolikus megvalósítása nehézkes. Munkánk célja egy olyan, integrált keretrendszer fejlesztése, mely lehetővé teszi a komplex sztochasztikus rendszerek kezelését a szimbolikus módszerek, hatékony mátrix reprezentációk és numerikus algoritmusok előnyeinek ötvözésével.

Egy teljesen szimbolikus algoritmust javasolunk a sztochasztikus viselkedéseket leíró mátrix-dekompozíciók előállítására a szimbolikus formában adott állapottérből kiindulva. Ez az eljárás lehetővé teszi a temporális logikai kifejezéseken alapuló szimbolikus technikák használatát.

A keretrendszerben megvalósítottuk a konfigurálható sztochasztikus analízist: megközelítésünk lehetővé teszi a különböző mátrix reprezentációk és numerikus algoritmusok kombinált használatát. Az implementált algoritmusokkal állandósult állapotbeli költség- és érzékenység analízis, tranziens költséganalízis és első hiba várható bekövetkezési idő analízis végezhető el sztochasztikus Petri-háló (SPN) markovi költségmodelleken. Az elkészített eszközt integráltuk a PetriDotNet modellező szoftverrel. Módszerünk gyakorlati alkalmazhatóságát szintetikus és ipari modelleken végzett mérésekkel igazoljuk.

Abstract Ensuring the correctness of critical systems – such as safety-critical, distributed and cloud applications – requires the rigorous analysis of the functional and extra-functional properties of the system. A large class of typical quantitative questions regarding dependability and performability are usually addressed by stochastic analysis.

Recent critical systems are often distributed/asynchronous, leading to the well-known phenomenon of *state space explosion*. The size and complexity of such systems often prevents the success of the analysis due to the high sensitivity to the number of possible behaviors. In addition, temporal characteristics of the components can easily lead to huge computational overhead.

Calculation of dependability and performability measures can be reduced to steady-state and transient solutions of Markovian models. Various approaches are known in the literature for these problems differing in the representation of the stochastic behavior of the models or in the applied numerical algorithms. The efficiency of these approaches are influenced by various characteristics of the models, therefore no single best approach is known.

The prerequisite of Markovian analysis is the exploration of the state space, i.e. the possible behaviors of the system. Symbolic approaches provide an efficient state space exploration and storage technique, however their application to support the vector operations and index manipulations extensively used by stochastic algorithms is cumbersome. The goal of our work is to introduce a framework that facilitates the analysis of complex, stochastic systems by combining together the advantages of symbolic algorithms, compact matrix representations and various numerical algorithms.

We propose a fully symbolic method to explore and describe the stochastic behaviors. A new algorithm is introduced to transform the symbolic state space representation into a decomposed linear algebraic representation. This approach allows leveraging existing symbolic techniques, such as the specification of properties with *Computational Tree Logic* (CTL) expressions.

The framework provides configurable stochastic analysis: an approach is introduced to combine the different matrix representations with numerical solution algorithms. Various algorithms are implemented for steady-state reward and sensitivity analysis, transient reward analysis and mean-time-to-first-failure analysis of stochastic models in the *Stochastic Petri Net* (SPN) Markov reward model formalism. The analysis tool is integrated into the PetriDotNet modeling application. Benchmarks and industrial case studies are used to evaluate the applicability of our approach.

Chapter 1

Introduction

The growing need for ensuring the correctness of critical systems – such as safety-critical, distributed and cloud applications – requires the rigorous analysis of the functional and extra-functional properties. A large class of typical quantitative questions regarding dependability and performability are usually addressed by stochastic analysis.

Recent critical systems are often distributed/asynchronous, leading to the well-known phenomenon of *state space explosion*. The size and complexity of such systems often prevents the success of the analysis due to the high sensitivity to the number of possible behaviors. In addition, temporal characteristics of the components can easily lead to huge computational overhead or prevent algorithms from convergence.

Calculation of dependability and performability measures can be reduced to steady-state and transient solutions of Markovian models. Various approaches are known in the literature for these problems differing in the representation of the stochastic behavior of the models or in the applied numerical algorithms. The efficiency of these approaches are influenced by various characteristics of the models, therefore no single best approach is known.

In this paper our goal is to propose a solution for the various problems occuring in stochastic analysis of complex systems.

The first step in Markovian analysis is the exploration of the state space, i.e. the possible behaviors of the system. Various algorithms exist for state space exploration. We addressed the state space traversal problem with the development of both explicit state algorithms and symbolic approaches. Explicit state traversal is fast in general and handles even systems with complex transition functions, while symbolic state space traversal can handle even huge state spaces. While symbolic approaches provide an efficient state space exploration and storage technique, their application to support the vector operations and index manipulations extensively used by stochastic algorithms is cumbersome. In this paper we propose a fully symbolic algorithm to bridge the gap between symbolic state space representation and the data structures intensively used

by our stochastic analysis algorithms. The new algorithm is introduced to transform the symbolic state space representation into a decomposed linear algebraic representation. This approach allows leveraging existing symbolic techniques, such as the specification of properties with *Computational Tree Logic* (CTL) expressions.

We introduce the concept of configurable stochastic analysis. We developed a framework to support the combination of

- various state space exploration techniques with
- decomposition algorithms and representation, techniques for the stochastic behaviour of the systems
- various numerical algorithms to solve the steady-state and transient analysis problem,
- computation of high level measures such as various reward, sensitivity and mean time to failure values.

Various problems were solved during our work: an approach is introduced to combine the different matrix representations with numerical solution algorithms. A diverse set of algorithms are implemented for steady-state reward and sensitivity analysis, transient reward analysis and mean-time-to-first-failure analysis of stochastic models in the *Stochastic Petri Net* (SPN) Markov reward model formalism. Several optimizations and improvements were applied to provide efficient algorithms. Most of the developed algorithms are parallelized to exploit the modern multicore architectures. Benchmarks and industrial case studies are used to evaluate the applicability of our approach.

The analysis framework is integrated into the PetriDotNet modeling application. More than 78 000 unit tests are generated with a combinatorial interface testing approach to ensure the correctness of the data structure. To validate the stochastic analysis pipeline and the implemented algorithms through software redundancy, 588 mathematically consistent configurations of the pipeline are executed and evaluated for several models.

The remainder of this work is structured as follows: Chapter 2 reviews some preliminaries of the stochastic analysis of stochastic Petri nets. Chapter 3 presents the configurable stochastic analysis pipeline. Chapter 4 describes the available state space exploration and algorithms and decompositions of stochastic behaviors, including the hierarchical decomposition algorithm for symbolic state spaces in Section 4.2.3. Chapter 5 presents numerical steady-state and transient analysis algorithms and their implementations in our framework, with special attention to the homogenous linear equation systems arising from steady-state analysis. After describing the testing and validation

methodologies applied to our framework in Chapter 6 as well as the benchmark results, we conclude our paper in Chapter 7.

Chapter 2

Background

2.1 Petri nets

Petri nets are a widely used graphical and mathematical modeling tool for systems which are concurrent, asynchronous, distributed, parallel or nondeterministic.

Definition 2.1 A *Petri net* is a 5-tuple $PN = (P, T, F, W, M_0)$, where
- $P = \{p_0, p_1, \ldots, p_{n-1}\}$ is a finite set of places;
- $T = \{t_0, t_1, \ldots, t_{m-1}\}$ is a finite set of transitions;
- $F \subseteq (P \times T) \cup (P \times T)$ is a set of arcs, also called the flow relation;
- $W : F \to \mathbb{N}^+$ is an arc weight function;
- $M_0 : P \to \mathbb{N}$ is the initial marking;
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$ [49].

Arcs from P to T are called *input arcs*. The input places of a transition t are denoted by ${}^\bullet t = \{p : (p, t) \in F\}$. In contrast, arcs of the form (t, p) are called *output arcs* and the output places of t are denoted by $t^\bullet = \{p : (t, p) \in F\}$.

A *marking* $M : P \to \mathbb{N}$ assigns a number of *tokens* to each place. The transition t is *enabled* in the marking M (written as $M[t\rangle$) when $M(p) \geq W(p, t)$ for all $p \in {}^\bullet t$.

Petri nets are graphically represented as edge weighted directed bipartite graphs. Places are drawn as circles, while transitions are drawn as bars or rectangles. Edge weights of 1 are ususally omitted from presentation. Dots on places correspond to tokens in the current marking.

If $M[t\rangle$ holds the transition t can be *fired* to get a new marking M' (written as $M[t\rangle M'$) by decreasing the token counts for each place $p \in {}^\bullet t$ by $W(p, t)$ and increasing the token counts for each place $p \in t^\bullet$ by $W(t, p)$. Note that in general, ${}^\bullet t$ and $t^\bullet$ need

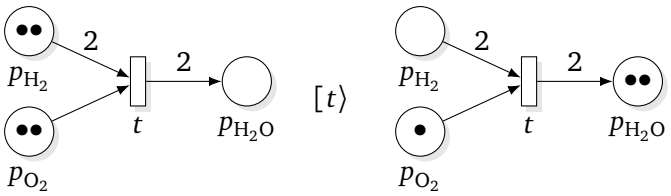


Figure 2.1 A Petri net model of the reaction of hydrogen and oxygen.

not be disjoint. Thus, the firing rule can be written as

$$M'(p) = M(p) - W(p,t) + W(t,p), \tag{2.1}$$

where we take $W(x,y) = 0$ if $(x,y) \notin F$ for brevity.

A marking M' is *reachable* from the marking M (written as $M \rightsquigarrow M'$) if there exists a sequence of markings and transitions for some finite k such that

$$M = M_1 \,[t_{i_1}\rangle\, M_2 \,[t_{i_2}\rangle\, M_3 \,[t_{i_3}\rangle \cdots [t_{i_{k-1}}\rangle\, M_{k-1} \,[t_{i_k}\rangle\, M_k = M'.$$

A marking M is in the reachable *state space* of the net if $M_0 \rightsquigarrow M$. The set of all markings reachable from M_0 is denoted by

$$RS = \{M : M_0 \rightsquigarrow M\}.$$

Definition 2.2 The Petri net PN is k-*bounded* if $M(p) \leq k$ for all $M \in RS$ and $p \in P$. PN is *bounded* if it is k-bounded for some (finite) k.

The reachable state space RS is finite if and only if the Petri net is bounded.

Example 2.1 The Petri net in Figure 2.1 models the chemical reaction

$$2\,H_2 + O_2 \rightarrow 2\,H_2O.$$

In the initial marking (left) there are two hydrogen and two oxygen molecules, represented by tokens on the places p_{H_2} and p_{O_2}, therefore the transition t is enabled. Firing t yields the marking on the right where the two tokens on p_{H_2O} are the products of the reaction. Now t is no longer enabled.

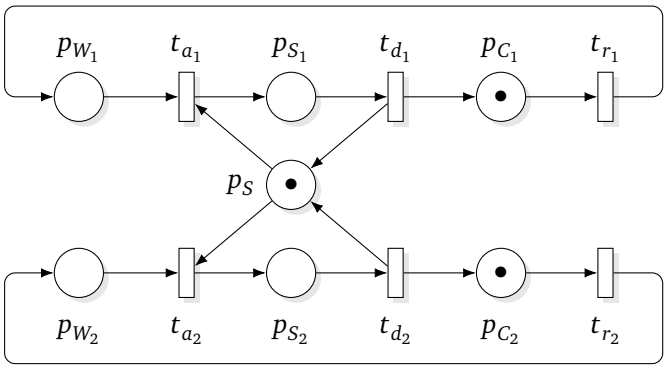


Figure 2.2 The *SharedResource* Petri net model.

Running example 2.2 In Figure 2.2 we introduce the *SharedResource* model which will serve as a running example throughout this report.

The model consists of a single shared resource S and two consumers. Each consumer can be in one of the following states: C_i (calculating locally), W_i (waiting for resource) and S_i (using shared resource). The transitions r_i (request resource), a_i (acquire resource) and d_i (done) correspond to behaviors of the consumers. The net is 1-bounded, therefore it has finite RS.

The Petri net model allows the verification of safety properties, e.g. we can show that there is mutual exclusion – $M(S_1) + M(S_2) \leq 1$ for all reachable markings – or that deadlock cannot occur.

2.1.1 Petri nets extended with inhibitor arcs

Inhibitor arcs are widely used extensions of Petri nets that can disable transitions even when the firing rule defined in Section 2.1 is satisfied. This modification gives Petri nets expressive power equivalent to Turing machines [17].

Definition 2.3 A *Petri net with inhibitor arcs* is a 3-tuple $PN_I = (PN, I, W_I)$, where

- $PN = (P, T, F, W, M_0)$ is a Petri net;
- $I \subseteq P \times T$ is the set of inhibitor arcs;
- $W_I : I \to \mathbb{N}^+$ is the inhibitor arc weight function.

Let ${}^\circ t = \{p : (p,t) \in I\}$ denote the set of inhibitor places of the transition t. The enablement rule for Petri nets with inhibitor arcs can be formalized as

$$M[t\rangle \iff M(p) \geq W(p,t) \text{ for all } p \in {}^\bullet t \text{ and } M(p) < W_I(p,t) \text{ for all } p \in {}^\circ t.$$

The firing rule (2.1) remains unchanged.

2.2 Continuous-time Markov chains

Continuous-time Markov chains are mathematical tools for describing the behavior of systems in countinous time where the stochastic behavior of the system only depends on its current state.

Definition 2.4 A *Continuous-time Markov Chain* (CTMC) $X(t) \in S, t \geq 0$ over a finite or countable infinite state space $S = \{0, 1, \ldots, n-1\}$ is a continuous-time random process with the *Markovian* or memoryless property:

$$\mathbb{P}(X(t_k) = x_k \mid X(t_{k-1}) = x_{k-1}, X(t_{k-2}) = x_{k-2}, \ldots, X(t_0) = x_0) \\ = \mathbb{P}(X(t_k) = x_k \mid X(t_{k-1}) = x_{k-1}),$$

where $t_0 \leq t_1 \leq \cdots \leq t_k$ and $X(t_k)$ is a continuous random variable denoting the current state of the CTMC at time t_k. A CTMC is said to be *time-homogenous* if it also satisfies

$$\mathbb{P}(X(t_k) = x_k \mid X(t_{k-1}) = x_{k-1}) = \mathbb{P}(X(t_k - t_{k-1}) = x_k \mid X(0) = x_{k-1}),$$

i.e. it is invariant to time shifting.

In this report we will restrict our attention to time-homogenous CTMCs over finite state spaces. The state probabilities of these stochastic processes at time t form a finite-dimensional vector $\boldsymbol{\pi}(t) \in \mathbb{R}$,

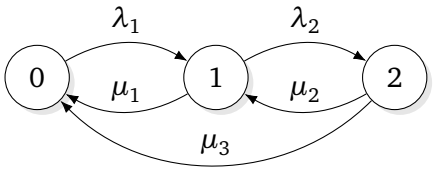
$$\pi(t)[x] = \mathbb{P}(X(t) = x)$$

that satisfies the differential equation

$$\frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t} = \boldsymbol{\pi}(t)Q \tag{2.2}$$

for some square matrix Q. The matrix Q is called the *infinitesimal generator matrix* of the CTMC and can be interpreted as follows:

- The diagonal elements $q[x, x] < 0$ describe the holding times of the CTMC. If $X(t) = x$, the *holding time* $h_x = \inf\{h > 0 : X(t + h) \neq x\}$ spent in state x is exponentially distributed with rate $\lambda_x = -q[x, x]$. If $q[x, x] = 0$, then no transitions are possible from state x and it is said to be *absorbing*.



$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} -\lambda_1 & \lambda_1 & 0 \\ \mu_1 & -\lambda_2 - \mu_1 & \lambda_2 \\ \mu_3 & \mu_2 & -\mu_2 - \mu_3 \end{pmatrix} \end{array}$$

Figure 2.3 Example CTMC with 3 states and its generator matrix.

- The off-diagonal elements $q[x, y] \geq 0$ describe the state transitions. In state x the CTMC will jump to state y at the next state transition with probability $-q[x, y]/q[x, x]$. Equivalently, there is expontentially distributed countdown in the state x for each $y : q[x, y] > 0$ with *transition rate* $\lambda_{xy} = q[x, y]$. The first countdown to finish will trigger a state change to the corresponding state y. Thus, the CTMC is a transition system with exponentially distributed timed transitions.

- Elements in each row of Q sum to 0, hence it satisfies $Q\mathbf{1}^{\mathrm{T}} = \mathbf{0}^{\mathrm{T}}$.

For more algebraic properties of infinitesimal generator matrices, we refer to Plemmons and Berman [51] and Stewart [66].

A state y is said to be *reachable* from the state x ($x \rightsquigarrow y$) if there exists a sequence of states

$$x = z_1, z_2, z_3, \ldots, z_{k-1}, z_k = y$$

such that $q[z_i, z_{i+1}] > 0$ for all $i = 1, 2, \ldots, k-1$. If y is reachable from x for all $x, y \in S$ y, the Markov chain is said to be *irreducible*.

The *steady-state probability distribution* $\boldsymbol{\pi} = \lim_{t\to\infty} \boldsymbol{\pi}(t)$ exists and is independent from the *initial distribution* $\boldsymbol{\pi}(0) = \boldsymbol{\pi}_0$ if and only if the finite CTMC is irreducible. The steady-state distribution is a stationary solution of eq. (2.2), therefore it satisfies the linear equation

$$\frac{\mathrm{d}\boldsymbol{\pi}}{\mathrm{d}t} = \boldsymbol{\pi} Q = \mathbf{0}, \quad \boldsymbol{\pi}\mathbf{1}^{\mathrm{T}} = 1. \tag{2.3}$$

Example 2.3 Figure 2.3 shows a CTMC with 3 states. The transitions from state 0 to 1 and from 1 to 2 are associated with exponentially distributed countdowns with rates λ_1 and λ_2 respectively, while transitions in the reverse direction have rates μ_1 and μ_2. The transition form state 2 to 0 is also possible with rate μ_3.

The rows (corresponding to source states) and columns (destination states) of the infinitesimal generator matrix Q are labeled with the state numbers. The diagonal element $q[1, 1]$ is $-\lambda_2 - \mu_1$, hence the holding time in state 1 is exponentially distributed with rate $\lambda_2 + \mu_1$. The transition to 0 is taken with probability

$-q[1,0]/q[1,1] = \mu_1/(\lambda_2+\mu_1)$, while the transition to 2 is taken with probability $\lambda_2/(\lambda_2+\mu_1)$.

The CTMC is irreducible, because every state is reachable from every other state. Therefore, there is a unique steady-state distribution $\boldsymbol{\pi}$ independent from the initial distribution $\boldsymbol{\pi}_0$.

2.2.1 Markov reward models

Continuous-time Markov chains may be employed in the estimation of performance measures of models by defining *rewards* that associate *reward rates* with the states of a CTMC. The momentary reward rate random variable $R(t)$ can describe performance measures defined at a single point of time, such as resource utilization or probability of failure, while the *accumulated reward* random variable $Y(t)$ may correspond to performance measures associated with intervals of time, such as total downtime.

Definition 2.5 A *Continuous-time Markov Reward Process* over a finite state space $S = \{0, 1, \ldots, n-1\}$ is a pair $(X(t), \mathbf{r})$, where $X(t)$ is a CTMC over S and $\mathbf{r} \in \mathbb{R}^n$ is a *reward rate vector*.

The element $r[x]$ of the reward vector is a momentary reward rate in state x, therefore the reward rate random variable can be written as $R(t) = r[X(t)]$. The accumulated reward until time t is defined by

$$Y(t) = \int_0^t R(\tau)\,\mathrm{d}\tau.$$

The computation of the distribution function of Y(t) is a computationally intensive task (a summary is available at [53, Table 1]), while its mean, $\mathbb{E}\,Y(t)$, can be computed efficiently as discussed below.

Given the initial probability distribution vector $\boldsymbol{\pi}(0) = \boldsymbol{\pi}_0$ the expected value of the reward rate at time t can be calculated as

$$\mathbb{E}\,R(t) = \sum_{i=0}^{n-1} \pi(t)[i]\,r[i] = \boldsymbol{\pi}(t)\,\mathbf{r}^{\mathrm{T}}, \tag{2.4}$$

which requires the solution of the initial value problem [32, 57]

$$\frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t} = \boldsymbol{\pi}(t)\,Q, \quad \boldsymbol{\pi}(0) = \boldsymbol{\pi}_0 \tag{2.5}$$

to form the inner product $\mathbb{E}\,R(t) = \boldsymbol{\pi}(t)\,\mathbf{r}^{\mathrm{T}}$.

To obtain the expected steady-state reward rate (if it exists) the linear equation (2.3) should be solved instead of eq. (2.5) in order to acquire the steady-state probability vector $\boldsymbol{\pi}$. The computation of the reward value proceeds by eq. (2.4) in the same way as in transient analysis.

The expected value of the accumulated reward is

$$\begin{aligned}\mathbb{E}\,Y(t) &= \mathbb{E}\left[\int_0^t R(\tau)\,\mathrm{d}\tau\right] = \int_0^t \mathbb{E}[R(\tau)]\,\mathrm{d}\tau \\ &= \int_0^t \sum_{i=0}^{n-1} \pi(\tau)[i] r[i]\,\mathrm{d}\tau = \sum_{i=0}^{n-1} \int_0^t \pi(\tau)[i]\,\mathrm{d}\tau\, r[i] \\ &= \int_0^t \boldsymbol{\pi}(t)\,\mathrm{d}\tau\,\mathbf{r}^{\mathrm{T}} = \mathbf{L}(t)\,\mathbf{r}^{\mathrm{T}},\end{aligned}$$

where $\mathbf{L}(t) = \int_0^t \boldsymbol{\pi}(t)\,\mathrm{d}\tau$ is the accumulated probability vector, which is the solution of the initial value problem [57]

$$\frac{\mathrm{d}\mathbf{L}(t)}{\mathrm{d}t} = \boldsymbol{\pi}(t), \quad \frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t} = \boldsymbol{\pi}(t)Q, \quad \mathbf{L}(0) = \mathbf{0}, \quad \boldsymbol{\pi}(0) = \boldsymbol{\pi}_0. \tag{2.6}$$

Example 2.4 Let c_0, c_1 and c_2 denote operating costs per unit time associated with the states of the CTMC in Figure 2.3. Consider the Markov reward process $(X(t), \mathbf{r})$ with reward rate vector

$$\mathbf{r} = \begin{pmatrix} c_0 & c_1 & c_2 \end{pmatrix}.$$

The random variable $R(t)$ describes the momentary operating cost, while $Y(t)$ is the total operating expenditure until time t. The steady-state expectation of R is the average maintenance cost per unit time of the long-running system.

2.2.2 Sensitivity

Sensitivity analysis is widely used to assess the robustnes of information systems. Consider a reward process $(X(t), \mathbf{r})$ where both the infinitesimal generator matrix $Q(\boldsymbol{\theta})$ and the reward rate vector $\mathbf{r}(\boldsymbol{\theta})$ may depend on some *parameters* $\boldsymbol{\theta} \in \mathbb{R}^m$. The *sensitivity* analysis of the rewards $R(t)$ may reveal performance or reliability bottlenecks of the modeled system and help designers in achieving desired performance measures and robustnes values.

Definition 2.6 The *sensitivity* of the expected reward rate $\mathbb{E}R(t)$ to the parameter $\theta[i]$ is the partial derivative

$$\frac{\partial\, \mathbb{E}R(t)}{\partial\, \theta[i]}.$$

Considering parameters with high absolute sensitivity the model reacts to the changes of those parameters more prominently, therefore they can be promising directions of system optimization.

To calculate the sensivity of $\mathbb{E}R(t)$, the partial derivative of both sides of eq. (2.4) is taken, yielding

$$\frac{\partial \mathbb{E}R(t)}{\partial \theta[i]} = \frac{\partial \boldsymbol{\pi}(t)}{\partial \theta[i]}\mathbf{r}^{\mathrm{T}} + \boldsymbol{\pi}(t)\left(\frac{\partial \mathbf{r}}{\partial \theta[i]}\right)^{\mathrm{T}} = \mathbf{s}_i(t)\,\mathbf{r}^{\mathrm{T}} + \boldsymbol{\pi}(t)\left(\frac{\partial \mathbf{r}}{\partial \theta[i]}\right)^{\mathrm{T}}, \tag{2.7}$$

where $\mathbf{s}_i$ is the sensitivity of $\boldsymbol{\pi}$ to the parameter $\theta[i]$.

In transient analysis, the sensitivity vector $\mathbf{s}_i$ is the solution of the initial value problem

$$\frac{\mathrm{d}\mathbf{s}_i(t)}{\mathrm{d}t} = \mathbf{s}_i(t)Q + \boldsymbol{\pi}(t)V_i, \quad \frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t} = \boldsymbol{\pi}_i(t)Q, \quad \mathbf{s}_i(0) = \mathbf{0}, \quad \boldsymbol{\pi}(0) = \boldsymbol{\pi}_0,$$

where $V_i = \partial Q(\theta)/\partial\theta[i]$ is the partial derivative of the generator matrix [55]. A similar initial value problem can be derived for the sensitivity of $\mathbf{L}(t)$ and $Y(t)$.

To obtain the sensitivity $\mathbf{s}_i$ of the steady-state probability vector $\boldsymbol{\pi}$, the system of linear equations

$$\mathbf{s}_i Q = -\boldsymbol{\pi}V_i, \quad \mathbf{s}_i\mathbf{1}^T = 0 \tag{2.8}$$

is solved [8].

Another type of sensitivity analysis considers *unstructured* small perturbations of the infinitesimal generator matrix Q instead of dependecies on parameters [30, 37]. This latter, unstructured analysis may be used to study the numerical stability and conditioning of the solutions of the Markov chain.

2.2.3 Time to first failure

Computing the first time of a system failure (provided it was fully operational when it was started) has many applications in reliability engineering.

Let $D \subsetneq S$ be a set of *failure states* of the CTMC $X(t)$ and $U = S \setminus D$ be a set of operating states. We will assume without loss of generality that $U = \{0, 1, \ldots, n_U - 1\}$ and $D = \{n_U, n_U + 1, \ldots, n - 1\}$.

The matrix

$$Q_{UD} = \begin{pmatrix} Q_{UU} & \mathbf{q}_{UD}^{\mathrm{T}} \\ \mathbf{0} & 0 \end{pmatrix}$$

is the infinitesimal generator of a CTMC $X_{UD}(t)$ in which all the failures states D were merged into a single state n_U and all outgoing transitions from D were removed. The

matrix Q_{UU} is the $n_U \times n_U$ upper left submatrix of Q, while the vector $\mathbf{q}_{UD} \in \mathbb{R}^{n_U}$ is defined as

$$q_{UD}[x] = \sum_{y \in D} q[x, y].$$

If the initial distribution $\boldsymbol{\pi}_0$ is 0 for all failure states (i.e. $\pi_0[x] = 0$ for all $x \in D$), the *Time to First Failure*

$$TFF = \inf\{t \geq 0 : X(t) \in D\} = \inf\{t \geq 0 : X_{UD}(t) = n_U\}$$

is *phase-type distributed* with parameters $(\boldsymbol{\pi}_U, Q_{UU})$ [50], where $\boldsymbol{\pi}_U$ is the vector containing the first n_U elements of $\boldsymbol{\pi}_0$. In particular, the *Mean Time to First Failure* is computed as follows:

$$MTFF = \mathbb{E}[TFF] = -\boldsymbol{\pi}_U Q_{UU}^{-1} \mathbf{1}^{\mathrm{T}}. \tag{2.9}$$

The probability of a D'-mode failure ($D' \in D$) is

$$\mathbb{P}(X(TFF_{+0}) = y) = -\boldsymbol{\pi}_D U Q_{UU}^{-1} \mathbf{q}_{UD'}^{\mathrm{T}}, \tag{2.10}$$

where $\mathbf{q}_{UD'} \in \mathbb{R}^{n_U}$, $q_{UD'}[x] = \sum_{y \in D'} q[x, y]$ is the vector of transition rates from operational states to failure states D'.

2.3 Stochastic Petri nets

While reward processes based on continuous-time Markov chains allow the study of dependability or reliability measurements, the explicit specification of stochastic processes and rewards is often cumbersome. More expressive formalisms include queueing networks, stochastic process algebras such as PEPA [24, 31], Stochastic Automata Networks [27] and Stochastic Petri Nets (SPN).

Stochastic Petri Nets extend Petri nets by assigning random exponentially distributed random delays to transitions [43]. After the delay associated with an enabled transition is elapsed the transition fires *atomically* are transitions delays are reset.

Definition 2.7 A Stochastic Petri Net is a pair $SPN = (PN, \Lambda)$, where PN is a Petri net (P, T, F, W, M_0) and $\Lambda : T \to \mathbb{R}^+$ is a transition rate function.

Likewise, a stochastic Petri net with inhibitor arcs is a pair $SPN_I = (PN_I, \Lambda)$, where PN_I is a Petri net with inhibitor arcs.

A finite CTMC can be associated with a bounded stochastic Petri net (with inhibitor arcs) as follows:

1. The reachable state space of the Petri net is explored. We associate consecutive natural numbers with the states such that the state space is

$$RS = \{M_0, M_1, M_2, \ldots, M_{n-1}\},$$

where M_0 is the initial marking. From now on, we will use markings $M_x \in RS$ and natural numbers $x \in \{0, 1, \ldots, n-1\}$ to refer to states of the model interchangably.

2. We define a CTMC $X(t)$ over the finite state space

$$S = \{0, 1, 2, \ldots, n-1\}.$$

The initial distribution vector will be set to

$$\boldsymbol{\pi}(0) = \boldsymbol{\pi}_0 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \end{pmatrix}$$

in the analysis step ($\pi_0[x] = \delta_{0,x}$).

3. The generator matrix $Q \in \mathbb{R}^{n \times n}$ encodes the possible state transitions of the Petri net and the associated transition rate $\Lambda(\cdot)$ as

$$q_O[x, y] = \sum_{\substack{t \in T \\ M_x[t\rangle M_y}} \Lambda(t) \quad \text{if } x \neq y, \tag{2.11}$$
$$q_O[x, x] = 0,$$
$$Q = Q_O - \text{diag}\{Q_O \mathbf{1}^{\mathrm{T}}\},$$

where the summation is done over all transition from the marking M_x to M_y, while Q_O and $Q_D = -\text{diag}\{Q_O \mathbf{1}^{\mathrm{T}}\}$ are the off-diagonal and diagonal parts of Q, respectively.

Running example 2.5 Figure 2.4 shows the SPN model for *SharedResouce*, which is the Petri net from Figure 2.2 on page 7 extended with exponential transition rates.

The transitions a_1, d_1, a_2 and d_2 have rates 1.0, 0.5, 1.0 and 1.1, respectively. The vector $\theta = (0.625, 1.25) \in \mathbb{R}^2$ of model parameters is introduced such that the transitions r_1 and r_2 have rates $1/\theta[0]$ and $1/\theta[1]$.

The reachable state space (Table 2.1) contains 8 markings which are mapped to the integers $S = \{0, 1, \ldots, 7\}$. The state space graph along with the transition rates

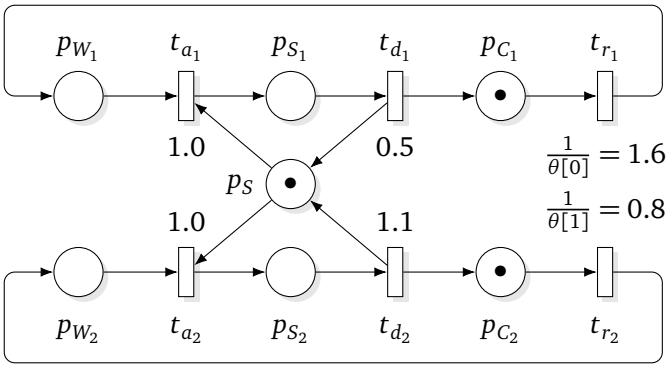


Figure 2.4 Example stochastic Petri net for the *SharedResource* model.

$RS =$

P:	S	C_1	W_1	S_1	C_2	W_2	S_2	
M_0	1	1	0	0	1	0	0	initial
M_1	1	0	1	0	1	0	0	client 1 waiting
M_2	1	1	0	0	0	1	0	client 2 waiting
M_3	1	0	1	0	0	1	0	1 waiting, 2 waiting
M_4	0	0	0	1	1	0	0	client 1 shared working
M_5	0	0	0	1	0	1	0	1 shared working, 2 waiting
M_6	0	1	0	0	0	0	1	client 2 shared working
M_7	0	0	1	0	0	0	1	1 waiting, 2 shared working

Table 2.1 Reachable state space of the *SharedResource* model.

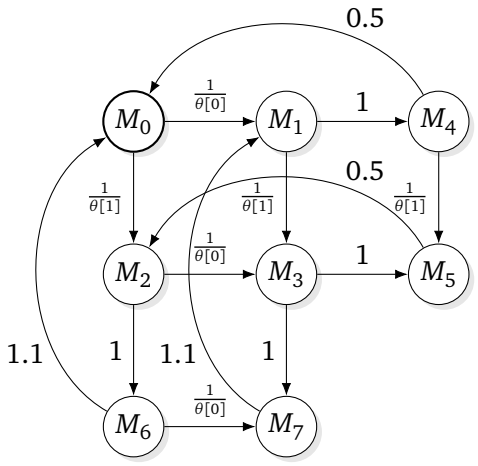


Figure 2.5 The CTMC associated with the *SharedResource* SPN model.

of the CTMC is shown in Figure 2.5. The generator matrix is (also depicting state indices):

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} * & \frac{1}{\theta[0]} & \frac{1}{\theta[1]} & 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & \frac{1}{\theta[1]} & 1 & 0 & 0 & 0 \\ 0 & 0 & * & \frac{1}{\theta[0]} & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & * & 0 & 1 & 0 & 1 \\ 0.5 & 0 & 0 & 0 & * & \frac{1}{\theta[1]} & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0 & * & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 & * & \frac{1}{\theta[0]} \\ 0 & 1.1 & 0 & 0 & 0 & 0 & 0 & * \end{pmatrix} \end{array},$$

where in each row the diagonal element is the negative of the sum of the other elemens so that $Q\mathbf{1}^{\mathrm{T}} = \mathbf{0}^{\mathrm{T}}$. The CTMC is irreducible, therefore it has a well-defined steady-state distribution.

Extensions of stochastic Petri nets include transitions with general or phase-type delay distributions [42, 44], Generalized Stochastic Petri Nets (GSPN) with immediate transitions [45, 67] and Deterministic Stochastic Petri Nets (DSPN) with deterministic firing delays [59]. Among these, only phase-type distributed delays and GSPNs can be handled with purely Markovian analysis. Stochastic Well-formed Nets (SWN) are a class of colored Petri nets especially amenable to stochastic analysis [16]. Stochastic Activity Networks (SAN) also allow colored places, moreover, they introduce input and output gates for more flexible modeling [36].

2.3.1 Stochastic reward nets

The stochastic reward net formalism is an extension of stochastic Petri nets that allows the definition of performance measures on the net level for use in the stochastic analysis workflow.

Definition 2.8 A *Stochastic Reward Net* is a triple $SRN = (SPN, rr, ir)$, where SPN is a stochastic Petri net, $rr : \mathbb{N}^P \to \mathbb{R}$ is a *rate reward function* and $ir : T \times \mathbb{N}^P \to \mathbb{R}$ is an *impulse reward* function. A stochastic Reward net with inhibitor arcs is a triple $SRN_I = (SPN_I, rr, ir)$, where SPN_I is a stochastic Petri net with inhibitor arcs.

The rate reward $rr(M)$ is the reward gained per unit time in marking M, while $ir(t, M)$ is the reward gained when the transition t fires in marking M.

If $ir(t,M) \equiv 0$, the SRN is equivalent to the Markov reward process $(X(t), \mathbf{r})$, where $X(t)$ is the CTMC associated with the stochastic Petri net and

$$\mathbf{r} \in \mathbb{R}^n, \quad r[x] = rr(M_x).$$

If there are impulse rewards, exact calculation of the expected reward rate $\mathbb{E}R(t)$ and expected accumulated reward $\mathbb{E}Y(t)$ can be performed on reward process $(X, \mathbf{r})$,

$$r[x] = rr(M_x) + \sum_{t \in T, M_x[t\rangle} \Lambda(t)\, ir(t, M_x),$$

where the summation is taken over all enabled transitions [20]. In general, the distribution of the accumulated reward $Y(t)$ cannot be derived by this method [54].

Running example 2.6 The SRN model

$$rr_1(M) = M(p_{S_1}) + M(p_{S_2}), \quad ir_1(t,M) \equiv 0 \tag{2.12}$$

describes the utilization of the shared resouce in the *SharedResouce* SPN (Figure 2.4 on page 15). $R_1(t) = 1$ if the resource is allocated, hence $\mathbb{E}R_1(t)$ is the probability that the resource is in use at time t, while $Y(t)$ is the total usage time until t.

Another reward structure

$$rr_2(M) \equiv 0, \quad ir_2(t,M) = \begin{cases} 1 & \text{if } t \in \{t_{r_1}, t_{r_2}\}, \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

counts the completed calculations, which are modeled by tokens leaving the places C_1 and C_2. The exprected steady-state reward rate $\lim_{t\to\infty} \mathbb{E}R(t)$ equals the number of calculations per unit time in a long-running system, while $Y(t)$ is the number of calculations performed until time t.

The reward vectors associated with these SRNs are

$$\begin{array}{rccccccccc} & 0 & & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \mathbf{r}_1 = \big(& 0 & & 0 & 0 & 0 & 1 & 1 & 1 & 1 \big), \\ \mathbf{r}_2 = \big(& \frac{1}{\theta[0]} + \frac{1}{\theta[1]} & & \frac{1}{\theta[1]} & \frac{1}{\theta[0]} & 0 & \frac{1}{\theta[1]} & 0 & \frac{1}{\theta[0]} & 0 \big). \end{array}$$

2.3.2 Superposed stochastic Petri nets

In this section we define the base formalism of the decomposition algorithm introduced in Chapter 4.

Definition 2.9 A *Superposed Stochastic Petri Net* (SSPN) is a pair $SSPN = (SPN, \mathcal{P})$, where $\mathcal{P} = \{P^{(0)}, P^{(1)}, \ldots, P^{(J-1)}\}$ is the partitioning of the set of places and $P = P^{(0)} \cup P^{(1)} \cup \cdots \cup P^{(J-1)}$ [23]. Superposed stochastic Petri nets with inhibitor arcs $SSPN_I = (SPN_I, \mathcal{P})$ are defined analogously.

The jth *local net* $LN^{(j)} = ((P^{(j)}, T^{(j)} = T_L^{(j)} \cup T_S^{(j)}, F^{(j)}, W^{(j)}, M_0^{(j)}), \Lambda^{(j)})$ can be constructed as follows:

- $P^{(j)}$ is the corresponding set from the partitioning of the original net.
- $T^{(j)}$ contains the local transition $T_L^{(j)}$ and synchronization transitions $T_S^{(j)}$.

  A transition is *local* to $LN^{(j)}$ if it only affects places in $P^{(j)}$, that is,

  $$T_L^{(j)} = \{t \in T : {}^\bullet t \cup t^\bullet \subseteq P^{(j)}\}. \tag{2.14}$$

  No transition may be local to more than one local net.

  A transition *synchronizes* with $LN^{(j)}$ if it affects some places in $P^{(j)}$ but it is not local to $LN^{(j)}$,

  $$T_S^{(j)} = \{t \in T : ({}^\bullet t \cup t^\bullet) \cap P^{(j)} \neq \emptyset\} \setminus T_L^{(j)}. \tag{2.15}$$

- The relation $F^{(j)}$ and the functions $W^{(j)}$, $M_0^{(j)}$, $\Lambda^{(j)}$ are the appropriate restrictions of the original structures, $F^{(j)} = F \cap ((P^{(j)} \times T^{(j)}) \cup (T^{(j)} \times J^{(j)}))$, $W^{(j)} = W|_{F^{(j)}}$, $M_0^{(j)} = M_0|_{P^{(j)}}$, $\Lambda^{(j)} = M_0|_{T^{(j)}}$.

If there are inhibitor arcs in $SSPN_I$, inhibitor arcs must be considered when a local net $LN_I^{(j)}$ is constructed. The set ${}^\bullet t \cup t^\bullet$ is replaced with ${}^\bullet t \cup t^\bullet \cup {}^\circ t$ in eqs. (2.14) and (2.15) so that the enabling of local transitions only depends on the marking of places in $P^{(j)}$ and only places in $P^{(j)}$ may be affected upon firing. In addition, the inhibitor arc relation and weight function are restricted as $I^{(j)} = I \cap (P^{(j)} \cap T^{(j)})$, $W_I^{(j)} = W_I|_{I^{(j)}}$.

The set of all synchronization transitions is denoted as $T_S = \bigcup_{j=0}^{J-1} T_S^{(j)}$. The *support* of the transition $t \in T$ is the set of components it is adjacent to, $\operatorname{supp} t = \{j : t \in T^{(j)}\}$.

Running example 2.7 Figure 2.6 shows a possible partitioning of the *Shared-Resource* SPN into a SSPN. The components $P^{(0)}$ and $P^{(1)}$ model the two consumers, while $P^{(2)}$ contains the unallocated resource S.

The transitions r_1 and r_2 are local to $LN^{(0)}$ and $LN^{(1)}$, respectively, while a_1, d_1, a_2 and d_2 synchronize $LN^{(2)}$ and the local net associated with their consumers.

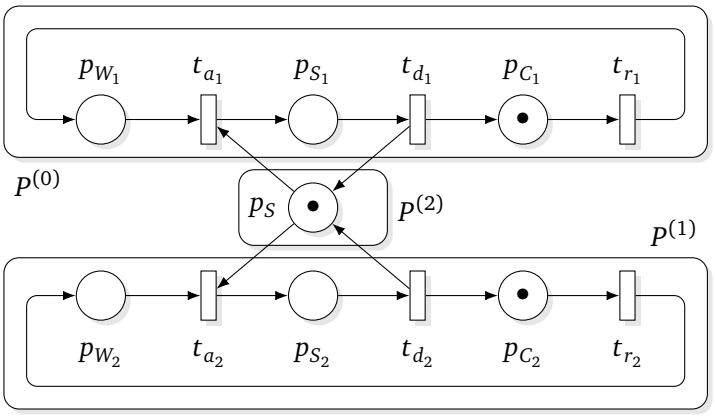


Figure 2.6 A partitioning of the *SharedResource* Petri net.

The *local reachable state space* $RS^{(j)}$ of $LN^{(j)}$ is the the set of markings beloning to the state space RS of the original net restricted to the places $P^{(j)}$ (duplicates removed),

$$RS^{(j)} = \{M^{(j)} : M \in RS, M^{(j)} = M|_{P^{(j)}}\}.$$

This is a *subset* of the reachable state space of $LN^{(j)}$, in particular, $RS^{(j)}$ is always finite if RS is finite, even if $LN^{(j)}$ is not bounded. Analysis techniques for generating local state spaces include *partial P-invariants* [13] and explicit projection of global reachable markings [10].

The *potential state space* PS of an SSPN is the Cartesian product of the local reachable state spaces of its components

$$PS = RS^{(0)} \times RS^{(1)} \times \cdots \times RS^{(J-1)},$$

which is a (possibly not proper) superset of the global reachable state space RS.

We will associate the natural numbers $S^{(j)} = \{0, 1, \ldots, n_j - 1\}$ with the local reachable markings $RS^{(j)} = \{M_0, M_1, \ldots, M_{n_j-1}\}$ to aid the construction of Markov chains and use them interchangably. The notation

$$M = \mathbf{x} = (x^{(0)}, x^{(1)}, \ldots, x^{(J-1)}) \tag{2.16}$$

refers to the global state $\mathbf{x}$ composed from the local markings $x^{(j)}$, i.e. the marking

$$M(p) = M^{(j)}_{x^{(j)}}(p), \quad \text{if } p \in P^{(j)},$$

which is the union of the local markings $M^{(0)}_{x^{(0)}}, M^{(1)}_{x^{(1)}}, \ldots, M^{(J-1)}_{x^{(J-1)}}$.

$$RS^{(0)} = \left\{ \begin{array}{|c|ccc|} \hline P: & C_1 & W_1 & S_1 \\ \hline M_0^{(0)} & 1 & 0 & 0 \\ M_1^{(0)} & 0 & 1 & 0 \\ M_2^{(0)} & 0 & 0 & 1 \\ \hline \end{array} \right\},$$

$$RS^{(1)} = \left\{ \begin{array}{|c|ccc|} \hline P: & C_2 & W_2 & S_2 \\ \hline M_0^{(1)} & 1 & 0 & 0 \\ M_1^{(1)} & 0 & 1 & 0 \\ M_2^{(1)} & 0 & 0 & 1 \\ \hline \end{array} \right\}, \quad RS^{(2)} = \left\{ \begin{array}{|c|c|} \hline P: & S \\ \hline M_0^{(2)} & 1 \\ M_1^{(2)} & 0 \\ \hline \end{array} \right\}$$

Table 2.2 Local reachable markings of the *SharedResouce* SSPN from Figure 2.6.

Running example 2.8 The local reachable markings of the *SharedResource* SSPN are enumerated in Table 2.2.

The transitions d_1 and d_2 are always enabled in $LN^{(2)}$ because all their input places are located in other components, thus $LN^{(2)}$ is an unbounded Petri net. Despite this, $RS^{(2)}$ is finite, because it only contains the local markings which are reachable in the original net.

The potential state space PS contains $3 \cdot 3 \cdot 2 = 18$ potential markings, although only 8 are reachable (Table 2.1 on page 15). For example, the marking $(2, 2, 0)$ is not reachable, as it would violate mutual exclusion.

2.4 Kronecker algebra

Kronecker algebra defines the building blocks of the decomposition algorithm being introduced in Chapter 4.

Definition 2.10 The *Kronecker product* of matrices $A \in \mathbb{R}^{n_1 \times m_1}$ and $B \in \mathbb{R}^{n_2 \times m_2}$ is the matrix $C = A \otimes B \in \mathbb{R}^{n_1 n_2 \times m_1 m_3}$, where

$$c[i_1 n_1 + i_2, j_1 m_1 + j_2] = a[i_1, j_1] b[i_2, j_2].$$

Some properties of the Kroncker product are

1. Associativity:

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C,$$

which makes Kronecker products of the form $A^{(0)} \otimes A^{(1)} \otimes \cdots \otimes A^{(J-1)}$ well-defined.

2. Distributivity over matrix addition:

$$(A+B) \otimes (C+D) = A \otimes C + B \otimes C + A \otimes D + B \otimes D,$$

3. Compatibility with ordinary matrix multiplication:

$$(AB) \otimes (CD) = (A \otimes C)(B \otimes D),$$

in particular,

$$A \otimes B = (A \otimes I_2)(I_1 \otimes B)$$

for identity matrices I_1 and I_2 with appropriate dimensions.

We will occasionally employ multi-index notation to refer to elements of Kronecker product matrices. For example, we will write

$$b[\mathbf{x}, \mathbf{y}] = b[(x^{(0)}, x^{(1)}, \ldots, x^{(J-1)}), (y^{(0)}, y^{(1)}, \ldots, y^{(J-1)})] = a^{(0)}[x^{(0)}, y^{(0)}] a^{(1)}[x^{(1)}, y^{(1)}] \cdots a^{(J-1)}[x^{(J-1)}, y^{(J-1)}],$$

where $\mathbf{x} = (x^{(0)}, x^{(1)}, \ldots, x^{(J-1)})$, $\mathbf{y} = (y^{(0)}, y^{(1)}, \ldots, y^{(J-1)})$ and B is the J-way Kronecker product $A^{(0)} \otimes A^{(1)} \otimes \cdots \otimes A^{(J-1)}$.

Definition 2.11 The *Kronecker sum* of matrices $A \in \mathbb{R}^{n_1 \times m_1}$ and $B \in \mathbb{R}^{n_2 \times m_2}$ is the matrix $C = A \oplus B \in \mathbb{R}^{n_1 n_2 \times m_1 m_3}$, where

$$C = A \otimes I_2 + I_1 \otimes B,$$

where $I_1 \in \mathbb{R}^{n_1 \times m_1}$ and $I_2 \in \mathbb{R}^{n_2 \times m_2}$ are identity matrices.

Example 2.9 Consider the matrices

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \qquad B = \begin{pmatrix} 0 & 1 \\ 2 & 0 \end{pmatrix}.$$

Their Kronecker product is

$$A \otimes B = \begin{pmatrix} 1 \cdot 0 & 1 \cdot 1 & 2 \cdot 0 & 2 \cdot 1 \\ 1 \cdot 2 & 1 \cdot 0 & 2 \cdot 2 & 2 \cdot 0 \\ 3 \cdot 0 & 3 \cdot 1 & 4 \cdot 0 & 4 \cdot 1 \\ 3 \cdot 2 & 3 \cdot 0 & 4 \cdot 2 & 4 \cdot 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 2 \\ 2 & 0 & 4 & 0 \\ 0 & 3 & 0 & 4 \\ 6 & 0 & 8 & 0 \end{pmatrix},$$

while their Kronecker sum is

$$A \oplus B = \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 0 & 4 & 0 \\ 0 & 3 & 0 & 4 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 2 & 0 \\ 2 & 1 & 0 & 2 \\ 3 & 0 & 4 & 1 \\ 0 & 3 & 2 & 4 \end{pmatrix}.$$

Chapter 3

Overview of the approach

3.1 General workflow

The tasks performed by stochastic analysis tools that operate on higher level formalisms can be often structured as follows (Figure 3.1):

1. *State space exploration.* The reachable state space of the higher level model, for example stochastic automata network or stochastic Petri net is explored to enumerate the possible behaviors of the model S. If the model is hierarchically partitioned, this step includes the exploration of the local state spaces of the component as well as the possible global combinations of states.

   If the set of reachable states is infinite, only special algorithms, e.g. matrix geometric methods [35] may be employed later in the workflow. In this work, we restrict our attention to finite cases.

2. *Descriptor generation.* The infinitesimal generator matrix Q of the Markov chain $X(t)$ defined over S is built. If the analyzed formalism is a Markov chain, Q is readily given. Otherwise, this matrix contains the transition rates between reachable states, which are obtained by evaluating rate expressions given in the model.

3. *Numerical solution.* Numerical algorithms are ran on the matrix Q for steady-state solutions $\boldsymbol{\pi}$, transient solutions $\boldsymbol{\pi}(t)$, $\mathbf{L}(t)$ or MTFF measures.

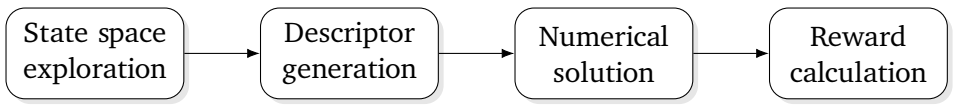


Figure 3.1 The general stochastic analysis workflow.

4. *Reward calculations.* The studied performance measures are calculated from the output of the previous step. This includes calculation of steady-state and transient rewards and sensitivities of the rewards. Additional algebraic manipulations (for example, the calculation of the ratio of an instantenous and accumulated reward) may be provided to the modeler for convenience.

In stochastic model checking, where the desired system behaviors are expressed in stochastic temporal logics [1, 7], these analytic steps are called as subrouties to evaluate propositions. In the sythesis and optimization of stochastic models [15], the workflow is executed as part of the fitness functions.

3.1.1 Challenges

The implementation of the stochastic analysis workflow poses several challenges.

Handling of large models is difficult due to the phenomenon of "state space explosion". As the size of the model grows, including the number of components, the number of reachable spaces can grow exponentially.

Methods such as the *saturation* algorithm [19] were developed to efficiently explore and represent large state spaces. However, in stochastic analysis, the generator matrix Q and several vectors of real numbers with lengths equal to the state space size must be stored in addition to the state space. This neccessitates the use of further decomposition techniques for data storage.

The convergence of the numerical methods depends on the structure of the model and the applied matrix decomposition. In addition, the memory requirements of the algorithms may constrain the methods that can be employed. As various numerical algorithms for stochastic analysis tasks are known with different characteristics, it is important to allow the modeler to select the algorithm suitable for the properties of the model, as well as the decomposition method and hardware environment.

The vector operations and vector-matrix products that are preformed by the numerical algorithms can also be performed in multiple ways. For example, multiplications with matrices can be implemented either sequentially or in parallel. Large matrices benefit from parallelization, while for small matrices managing multiple tasks yields overhead. Distributed or GPU impelemtations are also possible, albeit they are missing from the current version of our framework.

3.2 Our workflow

Our implementation of the general stochastic analysis workflow is illustrated in Figure 3.2.

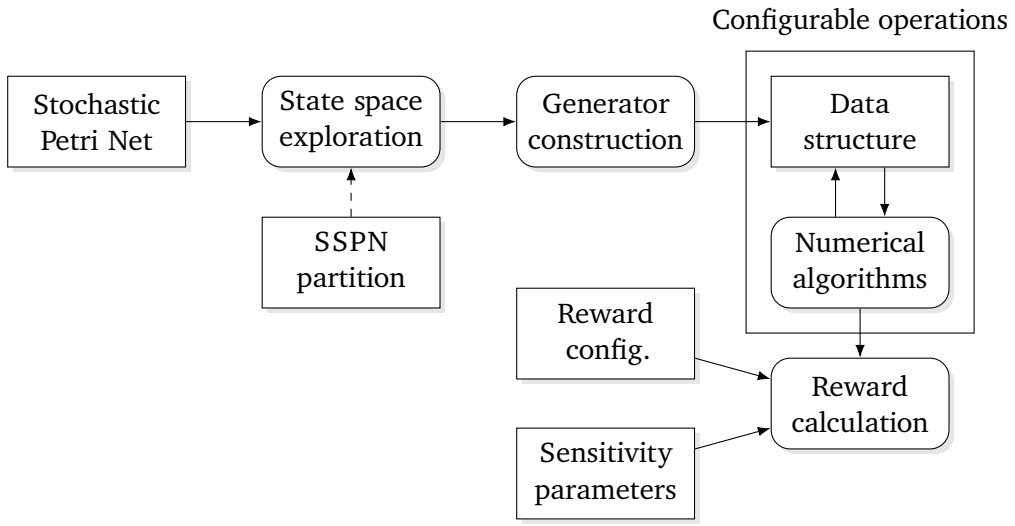


Figure 3.2 Configurable stochastic analysis workflow.

Table 3.1 Linear equation solvers supported by our framework.

	see	memory usage	parallel impl.	uses inner solver	block matrix
LU decomposition	p. 52	very high	–	–	–
Power method	p. 55	moderate	✓	–	✓
Jacobi over-relaxation	p. 56	moderate	✓	–	✓
Gauss–Seidel over-relaxation	p. 56	very low	–	–	✓
BiCGSTAB	p. 60	high	✓	–	✓
Group Jacobi	p. 58	moderate	✓	✓	required
Group Gauss–Seidel	p. 58	low	–	✓	required

The workflow is fully *configurable*, which means that the modeler may combine the available algorithms for the analysis steps arbitrarily. This is achieved by a layered architecture as shown in Figure 3.3.

- The model state space may be explored either by an explicit state space traversal, or by symbolic saturation [19]. As symbolic methods are usually much faster and use significantly less memory than explicit enumeration, they are the recommended approach for stochastic analysis. However the explicit algorithms are not sensitive to the structure of the model, they provide a robust solution as long as the state

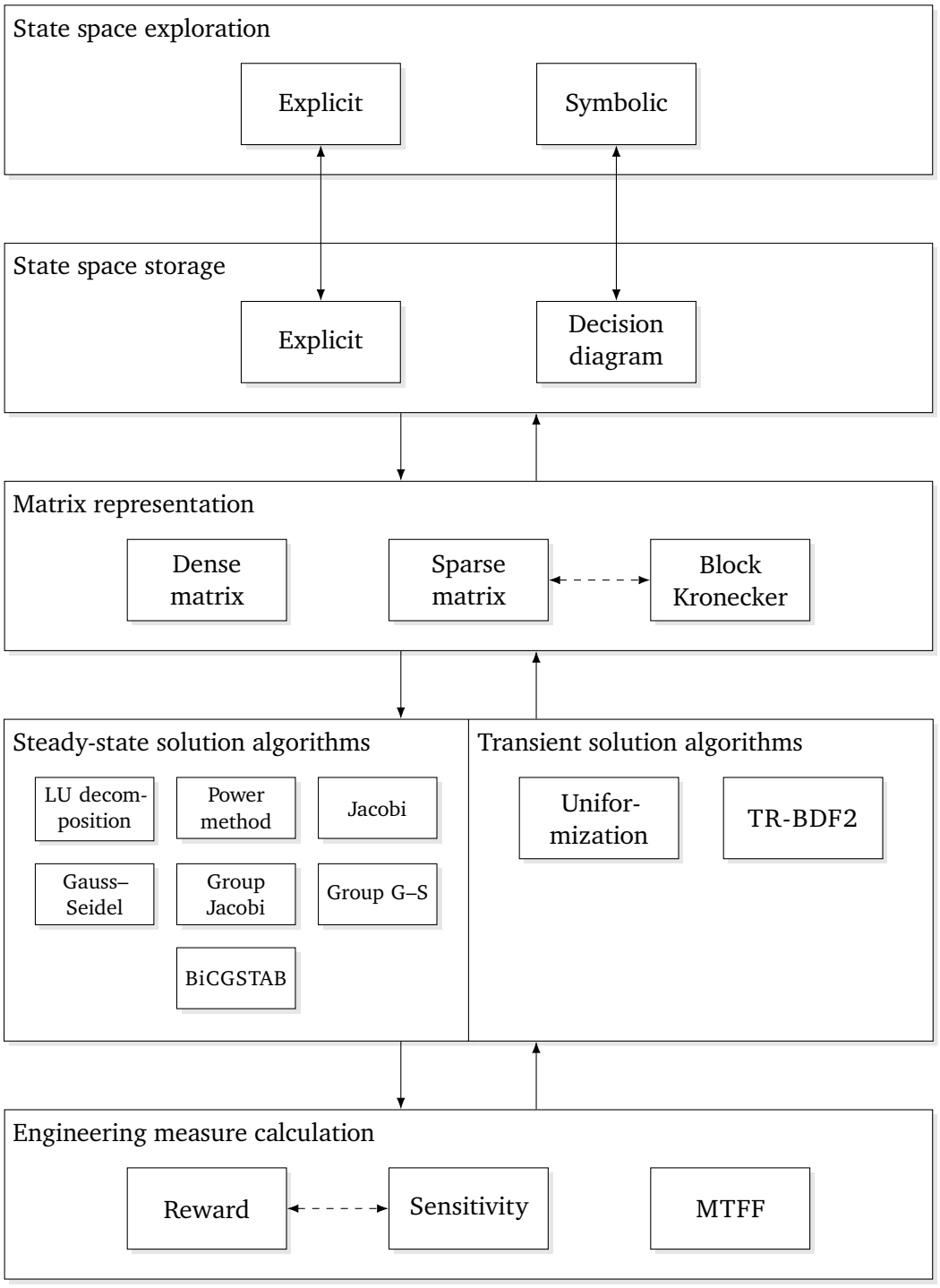


Figure 3.3 Architecture of the configurable stochastic analysis framework.

Table 3.2 Transient solvers supported by our framework.

	see	instantenous distribution	accumulated distribution	uses inner solver	block matrix
Uniformization	p. 61	✓	✓	–	✓
TR-BDF2	p. 62	✓	not impl.	✓	not impl.

space fits into memory. In addition they are provided for benchmarking and software redundancy reasons too.

The algorithms operating on a superposed SPN receive the model and a decomposition as an input. Partitions needed for the decomposition may be provided by the user as part of the model or generated on the fly.

- The generator matrix may be stored in sparse matrix representation or decomposed into block Kronecker form [13]. The matrix can be build from both explicitly or symbolically stored state spaces.

  To facilitate block Kronecker matrix generation, we propose a purely symbolic algorithm. The developed solution avoids any overheads of explicit state space operations.

- The resulting matrices, in a possibly decomposed form, are part of a specialized data structure. Extremely large matrices may be stored with the developed decomposition algorithms (e.g. linear combinations, Kronecker products, contatenations into block structures). The data structure defines generic vector and matrix operations, as well as more specific manipulations performed by stochastic analysis algorithms.

  State space exploration and generator matrix decomposition methods are presented in Chapter 4, including our theoretical and algorithmic contribution for block Kronecker decomposition.

- The execution of the operations on the data structures can be set at runtime. This allows the use of different implementations at the different stages of the workflow, or when different algorithms are employed to calculate multiple performance measures. Whenever possible, both sequential and parallel implementations of the most common operations are available for the supported datatypes.

- Several numerical algorithms are provided for steady-state and transient analysis of Markov chains. The user can select the algorithm most suitable for the model under study. The algorithm library supports the combination of the algorithms and data structures at different levels of computations. This allows us to fine

tune the numerical solution and solve every component with the most suitable algorithm.

Important considerations in solver selection are convergence properties and memory requirements. Matrix decompositions can reduce the storage space needed by the matrix Q by orders of magnitudes. We store all elements of probability vectors explicitly. Therefore, one should pay close attention to the number of temporary vectors used in the algorithm in order to avoid excessive memory consumption.

Numerical algorithms supported by our framework are discussed in Chapter 5. Linear equations solvers for steady-state CTMC analysis are shown in Table 3.1, while linear solver are shown in Table 3.2.

3.2.1 Formalisms

Our stochastic analysis framework supports models in the Stochastic Petri Net with inhibitor arcs formalism (see Definition 2.7 on page 13). Structured models are handled as Superposed Stochastic Petri Nets (see Definition 2.9 on page 18). However, any modeling formalism can be processed by integrating the appropriate state space exploration algorithms with the workflow.

Transition rates in the SPNs can be arbitrary algebraic expressions containing references to *sensitivity variables*. These variables correspond to the parameter vector $\boldsymbol{\theta}$ of the Markov chain sensitivity analysis. However, the rate expression may not depend on the marking of the net.

Reward structures are defined as Stochastic Reward Nets (see Definition 2.8 on page 16). An SRN reward structure may be specified by composing any *reward expressions* of the forms

1. (p, w), where $p \in P$ is a place and w is a *reward weight expression*. This reward expression is equivalent to a rate reward $rr(M) = M(p) \cdot w$, i.e. the value of w is multiplied by the numer of tokens on p.

2. (t, w), where $t \in T$ is a transition and w is a reward weight expression. This is equivalent to an impulse reward $ir(t, M) = w$ gained upon the firing of t.

3. $\varphi \rightarrow w$, where φ is a Computational Tree Logic (CTL) expression and w is a reward weight expression. This is equivalent to the rate reward $rr(M) = w$ if φ holds in M, 0 otherwise.

A reward weight expression is an algebraic expression that may refer to places and transition rates in the net. Refrences to places are replaced by the number of tokens upon evaluation. For example, the reward expression (p, w) may be written as $\mathsf{true} \rightarrow p \cdot w$ or $p > 0 \rightarrow p \cdot w$ using CTL.

Reward expressions with CTL are only allowed when symbolic state spaces representation is used, as CTL evaluation[1] is performed symbolically [22].

Running example 3.1 Consider the reward structures defined over the *SharedResource* Petri net from Running example 2.6 on page 17.

The utilization of the shared resource can be described by the reward expression

$$resourceUtilization = \{(p_{S_1}, 1), (p_{S_2}, 1)\},$$

which is equivalent to the SRN reward structure

$$rr_1(M) = M(p_{S_1}) + M(p_{S_2}), \quad ir_1(t, M) \equiv 0. \tag{2.12 revisited}$$

This can be also written as

$$resourceUtilization = \{p_{S_1} > 0 \vee p_{S_2} > 0 \to 1\}$$

using CTL, because the places S_1 and S_2 are 1-bounded in the *SharedResource* model.

Completed calculations are described by

$$completedCalculations = \{(t_{s_1}, 1), (t_{r_2}, 1)\},$$

which is equivalent to the reward structure

$$rr_2(M) \equiv 0, \quad ir_2(t, M) = \begin{cases} 1 & \text{if } t \in \{t_{r_1}, t_{r_2}\}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.13 revisited}$$

3.2.2 Analysis

The framework introducted in this paper supports the configurable stochastic analysis of the following problems:

- expected steady-state reward rates $\mathbb{E}R$ for any reward structure defined by reward expressions,
- expected transient reward rates $\mathbb{E}R(t)$ and accumulated rewards $\mathbb{E}Y(t)$,
- *complex rewards*, which are algebraic expressions of mean reward rates and accumulated rewards (e.g $1 + \mathbb{E}R(t)/\mathbb{E}Y(t)$),
- sensitivity of mean steady-state reward rates and complex rewards involving steady-state rates,

[1] The symbolic state space exploration and CTL evaluation component is currently provided by the PetriDotNet [26] tool.

- mean-time to failure *MTFF* and associated failure mode probabilities.

Configurable stochastic analysis provides the combination of multiple solver and representation algorithms for the efficient computation of the introduced properties.

3.2.3 Reward and sensitivity computation

Transition and reward rates are stored as algebraic expression trees in the input SPN models. Symbolic operations, such as partial differentiation may be performed exactly on the trees using algebraic laws, as the evaluation of the expressions can be delayed.

In reward and MTFF calculations, rate expressions are evaluated by replacing sensitivity parameters with their values before the matrix Q is composed. Thus, the elements of a matrix are not expression trees, but floating point numbers and matrix generation has to be performed only when sensitivity parameters are changed.

Reward weight expressions may refer to the token counts on places, therefore they must be evaluated for every marking individually. If a CTL reward expression $\varphi \to w$ is used, evaluation is skipped in markings where φ is false.

Steady-state sensitivity calculation, shown in Figure 3.4, is the most complicated post-processing in the workflow. Partial derivatives of the transition rate expressions and reward weight expressions are taken to calculate $\partial\, \mathbb{E}R/\partial\theta[i]$ using eqs. (2.7) and (2.8).

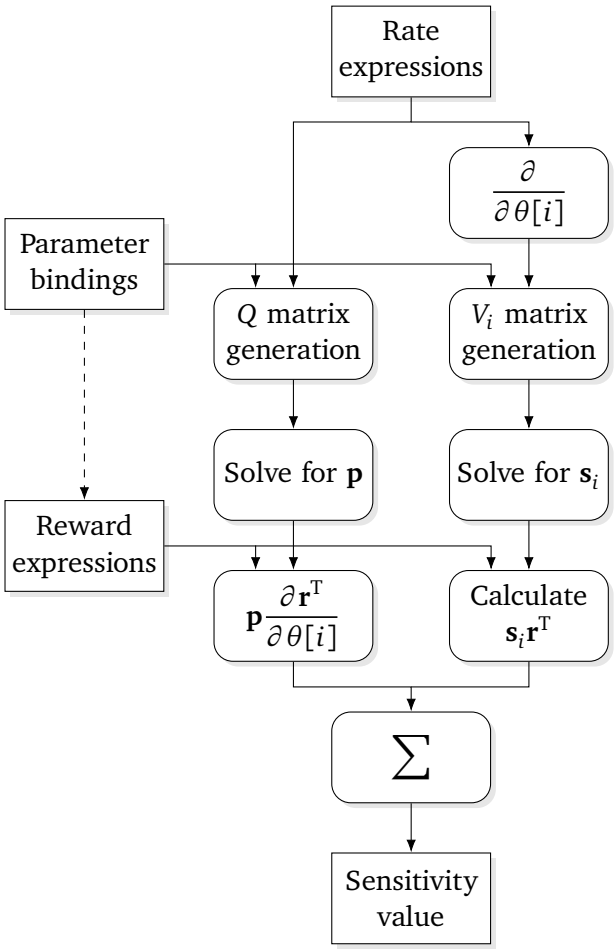


Figure 3.4 Reward and sensitivity calculation from expression tree inputs.

Chapter 4

Efficient generation and storage of continuous-time Markov chains

4.1 Explicit methods

4.1.1 Explicit state space and matrix construction

Explicit state space enumeration for Petri nets repeatedly applies the firing rule eq. (2.1) on page 6 starting from the initial marking M_0 until no new marking can be generated. At the end of the enumeration of the finite state space, all reachable markings $M_0 \rightsquigarrow M$ are discovered. We implemented detection of already encountered markings by hashing, while new markings are generated by breath-first search.

Given the finite state space of size $n = |RS|$ in an explicit form along with a bijection between the markings an the natural numbers $\{0, 1, \ldots, n-1\}$, the generator matrix Q can be directly created by Algorithm 4.1. The algorithm stores the transition rate $\Lambda(t)$ in Q for all pairs of reachable markings $M_x\,[t\rangle\,M_y$ and transitions $t \in T$.

The generator matrix requires $O(n^2)$ memory if a two-dimensional dense array format is used. Because firing a transition can only take the Petri net from a given marking M_x to a single target marking M_y in the SPN formalism, each column of Q may contain up to $|T|$ nonzero elements. Hence Q requires $O(|T|n)$ memory if a sparse format is chosen.

Unfortunately, both of these storage methods may be prohibitively costly for large models due to state space explosion. In addition, explicit enumeration of large RS may take an extreme amout of time.

Algorithm 4.1 Generator matrix construction from explicit state space.

Input: explicit state space RS, transitions T, transition rate function Λ
Output: generator matrix Q

```
allocate Q_O ∈ ℝ^{|RS|×|RS|}, d ∈ ℝ^{|RS|}
foreach y ∈ RS, t ∈ R do
    if there is a state x ∈ RS such that M_x [t⟩ M_y then
        q_D[x, y] ← q_D[x, y] + Λ(t)
d ← −Q_O 1^T
return Q_O + diag{d}
```

4.1.2 Block Kronecker generator matrices

Kronecker generator matrices

To alleviate the high memory requirements of Q, the Kronecker decomposition for a superposed SPN with J components expresses the infinitesimal generator matrix of the associated CTMC in the form

$$Q = Q_O + Q_D, \quad Q_O = \bigoplus_{j=0}^{J-1} Q_L^{(j)} + \sum_{t \in T_S} \Lambda(t) \bigotimes_{j=0}^{J-1} Q_t^{(j)}, \quad Q_D = -\operatorname{diag}\{Q_O \mathbf{1}^{\mathrm{T}}\}, \tag{4.1}$$

where Q_O and Q_D are the off-diagonal and diagonal parts of Q. The matrix

$$Q_L^{(j)} = \sum_{t \in T_L^{(j)}} \Lambda(t) Q_L^{(j)}$$

is the *local* transition matrix of the component j, while the matrix

$$Q_t^{(j)} \in \mathbb{R}^{n_j \times n_j}, \quad q_t^{(j)}[x^{(j)}, y^{(j)}] = \begin{cases} 1 & \text{if } x^{(j)}[t\rangle\, y^{(j)}, \\ 0 & \text{otherwise} \end{cases}$$

describes the effects of the transition t on $LN^{(j)}$. $Q_t^{(j)}$ has a nonzero element for every local state transition caused by t. If $j \notin \operatorname{supp} t$, $Q_t^{(j)}$ is an $n_j \times n_j$ identity matrix.

It can be seen that

$$\begin{aligned} q_O[\mathbf{x}, \mathbf{y}] &= \sum_{j=0}^{J-1} \sum_{t \in T_L^{(j)}} \Lambda(t)\, q_t^{(j)}[x^{(j)}, y^{(j)}] + \sum_{t \in T_S} \Lambda(t) \prod_{j=0}^{J-1} q_t^{(j)}[x^{(j)}, y^{(j)}] \\ &= \sum_{j=0}^{J-1} \sum_{\substack{t \in T_L^{(j)} \\ x^{(j)}[t\rangle y^{(j)}}} \Lambda(t) + \sum_{\in T_S, \mathbf{x}[t\rangle \mathbf{y}} \Lambda(t) = \sum_{t \in T, \mathbf{x}[t\rangle \mathbf{y}} \Lambda(t), \end{aligned} \tag{4.2}$$

which is the same as eq. (2.11) on page 14. Indeed, eq. (4.1) is a representation of the infinitesimal generator matrix.

The matrices $Q_L^{(j)}$ and $Q_t^{(j)}$ and the vector $-Q_O\mathbf{1}^{\mathrm{T}}$ together are usually much smaller than the full generator matrix Q even when stored in a sparse matrix form. Hence Kronecker decomposition may save a significant amount of storage at the expense of some computation time.

Unfortunately, the Kronecker generator Q is a $n_0 n_1 \cdots n_{J-1} \times n_0 n_1 \cdots n_{J-1}$ matrix, i.e. in encodes the state transitions in the potential state space PS instead of the reachable state space RS.

Potential Kronecker methods [11] perform computations with the $|PS| \times |PS|$ Q matrix and vectors of length $|PS|$. In addition to increasing storage requirements, this may lead to problems in some numerical solution algorithms, because the CTMC over PS is not neccessarily irreducible even if it is irreducible over RS.

In contrast, *actual Kronecker methods* [6, 11, 39] work with vectors of length $|RS|$. However, additional conversions must be performed between the actual dense indexing of the vectors and the potential sparse indexing of the Q matrix, which leads to implementation complexities and computational overhead.

A third approach, which we discuss in the next subsection, imposes a hierarchical structure on RS [4, 10, 13].

Macro state construction

The hierarchical structure of the reachable state space expresses RS as

$$RS = \bigcup_{\tilde{\mathbf{x}} \in \widetilde{RS}} \prod_{j=0}^{J-1} RS^{(j)}_{\tilde{x}^{(j)}}, \quad RS^{(j)} = \bigcup_{\tilde{x}^{(j)} \in \widetilde{RS}^{(j)}} RS^{(j)}_{\tilde{x}^{(j)}},$$

where $\widetilde{RS} = \{\tilde{0}, \tilde{1}_1, \ldots, \widetilde{\tilde{n}-1}\}$ a set of *global macro states*, $\widetilde{RS}^{(j)} = \{\tilde{0}^{(j)}, \tilde{1}^{(j)}, \ldots, \widetilde{\tilde{n}_j - 1}^{(j)}\}$ is the set of *local macro states* of $LN^{(j)}$, and $RS_x^{(j)} = \{0_x^{(j)}, 1_x^{(j)}, \ldots, (n_{j,x}-1)_x^{(j)}\}$ are the *local micro states* in the local macro state $\tilde{x}^{(j)}$. The product symbol denotes the composition of local markings, as in eq. (2.16) on page 19.

The local micro states form a partition $RS^{(j)} = \bigcup_{x \in \widetilde{RS}^{(j)}} RS_x^{(j)}$ of the state space of the jth SSPN component.

Construction of macro states is performed as follows [10]:

1. The equivalence relation $\sim^{(j)}$ is defined over $RS^{(j)}$ as

$$x^{(j)} \sim^{(j)} y^{(j)} \iff \{\hat{\mathbf{z}}^{(j)} : \mathbf{x} \in RS, z^{(j)} = x^{(j)}\} = \{\hat{\mathbf{z}}^{(j)} : \mathbf{x} \in RS, z^{(j)} = y^{(j)}\}, \tag{4.3}$$

   where $\hat{\mathbf{z}}^{(j)} = (z^{(0)}, \ldots, z^{(j-1)}, z^{(j+1)}, \ldots, z^{(J-1)})$, i.e. two local states are equivalent if they are reachable in the same combinations of local markings of the other

components. Therefore, the relation

$$\mathbf{x} \sim \mathbf{y} \quad \Longleftrightarrow \quad x^{(j)} \sim^{(j)} y^{(j)} \text{ for all } j = 0, 1, \ldots, J-1,$$

defined over PS, has the property that whether $\mathbf{x} \sim \mathbf{y}$, either both $\mathbf{x}$ and $\mathbf{y}$ are reachable (global) markings, or neither are.

2. Reachable local macro states are the partitions of $RS^{(j)}$ generated by $\sim^{(j)}$. A bijection $\widetilde{RS}^{(j)} \leftrightarrow RS^{(j)}/\sim^{(j)}$ is formed between the integers $0, 1, \ldots, \tilde{n}^{(j)} - 1$ and the local state partitions for each component $LN^{(j)}$.

3. The set of potential macro states is

$$\widetilde{PS} = \prod_{j=0}^{J-1} \widetilde{RS}^{(j)} \supseteq \widetilde{RS}$$

the Cartesian product of the local macro states. If macro state $\tilde{x} \in \widetilde{PS}$ contains a reachable state, all associated (micro) states are reachable, because $\widetilde{PS}$ is the partition $RS/\sim$ of RS generated by the relation $\sim$. Thus, $\widetilde{RS}$ is constructed by enumerating the reachable macro states in $\widetilde{PS}$. A bijection is formed between the reachable subset of $\widetilde{PS}$ and the integers $\tilde{0}, \tilde{1}, \ldots, \widetilde{\tilde{n}-1}$.

The pseudocode for this process is shown in Algorithm 4.2. The decomposition is extremely memory demanding due to the allocation of the bit vector $\mathbf{b}$ of length $|PS|$.

In [10], sorting the columns of B lexicographically was recommended to calculate the equality partition of the columns of B. In our implementation, we insert the rows of B into a bitwise trie and detect duplicates instead, so that no mapping between the original order and sorted ordering of columns needs to be maintained.

Running example 4.1 The macro states of the *RunningExample* SSPN model (Figure 2.6 on page 19) are obtained from its component state space (Table 2.2 on page 20) as follows:

1. The bit vector $\mathbf{b}$ is filled according to the reachable states RS,

$$\mathbf{b} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix},$$

where the mixed indices in small type refer to the states of the local nets $LN^{(0)}$, $LN^{(1)}$ and $LN^{(2)}$.

Algorithm 4.2 Hiearchical decomposition of the reachable state space into macro states by Buchholz [10].

Input: Reachable state space RS, reachable local state spaces $RS^{(j)}$
Output: Macro state space $\widetilde{RS}$, local macro state spaces $\widetilde{RS}^{(j)}$

allocate bit vector $\mathbf{b} \in \{0,1\}^{n_0 n_1 \cdots n_{J-1}}$ initialized with zeroes
foreach $\mathbf{x} \in RS$ **do**
  // Fill $\mathbf{b}$ with ones corresponding to reachable states
  $b[n_{J-1}n_{J-2}\cdots n_1 x^{(0)} + n_{J-1}n_{J-2}\cdots n_2 x^{(1)} + \cdots + n_{J-1}x^{(J-2)} + x^{(J-1)}] \leftarrow 1$
for $j \leftarrow 0$ **to** $J-1$ **do**
  Reshape $\mathbf{b}$ into matrix B with n_j columns
  Partition the columns of B by componentwise equality
  foreach subset S of the equality partition of the columns of B **do**
    Create a new local macro state $\tilde{y}^{(j)}$ in $\widetilde{RS}^{(j)}$
    Assign all local micro states $z \in S$ to $\tilde{y}^{(j)}$
    Drop all columns of B corresponding to S but a single representant of $\tilde{y}^{(j)}$
foreach $\tilde{\mathbf{x}} \in RS^{(0)} \times RS^{(1)} \times \cdots \times RS^{(J-1)}$ **do**
  if $b[\tilde{\mathbf{x}}] = 1$ **then** Add $\tilde{\mathbf{x}}$ to $\widetilde{RS}$ as a global macro state
return $\widetilde{RS}, \{\widetilde{RS}^{(j)}\}_{j=0}^{J-1}$

2. We reshape $\mathbf{b}$ into a matrix B so that each column corresponds to a local state of the component $LN^{(0)}$,

$$B = \begin{array}{c} \\ 0\,0 \\ 0\,1 \\ 1\,0 \\ 1\,1 \\ 2\,0 \\ 2\,1 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

in order to conclude that

$$\widetilde{RS}_0^{(0)} = \{0_0^{(0)} = M_0^{(0)}, 1_0^{(0)} = M_1^{(0)}\}, \qquad \widetilde{RS}_1^{(0)} = \{0_1^{(0)} = M_2^{(0)}\}.$$

3. After removing all local states of $LN^{(0)}$ except representants of $\widetilde{RS}^{(0)}$, $\mathbf{b}$ is

reshaped again

$$B = \begin{array}{c} \\ \tilde{0}\,0 \\ \tilde{0}\,1 \\ \tilde{1}\,0 \\ \tilde{1}\,1 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

to find that

$$\widetilde{RS}_0^{(1)} = \{0_0^{(1)} = M_0^{(1)}, 1_0^{(1)} = M_1^{(1)}\}, \qquad \widetilde{RS}_1^{(0)} = \{0_1^{(1)} = M_2^{(1)}\}.$$

4. Finally, we reshape

$$B = \begin{array}{c} \\ \tilde{0}\,\tilde{0} \\ \tilde{0}\,\tilde{1} \\ \tilde{1}\,\tilde{0} \\ \tilde{1}\,\tilde{1} \end{array} \begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \end{array}$$

and conclude

$$\widetilde{RS}_0^{(2)} = \{0_0^{(2)} = M_0^{(2)}\}, \qquad \widetilde{RS}_1^{(2)} = \{0_1^{(2)} = M_1^{(2)}\}.$$

5. Unfolding the matrix B

$$\mathbf{b} = \begin{array}{c} \begin{array}{cccccccc} \tilde{0} & \tilde{0} & \tilde{0} & \tilde{0} & \tilde{1} & \tilde{1} & \tilde{1} & \tilde{1} \end{array} \\ \begin{array}{cccccccc} \tilde{0} & \tilde{0} & \tilde{1} & \tilde{1} & \tilde{0} & \tilde{0} & \tilde{1} & \tilde{1} \end{array} \\ \begin{array}{cccccccc} \tilde{0} & \tilde{1} & \tilde{0} & \tilde{1} & \tilde{0} & \tilde{1} & \tilde{0} & \tilde{1} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

shows that the reachable global macro states are

$$\widetilde{RS} = \{\tilde{0} = (\tilde{0}^{(0)}, \tilde{0}^{(1)}, \tilde{0}^{(2)}), \tilde{1} = (\tilde{0}^{(0)}, \tilde{1}^{(1)}, \tilde{1}^{(2)}), \tilde{2} = (\tilde{1}^{(0)}, \tilde{0}^{(1)}, \tilde{1}^{(2)})\},$$

where $\tilde{0}$ corresponds to the free state of the resource, while in $\tilde{1}$ and $\tilde{2}$, the clients $LN^{(1)}$ and $LN^{(0)}$ are using the resource, respectively.

Block kronecker matrix composition

The *hierarchical* or *block* Kronecker form of Q expresses the infinitesimal generator of the CTMC over the reachable state space by the means of macro state decomposition.

The matrices $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}]$ and $Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] \in \mathbb{R}^{n_{j,x} \times n_{n,y}}$ describe the effects of a

single transition $t \in T$ and the aggregated effects of local transitions on $LN^{(j)}$ as its state changes from the local macro state $\tilde{x}^{(j)}$ to $\tilde{y}^{(j)}$, respectively. Formally,

$$q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}][a_x^{(j)}, b_y^{(j)}] = \begin{cases} 1 & \text{if } a_x^{(j)}\,[t\rangle\, b_y^{(j)}, \\ 0 & \text{otherwise,} \end{cases} \tag{4.4}$$

$$Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] = \sum_{t \in T_L^{(j)}} \Lambda(t)\, Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}]. \tag{4.5}$$

In the case $j \notin \operatorname{supp} t$, we define $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}]$ as an identity matrix if $\tilde{x}^{(j)} = \tilde{y}^{(j)}$ and a zero matrix otherwise.

Let us call macro state pairs $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$ *single local macro state transitions* (slmst.) at h if $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ differ only in a single index h ($\tilde{x}^{(h)} \neq \tilde{y}^{(j)}$).

The off-diagonal part Q_O of Q is written as a block matrix with $\tilde{n} \times \tilde{n}$ blocks. A single block is expressed as

$$Q_O[\tilde{\mathbf{x}}, \tilde{\mathbf{y}}] = \begin{cases} \displaystyle\bigoplus_{j=0}^{J-1} Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] \\ \qquad + \displaystyle\sum_{t \in T_S} \Lambda(t) \bigotimes_{j=0}^{J-1} Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] & \text{if } \tilde{\mathbf{x}} = \tilde{\mathbf{y}}, \\ I_{N_1 \times N_1} \otimes Q_L^{(h)}[\tilde{x}^{(h)}, \tilde{x}^{(h)}] \otimes I_{N_1 2 \times N_2} \\ \qquad + \displaystyle\sum_{t \in T_S} \Lambda(t) \bigotimes_{j=0}^{J-1} Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] & \text{if } (\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) \text{ slmst. at } h, \\ \displaystyle\sum_{t \in T_S} \Lambda(t) \bigotimes_{j=0}^{J-1} Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] & \text{otherwise,} \end{cases} \tag{4.6}$$

where $I_1 = \prod_{f=0}^{h-1} n_{h,x^{(h)}}$, $I_2 = \prod_{f=h+1}^{J-1} n_{h,x^{(h)}}$. If $\mathbf{x} = \mathbf{y}$, the matrix block describes transitions which leave the global macro state unchanged, therefore any local transition may fire. If $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$ is slmst. at h, only local transitions on the component h may cause the global state transition, since no other local transition may affect $LN^{(h)}$. In every other case, only synchronizing transitions may occur.

This expansion of block matrices is equivalent to eq. (4.1) on page 34 except the considerations to the hierarchical structure of the state space.

The full Q matrix is written as

$$Q = Q_O + Q_D, \quad Q_D = -\operatorname{diag}\{Q_O \mathbf{1}^{\mathrm{T}}\}$$

as usual.

Algorithm 4.3 shows the construction of the local transition matrices according to eqs. (4.4) and (4.5).

Algorithm 4.3 Transition matrix construction for block Kronecker matrices

Input: State spaces $\widetilde{RS}^{(j)} RS_x^{(j)}$, transitions T, transition rates Λ
Output: Transition matrices $Q_t^{(j)}, Q_L^{(j)}$

1 **for** $j \leftarrow 0$ **to** $J-1$ **do**
2 **foreach** $(\tilde{x}^{(j)}, \tilde{y}^{(j)}) \in \widetilde{RS}^{(j)} \times \widetilde{RS}^{(j)}$ **do**
3 **if** $j \in \operatorname{supp} t$ **then**
4 **allocate** $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \in \mathbb{R}^{n_{j,x} \times n_{n,y}}$
5 Fill in $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}]$ according to eq. (4.4) on page 39
6 **else if** $\tilde{x}^{(j)} = \tilde{y}^{(j)}$ **then** $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \leftarrow I_{n_{j,x} \times n_{j,y}}$
7 **else** $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \leftarrow 0_{n_{j,x} \times n_{j,y}}$
8 **allocate** $Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \in \mathbb{R}^{n_{j,x} \times n_{n,y}}$
9 **foreach** $t \in T_L^{(j)}$ **do**
10 $Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \leftarrow Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] + \Lambda(t) Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}]$

The construction of the block matrix Q is shown in Algorithm 4.4 on page 48. We optimized the formulation from eq. (4.2) in several ways:

- If a Kronecker product contains a 0 matrix term, it is itself zero, therefore, such products are discarded in line 23.
- For identity matrices $I_{N \times N} \otimes I_{n \times n}$ holds. This is exploited in line 21 to reduce the number of terms in the Kronecker producs.
- Instead of constructing Q_O and Q_D separately, the diagonal elements are added to the blocks of Q along its diagonal in line 26.

4.2 Symbolic methods

4.2.1 Multivalued decision diagrams

Multivalued decision diagrams (MDDs) [19] provide a compact, graph-based representation for functions of the form $\mathbb{N}^J \to \{0, 1\}$.

Definition 4.1 A quasi-reduced ordered *multivalued decision diagram* (MDD) encoding the function $f(x^{(0)}, x^{(1)}, \ldots, x^{(J-1)}) \in \{0, 1\}$ (where the domain of each variable $x^{(j)}$ is $D^{(j)} = \{0, 1, \ldots, n_j - 1\}$) is a tuple $MDD = (V, \underline{r}, \underline{0}, \underline{1}, level, children)$, where

- $V = \bigcup_{i=0}^{J} V_i$ is a finite set of *nodes*, where $V_0 = \{\underline{0}, \underline{1}\}$ are the *terminal nodes*, the rest of the nodes $V_N = V \setminus V_0$ are *nonterminal nodes*;
- $level : V \to \{0, 1, \ldots, J\}$ assigns nonnegative *level numbers* to each node ($V_i = \{\underline{v} \in V : level(\underline{v}) = i\}$);
- $\underline{v} \in V_J$ is the *root node*;
- $\underline{0}, \underline{1} \in V_0$ are the *zero* and *one terminal nodes*;
- $children : \left(\bigcup_{i=1}^{J} V_i \times D^{(i-1)}\right) \to V$ is a function defining edges between nodes labeled by the items of the domains, such that either $children(\underline{v}, x) = \underline{0}$ or $level(children(\underline{v}, x)) = level(\underline{v}) - 1$ for all $\underline{v} \in V$, $x \in D^{(level(\underline{v})-1)}$,
- if $\underline{n}, \underline{m} \in V_j, j > 0$ then the subgraphs formed by the nodes reachable from $\underline{n}$ and $\underline{m}$ are either non-isomorphic, or $\underline{n} = \underline{m}$.

We remark that due to the presence of the terminal level V_0 the indexing of the levels and the domains is shifted, i.e. the level V_i corresponds to the domain $D^{(i-1)}$.

According to the semantics of MDDs, $f(\mathbf{x}) = 1$ if the node $\underline{1}$ is reachable from $\underline{r}$ through the edges labeled with $x^{(0)}, x^{(1)}, \ldots, x^{(J-1)}$,

$$f(x^{(0)}, x^{(1)}, \ldots, x^{(J-1)}) = 1 \iff children(children(\ldots children(\underline{r}, x^{(J-1)}) \ldots, x^{(1)}), x^{(0)}) = \underline{1}.$$

Definition 4.2 A quasi-reduced ordered *edge-valued multivalued decision diagram* (EDD) [58] encoding the function $g(x^{(0)}, x^{(1)}, \ldots, x^{(J-1)}) \in \mathbb{N}$ is a tuple $EDD = (V, \underline{r}, \underline{0}, \underline{1}, level, children, label)$, where

- $MDD = (V, \underline{r}, \underline{0}, \underline{1}, level, children)$ is a quasi-reduced ordered MDD,
- $label : \left(\bigcup_{i=1}^{J} V_i \times D^{(i-1)}\right) \to \mathbb{N}$ is an edge label function.

According to the semantics of EDDs, the function g is evaluated as

$$g(\mathbf{x}) = \begin{cases} \text{undefined} & \text{if } f(\mathbf{x}) = 0, \\ \sum_{j=0}^{J-1} label(\underline{n}^{(j)}, x^{(j)}) & \text{if } f(\mathbf{x}) = 1, \end{cases}$$

where f is the function associated with the underlying MDD and $\underline{n}^{(j)}$ are the nodes along the path to $\underline{1}$, i.e.

$$\underline{n}^{(J-1)} = \underline{r}, \quad \underline{n}^{(j)} = children(\underline{n}^{(j+1)}, x^{(j+1)}).$$

4.2.2 Symbolic state spaces

Symbolic techniques involving MDDs can efficiently store large reachable state spaces of superposed Petri nets. Reachable states $x \in RS$ are associated with state codings

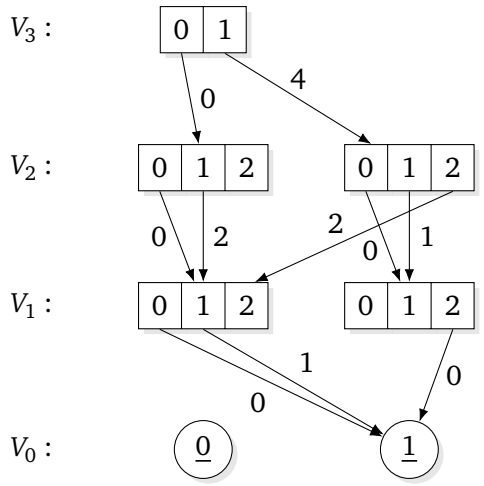


Figure 4.1 EDD state space mapping for the *SharedResource* SSPN.

$\mathbf{x} = (x^{(0)}, x^{(1)}, \ldots, x^{(J-1)})$. The function $f : PS \to \{0, 1\}$ can be stored as an MDD where $f(\mathbf{x}) = 1$ if and only if $x \in RS$. The domains of the MDD are the local state spaces $D^{(j)} = RS^{(j)}$.

Similarly, EDDs can efficiently store the mapping between symbolic state encodings $\mathbf{x}$ and reachable state indices $x \in RS = \{0, 1, \ldots, n-1\}$ as the function $g(\mathbf{x}) = x$. This mapping is used to refer to elements of state probability vectors $\boldsymbol{\pi}$ and the sparse generator matrix Q when these objects are created and accessed.

Running example 4.2 Figure 4.1 shows the state space of the *SharedResource* model encoded as an EDD. The edge labels express the lexicographic mapping of symbolic state codes $\mathbf{x}$ to state indices $\mathbf{x}$. Edges to the terminal zero node $\underline{0}$ were omitted for the sake of clarity.

Some iteration strategies for MDD state space exploration are *breath-first search* and *saturation* [19]. We use the implementation of saturation from the PetriDotNet framework [22, 26].

Algorithms 4.5 and 4.6 on page 49 illustrate the construction of a generator matrix based on the state space encoded as EDDs. The procedure FillIn descends the EDD following a path for the target and the source state simulatenously. The edge labels, represeting state indices, are summed on both paths. If a transition $\mathbf{x}\,[t\rangle\,\mathbf{y}$ is found, the matrix element $q_O[x, y]$ corresponding to the summed indices in increated by the transition rate $\Lambda(t)$. Algorithm 4.6 repeats FillIn for all transitions.

4.2.3 Symbolic hierarchical state space decomposition

The memory requirements and runtime of Algorithm 4.2 on page 37 may be significantly improved by the use of symbolic state space storage instead of a bit vector.

To symbolically partition the local states $RS^{(j)}$ into macro states $\widetilde{RS}^{(j)}$, we will use the following notations of above and below substates from Ciardo et al. [18]:

Definition 4.3 The set of *above* substates coded by the node $\underline{n}$ is

$$\mathcal{A}(\underline{n}) \subseteq \{(x^{(j+1)}, x^{(j+2)}, \ldots, x^{(J-1)}) \in RS^{(j+1)} \times RS^{(j+2)} \times \cdots \times RS^{(J-1)}\},$$

such that

$$\mathbf{x} \in \mathcal{A}(\underline{n}) \iff children(children(\ldots children(\underline{r}, x^{(J-1)}) \ldots, x^{(j+2)}), x^{(j+1)}) = \underline{n}$$

and $j = level(\underline{n}) - 1$, i.e. $\mathcal{A}(\underline{n})$ is the set of all paths in the MDD leading from $\underline{r}$ to $\underline{n}$.

Definition 4.4 The set of *below* substates coded by the node $\underline{n}$ is

$$\mathcal{B}(\underline{n}) \subseteq \{(x^{(0)}, x^{(1)}, \ldots, x^{(j)}) \in RS^{(0)} \times RS^{(1)} \times \cdots \times RS^{(j)}\},$$

such that

$$\mathbf{x} \in \mathcal{B}(\underline{n}) \iff children(children(\ldots children(\underline{n}, x^{(j)}) \ldots, x^{(1)}), x^{(0)}) = \underline{1}$$

and $j = level(\underline{n}) - 1$, i.e. $\mathcal{B}(\underline{n})$ is the set of all paths in the MDD leading from $\underline{n}$ to $\underline{1}$.

The relation $\sim^{(j)}$ over $RS^{(j)}$ can be expressed with $\mathcal{A}(\underline{n})$ and $\mathcal{B}(\underline{n})$ is a way that can be handled easily with symbolic techniques.

Observation 4.5 The set of states which contain some local state $x^{(j)}$ is

$$\{\hat{\mathbf{z}}^{(j)} : \mathbf{z} \in RS, z^{(j)} = x^{(j)}\} = \{(\mathbf{b}, \mathbf{a}) : \underline{n} \in V_{j+1}, children(\underline{n}, x^{(j)}) \neq \underline{n}, \mathbf{b} \in \mathcal{B}(children(\underline{n}, x^{(j)})), \mathbf{a} \in \mathcal{A}(\underline{n})\}.$$

Proof. Any reachable state $\mathbf{z} \in RS$ that has $z^{(j)} = x^{(j)}$ is represented by a path in the MDD that passes through a pair of nodes $\underline{n} \in V_{j+1}$ and $children(\underline{n}, x^{(j)}) \neq \underline{0}$.

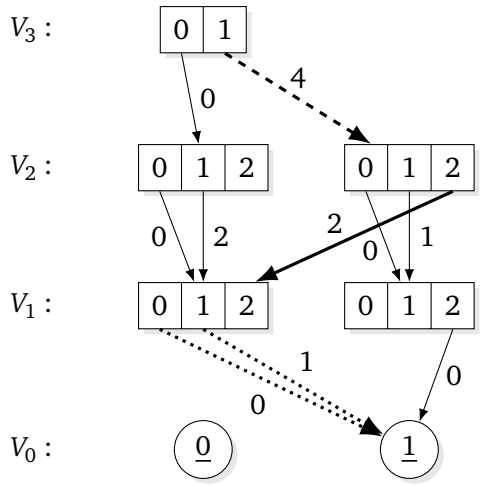


Figure 4.2 The set of all paths having $x^{(1)} = 2^{(1)}$ in the *SharedResource* EDD.

Therefore, some path $\mathbf{a} \in \mathcal{A}(\underline{n})$ must be followed from $\underline{r}$ to reach $\underline{n}$, then some path $\mathbf{b} \in \mathcal{B}(children(\underline{n}, x^{(j)}))$ must be followed from $children(\underline{n}, x^{(j)})$ to $\underline{1}$.

This means all paths from $\underline{r}$ to $\underline{1}$ containing $x^{(j)}$ are of the form $(\mathbf{b}, x^{(j)}, \mathbf{a})$ and the converse also holds. □

Running example 4.3 Figure 4.2 shows all path in the *SharedResource* MDD with $x^{(1)} = 2^{(1)}$.

The single path in the set $\mathcal{A}(\underline{n})$ is dashed, while paths in the set $\mathcal{B}(children(\underline{n}, 2^{(1)}))$ are drawn as dotted edges.

Observation 4.6 If $\underline{n}$ and $\underline{m}$ are distinct nonterminal nodes of a quasi-reduced ordered MDD, $\mathcal{A}(\underline{n}) \cup \mathcal{A}(\underline{m}) = \emptyset$ and $\mathcal{B}(\underline{n}) \neq \mathcal{B}(\underline{m})$.

Proof. We prove the statements indirectly. Let $\mathbf{a} \in \mathcal{A}(\underline{n}) \cup \mathcal{A}(\underline{m})$. If we follow the path $\mathbf{a}$ for $\underline{r}$, we arrive at $\underline{n}$, because $\mathbf{a} \in \mathcal{A}(\underline{n})$. However, we also arrive at $\underline{m}$, because $\mathbf{a} \in Above(\underline{m})$. This is a contradition, since $\underline{n} \neq \underline{m}$, $\mathcal{A}(\underline{n})$ and $\mathcal{A}(\underline{m}$ must be disjoint.

Now suppose that there are $\underline{n}, \underline{m} \in V_N$ such that $\mathcal{B}(\underline{n}) = \mathcal{B}(\underline{m})$. Because the paths $\mathcal{B}(\underline{n})$ describe the subgraph reachable from $\underline{n}$ completely, this means the subgraphs reachable from $\underline{n}$ and $\underline{m}$ are isomorphic. This is impossible, because then the MDD cannot be reduced, thus $\mathcal{B}(\underline{n})$ and $\mathcal{B}(\underline{m})$ must be distinct. □

Observation 4.7 The relation $x^{(j)} \sim^{(j)} y^{(j)}$ can be expressed as

$$x^{(j)} \sim^{(j)} y^{(j)} \iff \{(\underline{n}, children(\underline{n}, x^{(j)})) : \underline{n} \in V_{j+1}\} = \{(\underline{n}, children(\underline{n}, y^{(j)})) : \underline{n} \in V_{j+1}\}.$$

Proof. Let

$$X = \{\hat{\mathbf{z}}^{(j)} : \mathbf{z} \in RS, z^{(j)} = x^{(j)}\}, \qquad Y = \{\hat{\mathbf{z}}^{(j)} : \mathbf{z} \in RS, z^{(j)} = y^{(j)}\}.$$

According to eq. (4.3) on page 35, $x^{(j)} \sim^{(j)} y^{(j)}$ if and only if $X = Y$.

Define

$$X(\underline{n}) = \{\mathbf{b} : (\mathbf{b}, \mathbf{a}) \in X, \mathbf{b} \in \mathcal{A}(\underline{n})\}, \qquad Y(\underline{n}) = \{\mathbf{b} : (\mathbf{b}, \mathbf{a}) \in Y, \mathbf{b} \in \mathcal{A}(\underline{n})\}.$$

$X = Y$ holds precisely when $X(\underline{n}) = Y(\underline{n})$ for all $\underline{n} \in V_{j+1}$. We may notice that $\{X(\underline{n}) \times \mathcal{A}(\underline{n})\}_{\underline{n} \in V_{j+1}}$ and $\{Y(\underline{n}) \times \mathcal{A}(\underline{n})\}_{\underline{n} \in V_{j+1}}$ are partitions of X and Y, respectively, because the $\mathcal{A}$-sets are disjoint for each node.

According to Observation 4.5,

$$X(\underline{n}) = \mathcal{B}(children(\underline{n}, x^{(j)})), \qquad Y(\underline{n}) = \mathcal{B}(children(\underline{n}, y^{(j)})).$$

Thus, $X(\underline{n}) = Y(\underline{n})$ if and only if $children(\underline{n}, x^{(j)}) = children(\underline{n}, y^{(j)})$, because the $\mathcal{B}$-sets are distinct for each node. □

Observation 4.7 can be interpreted as the statement that $x^{(j)} \sim^{(j)} y^{(j)}$ if and only if the MDD edges corresponding to $x^{(j)}$ are always parallel, i.e. from the node $\underline{n}$ they all go to the same node $\underline{m}(\underline{n})$ for all $\underline{n} \in V_{j+1}$.

The macro states can be constructed from the paralell edges in the MDD by partition refinement. This process is performed by Algorithm 4.7 on page 50.

The key step in partition refinement is in line 15, where the candiate macro state S is split into S_1 and S_2. Edges in S_1 are all parallel and go from $\underline{n}$ to $\underline{m}$, while S_2 is further split. The process is repeated for each node $\underline{n}$ and level V_{j+1} until only parallel macro state candidates remain.

This procedure is based on an idea of Buchholz and Kemper [12], however, we employed partition refinement instead of hashing and and proved correctness of the algorithm formally.

A block Kronecker matrix may be constructed from the decomposed state space by Algorithm 4.4.

$$A = \begin{pmatrix} 1 & 0 & 0 & 2.5 \\ 3 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 \\ 5 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{aligned} A = \{&\{(1,0),(3,1),(4,2),(5,3)\}, \\ &\{(1,1)\}, \\ &\{\}, \\ &\{(2.5,0),(1,2)\}\} \end{aligned}$$

Figure 4.3 Compressed Column Storage of a matrix.

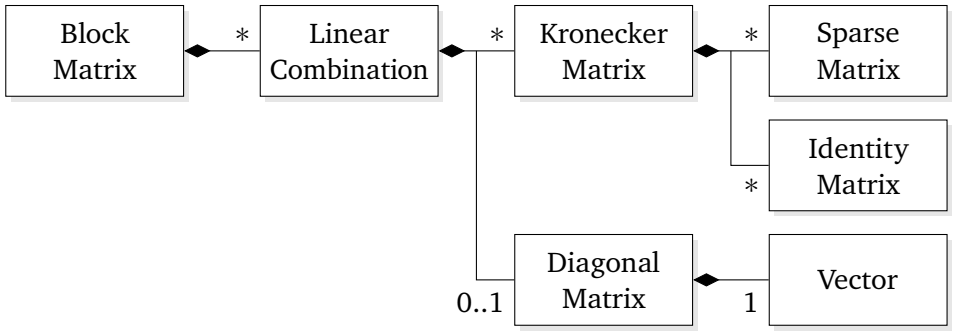


Figure 4.4 Data structure for block Kronecker matrices.

4.3 Matrix storage

Existing linear algebra and matrix libraries, such as [9, 25, 34, 46, 62], usually have unsatisfactory support for operations required in stochastic analysis algorithms with decomposed matrices, for example, multiplications with Kronecker and block Kronecker matrices. Therefore, we have decided to develop a linear algebra framework in C#.NET specifically for stochastic algorithms as a basis of our stochastic analysis framework.

Sparse matrices are stored in Compressed Column Storage (CCS) format, i.e. an array or values and row indices are stored for each column of the matrix, as illustrated in Figure 4.3. This facilitates multiplication from left with row vectors. To reduce pressure on the garbage collector (GC), matrices are vectors are stored in manually allocated and managed memory.

While other sparse matrix formats, such as sliced LAPACK are more amenable to parallel and SIMD processing Kreutzer et al. [40], CCS was selected due to implementation simplicity and the small number of nonzero entries in each column of the matrix, which reduces the potential benefits of SIMD implementations.

Decomposed Kronecker and block Kronecker matrices are stored as algebraic expression trees as shown in Figure 4.4. Matrix multiplication and manipulation algorithms for expression trees are detailed in Section 5.4 on page 64.

The expression tree approach allows the use of arbitrary matrix decompositions that can be expressed with block matrices, linear combinations and Kronecker products. The implementation of additional opeartional primitives is also straightforward. The data structure forms a flexible basis for the development of stochastic analysis algorithms with decomposed matrix representations.

Algorithm 4.4 Block Kronecker matrix construction.

Input: State spaces $\widetilde{RS}, \widetilde{RS}^{(j)} RS_x^{(j)}$, transitions T, transition rates Λ, matrices $Q_t^{(j)}, Q_L^{(j)}$
Output: Infinitesimal generator Q

allocate block matrix Q with $\tilde{n} \times \tilde{n}$ blocks
foreach $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) \in \widetilde{RS} \times \widetilde{RS}$ **do**
  Initialize $Q[\tilde{\mathbf{x}}, \tilde{\mathbf{y}}]$ as a linear combination of matrices
  if $\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$ **then**
    for $j \leftarrow 0$ **to** $J-1$ **do**
      if $Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \neq 0$ **then**
        $I_1 \leftarrow I_{\prod_{f=0}^{j-1} n_{f,x^{(f)}} \times \prod_{h=0}^{j-1} n_{f,x^{(f)}}}, \quad I_2 \leftarrow \mathrm{I}_{\prod_{g=j+1}^{J-1} n_{f,x^{(f)}} \times \prod_{f=j+1}^{J-1} n_{f,x^{(f)}}}$
        $Q[\tilde{\mathbf{x}}, \tilde{\mathbf{x}}] \leftarrow Q[\tilde{\mathbf{x}}, \tilde{\mathbf{x}}] + I_1 \otimes Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] \otimes I_2$
  else if $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$ is a slmst. at h **then**
    $I_1 \leftarrow I_{\prod_{f=0}^{h-1} n_{f,x^{(f)}} \times \prod_{h=0}^{h-1} n_{f,x^{(f)}}}, \quad I_2 \leftarrow \mathrm{I}_{\prod_{f=f+1}^{J-1} n_{f,x^{(f)}} \times \prod_{f=h+1}^{J-1} n_{f,x^{(f)}}}$
    $Q[\tilde{\mathbf{x}}, \tilde{\mathbf{x}}] \leftarrow Q[\tilde{\mathbf{x}}, \tilde{\mathbf{x}}] + I_1 \otimes Q_L^{(h)}[\tilde{x}^{(h)}, \tilde{x}^{(h)}] \otimes I_2$
  foreach $t \in T_S$ **do**
    Initialize B as an empty Kronecker product
    zeroProduct $\leftarrow$ false
    for $j \leftarrow 0$ **to** $J-1$ **do**
      if $Q^{(j)}[\mathbf{x}, \mathbf{y}] = 0$ **then**
        zeroProduct $\leftarrow$ true
        break
      else if $Q^{(j)}[\mathbf{x}, \mathbf{y}]$ is an identity matrix **then**
        if the last term of B is an indentity matrix $I_{N,N}$ **then**
          Enlarge the last term of B to $I_{N n_{j,x} \times N n_{j,y}}$
        else $B \leftarrow B \otimes Q^{(j)}[\mathbf{x}, \mathbf{y}]$
    if $\neg$*zeroProduct* **then** $Q[\mathbf{x}, \mathbf{y}] \leftarrow Q[\mathbf{x}, \mathbf{y}] + \Lambda(t) B$
allocate block vector $\mathbf{d}$ with $\tilde{n}$ blocks
$\mathbf{d} \leftarrow -Q\mathbf{1}^{\mathrm{T}}$
foreach $\tilde{x} \in \widetilde{RS}$ **do** $Q[\tilde{x}, \tilde{x}] \leftarrow Q[\tilde{x}, \tilde{x}] + \mathrm{diag}\{\mathbf{d}[\tilde{x}]\}$
return Q

Algorithm 4.5 FillIn procedure for matrix construction from EDD state space.

Input: node for target state $\underline{t}$, node for source state $\underline{s}$, target state offset y, source state offset y, transition t, transition rate λ, matrix Q_O

```
1  if level(t) = 0 then
2      if t = 1 ∧ s = 1 then q_O[x, y] ← q_O[x, y] + λ
3  else
4      j ← level(t) − 1
5      foreach y^(j) ∈ RS^(j) do
6          if children(t, y^(j)) = 0 then return
7          Find x^(j) such that x^(j) [t⟩ y^(j)
8          if children(s, x^(j)) = 0 then return
9          FillIn(children(t, y^(j)), children(s, x^(j)),
10                 y + label(t, y^(j)), x + label(t, x^(j)), t, λ, Q_O)
```

Algorithm 4.6 Sparse matric construction from EDD state space.

Input: state space MDD root $\underline{r}$, state space size n, transitions T, transition rate function Λ

Output: generator matrix $Q \in \mathbb{R}^{n \times n}$

```
1  allocate Q_O ∈ ℝ^(n×n)
2  foreach t ∈ T do
3      FillIn(r, r, 0, 0, t, Λ(t), Q_O)
4  d ← −Q_O 1^T
5  return Q_O + diag{d}
```

Algorithm 4.7 Local macro state construction by partition refinement.

Input: Symbolic state space *MDD*

Output: Local macro states $\widetilde{RS}^{(j)}$, $RS_x^{(j)}$

for $j \leftarrow 0$ **to** $J-1$ **do**
  Initialize the empty queue Q
  $Done \leftarrow \{RS^{(j)}\}$
  foreach $\underline{n} \in V_{j+1}$ **do**
    foreach $S \in Done$ **do**
      Enqueue(Q, S)
    $Done \leftarrow \emptyset$
    while $\neg$Empty(Q) **do**
      $S \leftarrow$ Dequeue(Q)
      $S_1 \leftarrow \emptyset$
      $S_2 \leftarrow \emptyset$
      Let x_0 be any element of S
      $\underline{m} \leftarrow children(\underline{n}, x_0)$
      foreach $x \in S \setminus \{x_0\}$ **do**
        if $\underline{m} = children(\underline{n}, x)$ **then** $S_1 \leftarrow S_1 \cup \{x\}$ **else** $S_2 \leftarrow S_2 \cup \{x\}$
      if $S_2 \neq \emptyset$ **then**
        Enqueue(Q, S_2)
      $Done \leftarrow Done \cup \{S_1\}$
  $\tilde{n}_j \leftarrow |Done|$
  $\widetilde{RS}^{(j)} \leftarrow \{\tilde{0}^{(j)}, \tilde{1}^{(j)}, \ldots, \widetilde{\tilde{n}_j - 1}^{(j)}\}$
  Each set $S \in Done$ is a local macro state $\widetilde{RS}_x^{(j)}$

Chapter 5

Algorithms for stochastic analysis

Steady state, transient, accumulated and sensitivity analysis problems pose several numerical challanges, especially when the state space of the CTMC and the vectors and matrices involved in the computation are extemely large.

In steady-state and sensitivty analysis, linear equations of the form $\mathbf{x}A = \mathbf{b}$ are solved, such as eqs. (2.3) and (2.8) on page 9 and on page 12. The steady-state probability vector is the solution of the linear system

$$\frac{\mathrm{d}\boldsymbol{\pi}}{\mathrm{d}t} = \boldsymbol{\pi}Q = \mathbf{0}, \quad \boldsymbol{\pi}\mathbf{1}^{\mathrm{T}} = 1, \qquad \text{(2.3 revisited)}$$

where the infinitesimal generator Q is a rank-deficient matrix. Therefore, steady-state solution methods must handle various generator matrix decompositions and homogenous linear equation with rank deficient matrices. Convergence and compuation times of linear equations solvers depend on the numerical properties of the Q matrices, thus different solvers may be preferred for different models.

In transient analysis, initial value problems with first-order linear differetial equations such as eqs. (2.2) and (2.6) on page 8 and on page 11 are considered. The decomposed generator matrix Q must be also handled efficiently. Another difficulty is caused by the *stiffness* of differential equations arising from some models, which may significantly increase computation times.

To facilitate configurable stochastic analysis, we developeds several linear equation solvers and transient analysis methods. Where it is reasonable, the implementation is independent of the form of the generator matrix Q. We achieved genericity by defining an interface between the algorithms and the data structures with operations including

- multiplication of a matrix with a vector from left or right,
- scalar product of vectors with other vectors and columns of matrices,
- specialized operations like accessing the diagonal or off-diagonal parts of a matrix and replacing columns of matrices.

The implementation of these low-level operations is also decoupled from the data structure. This strategy enables further configurability by replacing the operations at runtime, for example, switching between sequential and parallel execution for different parts of the analysis workflow.

While high level configurability allows the modeler to select analysis algorithms appropriate for the model and performance measures under study, low leven configurability of the operations enables additional customization of algorithm execution for the structure of the model as well as the hardware in use. Benchmark results for the workflow are discussed in Section 6.4 on page 74.

In this chapter, we describe the algorithms implemented in our stochastic analysis framework. The pseudocode of the algorithms is annotated with the low level operations performed on the configurable data structure by the high level algorithms.

5.1 Linear equation solvers

5.1.1 Explicit solution by LU decomposition

LU decomposition is a direct method for solving linear equations with forward and backward substitution, i.e. it does not require iteration to reach a given precision.

The decomposition computes the lower triangular matrix L and upper triangular matrix U such that

$$A = LU.$$

To solve the equation

$$\mathbf{x}A = \mathbf{x}LU = \mathbf{b}$$

forward substitution is applied first to find $\mathbf{z}$ in

$$\mathbf{z}U = \mathbf{b},$$

then $\mathbf{x}$ is computed by back substitution from

$$\mathbf{x}L = \mathbf{b}.$$

We used Crout's LU decomposition [52, Section 2.3.1], presented in Algorithm 5.1), which ensures

$$u[i,i] = 1 \text{ for all } i = 0, 1, \ldots, n-1,$$

i.e. the diagonal of the U matrix is uniformly 1. The matrix is filled in during the decomposition even if it was initially sparse, therefore it should first be copied to a dense array storage for efficiency reasons. This considerably limits the size of Markov chains that can be analysed by direct solution due to memory requirements. Our data structure allows access to upper and lower diagonal parts to matrices and linear combinations, therefore no additional storage is needed other than A itself.

Algorithm 5.1 Crout's LU decomposition without pivoting.

Input: the matrix $A \in \mathbb{R}^{n \times n}$ operated on in-place
Output: $L, U \in \mathbb{R}^{n \times n}$ such that $A = LU$, $u[i,i] = 1$ for all $i = 0, 1, \ldots, n-1$

```
1  for i ← 0 to n − 1 do
2      for j ← 0 to i do a[i, j] ← a[i, j] − Σ_{k=0}^{j−1} a[i, k]a[k, j]
3      for j ← i + 1 to n − 1 do a[i, j] ← (a[i, j] − Σ_{k=0}^{i−1} a[i, k]a[i, j]) / a[i, i]
4  Let A_L, A_D and A_U refer to the strictly lower triangular, diagonal and strictly
   upper triangular parts of A, respectively.
5  L ← A_L + A_D
6  U ← A_U + I
7  return L, U
```

Algorithm 5.2 Forward and back substitution.

Input: $U, L \in \mathbb{R}^{n \times n}$, right vector $\mathbf{b} \in \mathbb{R}^n$
Output: solution of $\mathbf{x}LU = \mathbf{b}$

```
1  allocate x, z ∈ R^n
2  if b = 0 then z ← 0        // Skip forward substitution for homogenous equations
3  else for j ← 0 to n − 1 do z[j] ← b[j] · Σ_{i=0}^{j−1} u[i, j]
4  if l[n − 1, n − 1] ≈ 0 then
5      if z[n − 1] ≈ 0 then x[n − 1] ← 0        // Set the free parameter to 1
6      else error "inconsistent linear equation system"
7  else x[n − 1] ← z[n − 1]/l[n − 1, n − 1]
8  for j ← n − 2 downto 0 do
9      if l[j, j] ≈ 0 then error "more than one free parameter"
10     x[j] ← (z[i] − Σ_{i=j+1}^{n−1} x[i]l[i, j]) / l[j, j]
11 return x
```

The forward and back substitution process is shown in Algorithm 5.2. If multiple equations are solver with the same matrix, its LU decomposition may be cached.

Matrices of less than full rank

If the matrix Q is of rank $n-1$, the element $l[n-1, n-1]$ in Crout's LU decomposition will be 0. In this case, $x[n-1]$ is a free parameter and will be set to 1 to yield a nonzero solution vector when $z[n-1] = 0$. If $z[n-1] \neq 0$, the equation $\mathbf{x}L = \mathbf{z}$ does not have a solution and the error condition in line 6 is triggered. A matrix of rank less than $n-1$ triggers the error condition in line 9.

In practice, the algorithm can be used to solve homogenous equations in Markovian

Algorithm 5.3 Basic iterative scheme for solving linear equations.

Input: matrix $A \in \mathbb{R}^{n\times n}$, right vector $\mathbf{b} \in \mathbb{R}^n$, initial guess $\mathbf{x} \in \mathbb{R}^n$, tolerance $\tau > 0$
Output: approximate solution of $\mathbf{x}A = \mathbf{b}$ and its residual norm

```
allocate x' ∈ R^n                          // Previous iterate for convergence test
repeat
    x' ← x                                 // Save the previous vector
    x ← f(x')
until ||x' − x|| ≤ τ
return x and ||xQ − b||
```

analysis, because the infinitesimal generator matrix Q of an irreducible CTMC is always of rank $n-1$. The solution vector $\mathbf{x}$ is not a probability vector in general, so it must be normalized as $\boldsymbol{\pi} = \mathbf{x}/\mathbf{x}\mathbf{1}^{\mathrm{T}}$ to get a stationary probability distribution vector.

5.1.2 Iterative methods

Iterative methods express the solution of the linear equation $\mathbf{x}A = \mathbf{b}$ as a recurrence

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}),$$

where $\mathbf{x}_0$ is an initial guess vector. The iteration converges to a solution vector when $\lim_{k\to\infty} \mathbf{x}_k = \mathbf{x}$ exists and $\mathbf{x}$ equals the true solution vector $\mathbf{x}^*$. The iteration is illustrated in Algorithm 5.3.

The process is assumed to have converged if subsequent iterates are sufficiently close, i.e. the stopping criterion at the kth iteration is

$$\|\mathbf{x}_k - \mathbf{x}_{k-1}\| \leq \tau \tag{5.1}$$

for some prescribed tolerance τ. In our implementation, we selected the L^1-norm

$$\|\mathbf{x}_k - \mathbf{x}_{k-1}\| = \sum_i \left| x_k[i] - x_{k-1}[i] \right|$$

as the vector norm used for detecting convergence.

Premature termination may be avoided if iterates spaced $m > 1$ iterations apart are used for convergence test ($\|\mathbf{x}_k - \mathbf{x}_{k-m}\| \leq \tau$), but only at the expense of additional memory required for storing m previous iterates. In order to handle large Markov chains with reasonable memory consumption, we only used the convergence test with a single previous iterate.

Correctness of the solution can be checked by observing the norm of the residual $\mathbf{x}_k A - \mathbf{b}$, since the error vector $\mathbf{x}_k - \mathbf{x}^*$ is generally not available. Because the additional matrix multiplication may make the latter check costly, it is performed only after

Algorithm 5.4 Power iteration.

```
1 α^{-1} ← 1/max_i |a[i,i]|
2 repeat
3 |   x' ← xA                                  // Vector-matrix product
4 |   x' ← x' + (−1) · x                      // In-place scaled vector addition
5 |   ε ← α^{-1} ||x'||                        // Vector norm calculation
6 |   x ← x + α^{-1} x'                       // In-place scaled vector addition
7 until ε ≤ τ
```

detecting convergence by eq. (5.1) on page 54. Unfortunately, the residual norm may not be representative of the error norm if the problem is ill-conditioned.

For a detailed discussion stopping criterions and iterate normalization in steady-state CTMC analysis, we refer to [65, Section 10.3.5].

Power iteration

Power iteration [65, Section 10.3.1] is the one of the simplest iterative methods for Markovian analysis. Its iteration function has the form

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}) = \mathbf{x}_{k-1} + \frac{1}{\alpha}(\mathbf{x}_{k-1}A - \mathbf{b}).$$

The iteration converges if the diagonal elements $a[i,i]$ of A are strictly negative, the off-diagonal elements $a[i,j]$ are nonnegative and $\alpha \geq \max_i |a[i,i]|$. The matrix A satisfies these properties if it is an inifinitesimal generator matrix of an irreducible CTMC. The fastest convergence is achieved when $\alpha = \min_i |a[i,i]|$.

Power iteration can be realized by replacing lines 2–5 in Algorithm 5.3 on page 54 with the loop in Algorithm 5.4.

This realization uses memory efficiently, because it only requires the allocation of a single vector $\mathbf{x}'$ in addition to the initial guess $\mathbf{x}$.

Observation 5.1 If $\mathbf{b} = 0$ and A is an inifitesimal generator matrix, then

$$\begin{aligned}\mathbf{x}_k\mathbf{1}^{\mathrm{T}} &= \left[\mathbf{x}_{k-1} + \frac{1}{\alpha}(\mathbf{x}_{k-1}A - \mathbf{b})\right]\mathbf{1}^{\mathrm{T}} \\ &= \mathbf{x}_{k-1}\mathbf{1}^{\mathrm{T}} + \frac{1}{\alpha}\mathbf{x}_{k-1}A\mathbf{1}^{T} - \mathbf{b}\mathbf{1}^{\mathrm{T}} \\ &= \mathbf{x}_{k-1}\mathbf{1}^{\mathrm{T}} + \frac{1}{\alpha}\mathbf{x}_{k-1}\mathbf{0}^{\mathrm{T}} - \mathbf{0}\mathbf{1}^{\mathrm{T}} = \mathbf{x}_{k-1}\mathbf{1}^{\mathrm{T}}.\end{aligned}$$

This means the sum of the elements of the result vector $\mathbf{x}$ and the initial guess vector $\mathbf{x}_0$ are equal, because the iteration leaves the sum unchanged.

To solve an equation of the form

$$\mathbf{x}Q = \mathbf{0}, \quad \mathbf{x}\mathbf{1}^{\mathrm{T}} = 1 \tag{5.2}$$

where Q is an infinitesimal generator matrix, the initial guess $\mathbf{x}_0$ is selected such that $\mathbf{x}_0\mathbf{1}^{\mathrm{T}} = 1$. If the CTMC described by Q is irreducible, we may select

$$x_0[i] \equiv \frac{1}{n}, \tag{5.3}$$

where n is the dimensionality of $\mathbf{x}$. After the initial guess is selected, the equation $\mathbf{x}\mathbf{1}^{\mathrm{T}}$ may be ignored to solve $\mathbf{x}Q = \mathbf{0}$ with the power method. This process yields the solution of the original problem (5.2).

Jacobi and Gauss–Seidel iteration

Jordan and Gauss–Seidel iterative methods [65, Section 10.3.2–3] repeatedly solve a system of simultaneous equations of a specific form.

In Jordan iteration, the system

$$\left.\begin{aligned} b[0] &= x_k[0]a[0,0] + x_{k-1}[1]a[1,0] + \cdots + x_{k-1}[n-1]a[n-1,0], \\ b[1] &= x_{k-1}[0]a[0,1] + x_k[1]a[1,1] + \cdots + x_{k-1}[n-1]a[n-1,1], \\ &\vdots \\ b[n-1] &= x_{k-1}[0]a[0,n-1] + x_{k-1}[1]a[1,n-1] + \cdots + x_k[n-1]a[n-1,n-1], \end{aligned}\right\}$$

is solved for $\mathbf{x}_k$ at each iteration, i.e. there is a single unknown in each row and the rest of the variables are taken from the previous iterate. In vector form, the iteration can be expressed as

$$\mathbf{x}_k = A_D^{-1}(\mathbf{b} - A_O\mathbf{x}_{k-1}),$$

where A_D and A_O are the diagonal (all off-diagonal elements are zero) and off-diagonal (all diagonal elements are zero) parts of $A = A_D + A_O$.

In Gauss–Seidel iteration, the linear system

$$\left.\begin{aligned} b[0] &= x_k[0]a[0,0] + x_{k-1}[1]a[1,0] + \cdots + x_{k-1}[n-1]a[n-1,0], \\ b[1] &= x_k[0]a[0,1] + x_k[1]a[1,1] + \cdots + x_{k-1}[n-1]a[n-1,1], \\ &\vdots \\ b[n-1] &= x_k[0]a[0,n-1] + x_k[1]a[1,n-1] + \cdots + x_k[n-1]a[n-1,n-1], \end{aligned}\right\}$$

is considered, i.e. the ith equation contains the first i elements of $\mathbf{x}_k$ as unknowns. The equations are solved for successive elements of $\mathbf{x}_k$ from top to bottom.

Jacobi over-relaxation, a generalized form of Jacobi iteraion, is realized in Algorithm 5.5. The value 1 of the over-relaxation paramter ω corresponds to ordinary Jacobi iteration. Values $\omega > 1$ may accelerate convergence, while $0 < \omega < 1$ may help diverging Jacobi iteration converge.

Algorithm 5.5 Jacobi over-relaxation.

Input: matrix $A \in \mathbb{R}^{n \times n}$, right vector $\mathbf{b} \in \mathbb{R}^n$, initial guess $\mathbf{x} \in \mathbb{R}^n$, tolerance $\tau > 0$, over-relaxation parameter $\omega > 0$
Output: approximate solution of $\mathbf{x}A = \mathbf{b}$

```
1  allocate x' ∈ ℝ^n
2  Let A_O refer to the off-diagonal part of A.
3  repeat
4      x' ← xA_O                                  // Matrix-vector product
5      x' ← x' + (−1) · b                         // In-place scaled vector addition
6      ϵ ← 0
7      for i ← 0 to n − 1 do
8          y ← (1 − ω)x[i] − ωx'[i]/a[i, i]
9          ϵ ← ϵ + |y − x[i]|
10         x[i] ← y
11 until ϵ ≤ τ
12 return x
```

Jacobi over-relaxation has many parallelization opportunities. The matrix multiplication in line 4 and the vector addition in line 5 can be parallelized, as well as the for loop in line 7. Our implementation takes advantage of the configurable linear algebra operations framework to execute lines 4 and 5 with possible paralellization considering the structures of both the vectors $\mathbf{x}, \mathbf{x}'$ and the matrix A. However, the inner loop is left sequential to reduce implementation complexity, as it represents only a small fraction of execution time compared to the matrix-vector product.

Algorithm 5.6 shows an implementation of successive over-relaxation for Gauss–Seidel iteration, where the notation $\mathbf{a}_O[\cdot, i]$ refers to the ith column of A_O.

Gauss–Seidel iteration cannot easily be parallelized, because calculation of successive elements $x[0], x[1], \ldots$ depend on all of the prior elements. However, in contrast with Jacobi iteration, no memory is required in addition to the vectors $\mathbf{x}$, $\mathbf{b}$ and the matrix X, which makes the algorithm suitable for very large vectors and memory-constrained situations. In addition, convergence is often significantly faster.

The sum of elements $\mathbf{x}\mathbf{1}^{\mathrm{T}}$ does not stay constant during Jacobi or Gauss–Seidel iteration. Thus, when solving equations of the form $\mathbf{x}Q = \mathbf{0}, \mathbf{x}\mathbf{1}^{\mathrm{T}} = 1$, normalization cannot be entierly handled by the initial guess. We instead transform the equation into

Algorithm 5.6 Gauss–Seidel successive over-relaxatation.

Input: matrix $A \in \mathbb{R}^{n\times n}$, right vector $\mathbf{b} \in \mathbb{R}^n$, initial guess $\mathbf{x} \in \mathbb{R}^n$, tolerance $\tau > 0$, over-relaxation parameter $\omega > 0$
Output: approximate solution of $\mathbf{x}A = \mathbf{b}$

allocate $\mathbf{x}' \in \mathbb{R}^n$
Let A_O refer to the off-diagonal part of A.
repeat
  $\epsilon \leftarrow 0$
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $scalarProduct \leftarrow \mathbf{x} \cdot \mathbf{a}_O[\cdot, i]$ // Scalar product with column of matrix
    $y \leftarrow \omega(b[i] - scalarProduct)/a[i,i] + (1-\omega) \cdot x[i]$
    $\epsilon \leftarrow \epsilon + |y - x[i]|$
    $x[i] \leftarrow y$
until $\epsilon \le \tau$
return $\mathbf{x}$

the form

$$\mathbf{x}\begin{pmatrix} q[0,0] & q[0,1] & \cdots & q[0,n-2] & 1 \\ q[1,0] & q[1,1] & \cdots & q[1,n-2] & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ q[n-2,0] & q[n-2,1] & \cdots & q[n-2,n-2] & 1 \\ q[n-1,0] & q[n-1,1] & \cdots & q[n-2,n-1] & 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}, \tag{5.4}$$

where we take advantage of the fact that the infinitesimal generator matrix is not of full rank, therefore one of the columns is redundant and can be replaced with the condition $\mathbf{x}\mathbf{1}^{\mathrm{T}} = 1$. While this transformation may affect the convergence behavior of the algorithm, it allows uniform handling of homogenous and non-homogenous linear equations.

Group iterative methods

Group or *block* iterative methods Stewart [65, Section 10.4] assume the block structure for the vectors $\mathbf{x}$, $\mathbf{b}$ and the matrix A

$$\mathbf{x}[i] \in \mathbb{R}^{n_i}, \mathbf{b}[j] \in \mathbb{R}^{n_j}, A[i,j] \in \mathbb{R}^{n_i \times n_j} \text{ for all } i, j \in \{0, 1, \ldots, N-1\},$$

Infinitesimal generator matrices in the block Kronecker decomposition along with appropriately partitioned vectors match this structure (see eq. (4.6) on page 39). Each block of $\mathbf{x}$ corresponds to a group a variables that are simultaneously solved for.

Algorithm 5.7 Group Jacobi over-relaxation.

Input: block matrix A, block right vector $\mathbf{b}$, block initial guess $\mathbf{n}$, tolerance $\tau > 0$, over-relaxation parameter $\omega > 0$
Output: approximate solution of $\mathbf{x}A = \mathbf{b}$ and its residual norm

allocate $\mathbf{x}'$ and $\mathbf{c}$ with the same block structure as $\mathbf{x}$ and $\mathbf{b}$
Let A_{OB} represent the off-diagonal part of the block matrix A with the blocks along the diagonal set to zero.
repeat
  $\mathbf{x}' \leftarrow \mathbf{x}, \mathbf{c} \leftarrow \mathbf{b}$
  $\mathbf{c} \leftarrow \mathbf{c} + (-1) \cdot \mathbf{x}' A_{OB}$ // Scaled accumulation of vector-matrix product
  parallel for $i \leftarrow 0$ **to** $N-1$ **do** // Loop over all blocks
    Solve $\mathbf{x}[i]A[i,i] = \mathbf{c}[i]$ for $\mathbf{x}[i]$
  $\epsilon \leftarrow 0$
  for $k \leftarrow 0$ **to** $n-1$ **do** // Loop over all elements
    $y \leftarrow \omega x[k] + (1-\omega)x'[k]$
    $\epsilon \leftarrow \epsilon + |y - x'[k]|$
    $x[k] \leftarrow y$
until $\epsilon \leq \tau$

Group Jacobi iteration solves the linear system

$$\left.\begin{aligned}
\mathbf{b}[0] &= \mathbf{x}_k[0]A[0,0] + \mathbf{x}_{k-1}[1]A[1,0] + \cdots + \mathbf{x}_{k-1}[n-1]A[n-1,0],\\
\mathbf{b}[1] &= \mathbf{x}_{k-1}[0]A[0,1] + \mathbf{x}_k[1]A[1,1] + \cdots + \mathbf{x}_{k-1}[n-1]A[n-1,1],\\
&\vdots\\
\mathbf{b}[n-1] &= \mathbf{x}_{k-1}[0]A[0,n-1] + \mathbf{x}_{k-1}[1]A[1,n-1] + \cdots + \mathbf{x}_k[n-1]A[n-1,n-1],
\end{aligned}\right\}$$

while group Gauss–Seidel considers

$$\left.\begin{aligned}
\mathbf{b}[0] &= \mathbf{x}_k[0]A[0,0] + \mathbf{x}_{k-1}[1]A[1,0] + \cdots + \mathbf{x}_{k-1}[n-1]A[n-1,0],\\
\mathbf{b}[1] &= \mathbf{x}_k[0]A[0,1] + \mathbf{x}_k[1]A[1,1] + \cdots + \mathbf{x}_{k-1}[n-1]A[n-1,1],\\
&\vdots\\
\mathbf{b}[n-1] &= \mathbf{x}_k[0]A[0,n-1] + \mathbf{x}_k[1]A[1,n-1] + \cdots + \mathbf{x}_k[n-1]A[n-1,n-1].
\end{aligned}\right\}$$

Implementations of group Jacobi over-relaxation and group Gauss–Seidel successive over-relaxation are shown in Algorithms 5.7 and 5.8 on this page and. The inner linear equations of the form $\mathbf{x}[i]A[i,i] = \mathbf{c}$ may be solved by any algorithm, for example, LU decomposition, iterative methods, or even block-iterative methods if A has a two-level block structure. The choice of the inner algorithm may significantly affect performance and care must be taken to avoid diverging inner solutions in an iterative solver is used.

Algorithm 5.8 Group Gauss–Seidel successive over-relaxation.

Input: block matrix A, block right vector $\mathbf{b}$, block initial guess $\mathbf{n}$, tolerance $\tau > 0$, over-relaxation parameter $\omega > 0$
Output: approximate solution of $\mathbf{x}A = \mathbf{b}$ and its residual norm

allocate $\mathbf{x}'$ and $\mathbf{c}$ large enough to store a single block of $\mathbf{x}$ and $\mathbf{b}$.
repeat
  $\epsilon \leftarrow 0$
  for $i \leftarrow 0$ **to** $N-1$ **do** // Loop over all blocks
    $\mathbf{x}' \leftarrow \mathbf{x}[i], \mathbf{c} \leftarrow \mathbf{b}[i]$
    for $j \leftarrow 0$ **to** $N-1$ **do**
      if $i \neq j$ **then** // Scaled accumulation of vector-matrix product
        $\mathbf{c} \leftarrow \mathbf{c} + (-1) \cdot \mathbf{x}[j]A[i,j]$
    Solve $\mathbf{x}[i]A[i,i] = \mathbf{c}$ for $\mathbf{x}[i]$
    for $k \leftarrow 0$ **to** $n_i - 1$ **do**
      $y \leftarrow \omega x[i][k] + (1-\omega)x'[k]$
      $\epsilon \leftarrow \epsilon + |y - x'[k]|$
      $x[i][k] \leftarrow y$
until $\epsilon \leq \tau$

In Jacobi over-relaxation, paralellization of both the matrix multiplication and the inner loop is possible. However, two vectors of the same size as $\mathbf{x}$ are required for temporary storage.

Gauss–Seidel successive over-relaxation cannot be parallelized easily, but it requires only two temporary vectors of size equal to the largest block of $\mathbf{x}$, much less than Jacobi over-relaxation. Moreover, it often requires fewer steps to converge, making it preferable over Jacobi iteration.

Because the inner solver may be selected by the user and thus its convergence behaviour varies widely, we do not perform the transformation for homogenous equations (5.4). Instead, the normalization $\boldsymbol{\pi} = \mathbf{x}/\mathbf{x}\mathbf{1}^{\mathrm{T}}$ is performed only after finding any nonzero solution of $\mathbf{x}Q = \mathbf{0}$.

For a detailed analysis of the convergence behaviour of group iterative methods, we refer to Greenbaum [33, Chapter 14] and

BiConjugate Gradient Stabilized (BiCGSTAB)

BiConjugate Gradient Stabilized (BiCGSTAB) [61, Section 7.4.2; 68] is an iterative algorithm belonging to the class of Krylov subspace methods, which includes other algorithms such as the Generalized Minimum Residual (GMRES) [60], Conjugate Gradient Squared (CGS) [63] and IDR(s) [64].

We selected BiCGSTAB as the Krylov subspace solver in our framework because of its good convergence behaviour and low memory requirements. BiCGSTAB only requires the storage of 7 vectors, which makes it suitable even for large state spaces with large states vectors, unlike e.g. GMRES, which allocates an additional vector every iteration.

Algorithm 5.9 on page 66 shows the pseudocode for BiCGSTAB. Our implementation is based on the MATLAB code[1] by Barrett et al. [3].

Solving preconditioned equations in the form $\mathbf{x}AM^{-1} = \mathbf{b}M^{-1}$ could improve convergence, but was omitted from our current implementation. As the choice is appropriate preconditioner matrices M is not trivial [41], implementation and sudy of preconditioners for Markov chains, especially with block Kronecker decomposition, is in the scope of our future work.

Because six vectors are allocated in addition to $\mathbf{x}$ and $\mathbf{b}$, the amount of available memory may be a significant bottleneck.

Similar to Observation 5.1 on page 55, it can be seen that the sum $\mathbf{x}\mathbf{1}^{\mathrm{T}}$ stays constant throughout BiCGSTAB iteration. Thus, we can find probability vectors satisfying homogenous equations by the initialization in eq. (5.3) on page 56.

5.2 Transient analysis

5.2.1 Uniformization

The *uniformization* or *randomization* method solves the initial value problem

$$\frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t} = \boldsymbol{\pi}(t)Q, \quad \boldsymbol{\pi}(t) = \boldsymbol{\pi}0 \tag{2.2 revisited}$$

by computing

$$\boldsymbol{\pi}(t) = \sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k e^{-\alpha t} \frac{(\alpha t)^k}{k!}, \tag{5.5}$$

where $P = \alpha^{-1}Q + I$, $\alpha \geq \max_i |a[i,i]|$ and $e^{-\alpha t}\frac{(\alpha t)^k}{k!}$ is the value of the Poisson probabilty function with rate αt at k.

Integrating both sides of eq. (5.5) to compute $\mathbf{L}(t)$ yields [57]

$$\begin{aligned}\int_0^t \boldsymbol{\pi}(u)\,\mathrm{d}u = \mathbf{L}(t) &= \sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k \int_0^t e^{-\alpha u}\frac{(\alpha u)^k}{k!}\,\mathrm{d}u \\ &= \sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k \frac{1}{\alpha} \sum_{l=k+1}^{\infty} e^{-\alpha t}\frac{(\alpha t)^l}{l!}\end{aligned}$$

[1] http://www.netlib.org/templates/matlab//bicgstab.m

$$= \frac{1}{\alpha} \sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k \left(1 - \sum_{l=0}^{k} e^{-\alpha t} \frac{(\alpha t)^l}{l!} \right). \tag{5.6}$$

Both eqs. (5.5) and (5.6) on page 61 and on the current page can be realized as

$$\mathbf{x} = \frac{1}{W} \left(\sum_{k=0}^{k_{\text{left}}-1} w_{\text{left}} \boldsymbol{\pi}_0 P^k + \sum_{k=k_{\text{left}}}^{k_{\text{right}}} w[k - k_{\text{left}}] \boldsymbol{\pi}_0 P^k \right), \tag{5.7}$$

where $\mathbf{x}$ is either $\boldsymbol{\pi}(t)$ or $\mathbf{L}(t)$, k_{left} and k_{right} are *trimming constants* selected based on the required precision, $\mathbf{w}$ is a vector of (possibly accumulated) Poisson weights and W is a scaling factor. The weight before the left cutoff w_{left} is 1 if the accumulated probability vector $\mathbf{L}(t)$ is calculated, 0 otherwise.

Eq. (5.7) is implemented by Algorithm 5.10 on page 67. The algorithm performs *steady-state* detection in line 9 to avoid unneccessary work once the iteration vector $\mathbf{p}$ reaches the steady-state distribution $\boldsymbol{\pi}(\infty)$, i.e. $\mathbf{p} \approx \mathbf{p}P$. If the initial distribution $\boldsymbol{\pi}_0$ is not further needed or can be generated efficiently (as it is the case with a single initial state), the result vector $\mathbf{x}$ may share the same storing, resulting in a memory overhead of only two vectors $\mathbf{p}$ and $\mathbf{q}$.

The weights and trimming constants may be calculated by the famous algorithm of Fox and Glynn [29]. However, their algorithm is extremely complicated due to the limitations of single-precision floating-point arithmetic [38]. We implemented Burak's significantly simpler algorithm [14] in double precision instead (Algorithm 5.11 on page 68), which avoids underflow by a scaling factor $W \gg 1$.

5.2.2 TR-BDF2

A weakness of the uniformization algorithm is the poor tolerance of *stiff* Markov chains. The CTMC is called stiff if the $|\lambda_{\min}| \ll |\lambda_{\max}|$, where $\lambda_{\min}$ and $\lambda_{\max}$ are the nonzero eigenvalues of the infinitesimal generator matrix Q of minimum and maximum absolute value [56]. In other words, stiff Markov chains have behaviors on drastically different timescales, for example, clients are served frequently while failures happen infrequently.

Stiffness leads to very large small rates α in line 2 of Algorithm 5.10 on page 67, thus a large right cutoff k_{right} is required for computing the transient solution with sufficient accuracy. Moreover, the slow stabilization results in taking many iterations before steady-state detection in line 9.

Some methods that can handle stiff CTMCs efficiently are stochastic complementation [47], which decouples the slow and fast behaviors of the system, and adaptive uniformization [48], which varies the uniformization rate α. Alternatively, an L-stable differential equation solver may be used to solve eq. (2.2) on page 8, such as TR-BDF2 [2, 56].

TR-BDF2 is an implicit integrator with alternating trapezoid rule (TR) steps

$$\boldsymbol{\pi}_{k+\gamma}(2I+\gamma h_k Q) = 2\boldsymbol{\pi}_k + \gamma h_k \boldsymbol{\pi}_k Q$$

and second order backward difference steps

$$\boldsymbol{\pi}_{k+1}[(2-\gamma)I-(1-\gamma)h_k Q] = \frac{1}{\gamma}\boldsymbol{\pi}_{k+\gamma} - \frac{(1-\gamma)^2}{\gamma}\boldsymbol{\pi}_k,$$

which advance the time together by a step of size h_k. The constant $0 < \gamma < 1$ sets the breakpoint between the two steps. We set it to $\gamma = 2-\sqrt{2} \approx 0.59$ following the recommendation of Bank et al. [2].

As a guess for the initial step size h_0, we chose the uniformization rate of Q. The kth step size $h_k > 0$, including the 0th one, is selected such that the local error estimate

$$LTE_{k+1} = \left\| 2\frac{-3\gamma^4+4\gamma-2}{24-12\gamma} h_k \left[-\frac{1}{\gamma}\boldsymbol{\pi}_k + \frac{1}{\gamma(1-\gamma)}\boldsymbol{\pi}_{k+\gamma} - \frac{1}{1-\gamma}\boldsymbol{\pi}_{k+1} \right] \right\| \tag{5.8}$$

is bounded by the local error tolerance

$$LTE_{k+1} \leq \left(\frac{\tau - \sum_{i=0}^{k} LTE_i}{t - \sum_{i=0}^{k} k_i} \right) h_{k+1}.$$

This Local Error per Unit Step (LEPUS) error control "produces excellent results for many problems", but is usually costly [56]. Moreover, the accumulated error at the end of integration may be larger than the prescribed tolerance τ, since eq. (5.8) is only an approximation of the true error.

An implementation of TR-BDF2 based on the pseudocode of A. L. Reibman and Trivedi [56] is shown in Algorithm 5.12 on page 69.

In lines 12 and 16 any linear equation solver from Section 5.1 on page 52 may be used except power iteration, since the matrices, in general, do not have strictly negative diagonals. Due to the way the matrices, which are linear combinations of I and Q, are passed to the inner solvers, our TR-BDF2 integrator is currently limited to Q matrices which are not in block form.

The vectors $\boldsymbol{\pi}_0, \boldsymbol{\pi}_k$ and $\boldsymbol{\pi}_{k+\gamma}, \mathbf{d}_{k+1}$ may share storage, respectively, therefore only 4 state-space sized vectors are required in addition to the initial distribution $\boldsymbol{\pi}_0$.

The most computationally intensive part is the solution of two linear equation per every attempted step, which may make TR-BDF2 extremely slow. However, its performance does *not* depend on the stiffness of the Markov chain, which may make it better suited to stiff CTMCs than uniformization [56].

5.3 Mean time to first failure

In MTFF calculation (Section 2.2.3 on page 12), quantities of the forms

$$MTFF = -\underbrace{\boldsymbol{\pi}_U Q_{UU}^{-1}}_{\gamma} \mathbf{1}^{\mathrm{T}}, \quad \mathbb{P}(X(TFF_{+0}) = y) = -\underbrace{\boldsymbol{\pi}_U Q_{UU}^{-1}}_{\gamma} \mathbf{q}_{UD'}^{\mathrm{T}} \qquad \text{(2.9, 2.10 revisited)}$$

are computed, where U, D, D' are the set of operations states, failure states and a specific failure mode $D' \subsetneq D$, respectively.

The vector $\boldsymbol{\gamma} \in \mathbb{R}^{|U|}$ is the solution of the linear equation

$$\boldsymbol{\gamma} Q_{UU} = \boldsymbol{\pi}_U \tag{5.9}$$

and may be obtained by any linear equation solver.

The sets $U, D = D_1 \cup D_2 \cup \cdots$ are constructed by the evaluation of CTL expressions. If the failure mode D_i is described by φ_i, then the sets D and U are described by CTL formulas $\varphi_D = \neg\mathbf{AX}\,\mathsf{true} \vee \varphi_1 \vee \varphi_2 \vee \cdots$ and $\varphi_U = \neg\varphi_D$, where the deadlock condition $\neg\mathbf{AX}\,\mathsf{true}$ is added to make (5.9) irreducible.

After the set U is generated symbolically, the matrix Q_{UU} may be decomposed in the same way as the whole state space S. Thus, the vector-matrix operations required for solving (5.9) can be executed as in steady-state analysis.

5.4 Efficient vector-matrix products

Iterative linear equation and transient distribution solvers require several vector-matrix products per iteration. Therefore, efficient vector-matrix multiplication algorithms are required for the various matrix storage methods (i.e. dense, sparse and block Kronecker matrices) to support configurable stochastic analysis.

Our data structure supports run-time reconfiguration of operations, for example, to switch between parallel and sequential matrix multiplication implementations for different parts of an algorithm, depending on the characteristics of the model and the hardware which runs the analysis.

Implemented matrix multiplication for the data structure (see Figure 4.4 on page 46) routines are

- Multiplication of vectors with dense and sparse matrices. Sparse matrix multiplication may be parallelized by splitting the columns of the matrix into chunck and submitting each chunk to the executor thread pool.

  Operations with vectors and sparse matrices are implemented in an `unsafe`[2] context. The elements of the data structures are not under the influence of the

[2] `https://msdn.microsoft.com/en-us/library/chfa2zb8.aspx`

Garbage Collector runtime, but stored in natively allocated memory. This allows the handling of large matrices without adversely impacting the performance of other parts of the program, albeit the cost of allocations in increased.

- Multiplication with block matrices by delegation to the constituent blocks of the matrix (Algorithm 5.13 on page 70). The input and output vectors are converted to block vectors before multiplication. If parallel execution is required, each block of the output vector can be computed in a different task, since it is independent from the others.

- Multiplication by a linear combination of matrices is delegated to the constituent matrices (Algorithm 5.14 on page 70). An in-place scaled addition of vector-matrix product to a vector operation is required for this delegation. To facilitate this, each vector-matrix multiplication algorithm is implemented also as an in-place addition and in-place scaled addition of vector-matrix product, and the appropriate implementation is selected based on the function call aruments.

- Multiplications $\mathbf{b} \cdot \mathrm{diag}\{\mathbf{a}\}$ by diagonal matrices are executed as elementwise product $\mathbf{b} \odot \mathbf{a}$. The special case of multiplication by an identity matrix is equivalent to a vector copy.

- Multiplications by Kronecker products is performed by the SHUFFLE algorithm [5, 11] as shown in Algorithm 5.15 on page 70.

  The algorithm requires access to slices of a vector, denoted as $\mathbf{x}[i_0{:}s{:}l]$, which refers to the elements $x[i], x[i{+}s], x[i{+}2s], \ldots, x[i{+}(l{-}1)s]$. Thus, slices were integrated into the operations framework as first-class elements, and multiplication algorithms are implemented with support for vector slice indexing.

  SHUFFLE rewrites the Kronecker products as

$$\bigotimes_{h=0}^{k-1} A^{(h)} = \prod_{h=0}^{k-1} I_{\prod_{f=0}^{h-1} n_f \times \prod_{f=0}^{h-1} n_f} \otimes A^{(h)} \otimes I_{\prod_{f=h+1}^{k-1} m_f \times \prod_{f=h+1}^{k-1} m_f},$$

  where $I_{a\times a}$ denotes an $a \times a$ identity matrix. Multiplications by terms of the form $I_{N\times N} \otimes A^{(h)} \otimes I_{M\times M}$ are carried out in the loop at line 8 of Algorithm 5.15.

  The temporary vectors $\mathbf{x}, \mathbf{x}'$ are large enough store the results of the successive matrix multiplications. They are cached for every worker thread to avoid repeated allocations.

  Other algorithms for vector-Kronecker product multiplication are the SLICE [28] and SPLIT [21] algorithms, which are more amenable to parallel execution than SHUFFLE. Their implementation is in the scope of our future work.

Algorithm 5.9 BiCGSTAB iteration without preconditioning.

Input: matrix $A \in \mathbb{R}^{n \times n}$, right vector $\mathbf{b} \in \mathbb{R}^n$, initial guess $\mathbf{x} \in \mathbb{R}^n$, tolerance $\tau > 0$
Output: approximate solution of $\mathbf{x}A = \mathbf{b}$

allocate $\mathbf{r}, \mathbf{r}_0, \mathbf{v}, \mathbf{p}, \mathbf{s}, \mathbf{t} \in \mathbb{R}^n$
$\mathbf{r} \leftarrow \mathbf{b}$
$\mathbf{r} \leftarrow \mathbf{r} + (-1) \cdot \mathbf{x}A$ // Scaled accumulation of vector-matrix product
if $\|\mathbf{r}\| \leq \tau$ **then**
    message "initial guess is correct, skipping iteration"
    return $\mathbf{x}$
$\mathbf{r}_0 \leftarrow \mathbf{r}, \mathbf{v} \leftarrow \mathbf{0}, \mathbf{p} \leftarrow \mathbf{0}, \rho' \leftarrow 1, \alpha \leftarrow 1, \omega \leftarrow 1$
while true **do**
    $\rho \leftarrow \mathbf{r_0} \cdot \mathbf{r}$ // Scalar product
    if $\rho \approx 0$ **then error** "breakdown: $\mathbf{r} \perp \mathbf{r}_0$"
    $\beta \leftarrow \rho / \rho' \cdot \alpha / \omega$
    $\mathbf{p} \leftarrow \mathbf{r} + \beta \cdot \mathbf{p}$ // Scaled vector addition
    $\mathbf{p} \leftarrow \mathbf{p} + (-\beta\omega) \cdot \mathbf{v}$ // In-place scaled vector addition
    $\alpha \leftarrow \rho / (\mathbf{r}_0 \cdot \mathbf{v})$ // Scalar product
    $\mathbf{r} \leftarrow \mathbf{s} + (-\alpha) \cdot \mathbf{s}$ // Scaled vector addition
    if $\|\mathbf{s}\| < \tau$ **then**
        $\mathbf{x} \leftarrow \mathbf{x} + \alpha \cdot \mathbf{p}$ // In-place scaled vector addition
        message "early return with vanishing $\mathbf{s}$"
        return $\mathbf{x}$
    $\mathbf{t} \leftarrow \mathbf{s}A$ // Vector-matrix multiplication
    $tLengthSquared \leftarrow \mathbf{t} \cdot \mathbf{t}$ // Scalar product
    if $tLengthSquared \approx 0$ **then error** "breakdown: $\mathbf{t} \approx \mathbf{0}$"
    $\omega \leftarrow (\mathbf{t} \cdot \mathbf{s}) / tLengthSquared$ // Scalar product
    if $\omega \approx 0$ **then error** "breakdown: $\omega \approx 0$"
    $\epsilon \leftarrow 0$
    for $i \leftarrow 0$ **to** $n - 1$ **do**
        $change \leftarrow \alpha p[i] + \omega s[i]$
        $\epsilon \leftarrow \epsilon + |change|$
        $x[i] \leftarrow x[i] + change$
    if $\epsilon \leq \tau$ **then return** $\mathbf{x}$
    $\mathbf{s} \leftarrow \mathbf{t} + (-\omega) \cdot \mathbf{r}$ // Scaled vector addition
    $\rho' \leftarrow \rho$

Algorithm 5.10 Uniformization.

Input: infinitesimal generator $Q \in \mathbb{R}^{n \times n}$, initial probability vector $\boldsymbol{\pi}_0 \in \mathbb{R}^n$, truncation parameters $k_{\text{left}}, k_{\text{right}} \in \mathbb{N}$, weights $w_{\text{left}} \in \mathbb{R}$, $\mathbf{w} \in \mathbb{R}^{k_{\text{right}} - k_{\text{left}}}$, scaling constant $W \in \mathbb{R}$, tolerance $\tau > 0$

Output: instantenous or accumulated probability vector $\mathbf{x} \in \mathbb{R}^n$

allocate $\mathbf{x}, \mathbf{p}, \mathbf{q} \in \mathbb{R}^n$
$\alpha^{-1} \leftarrow 1/\max_i |a[i,i]|$
$\mathbf{p} \leftarrow \boldsymbol{\pi}_0$
if $w_{\text{left}} = 0$ **then** $\mathbf{x} \leftarrow \mathbf{0}$ **else** $\mathbf{x} \leftarrow w_{\text{left}} \cdot \mathbf{p}$ // Vector scaling
for $k \leftarrow 1$ **to** k_{right} **do**
    $\mathbf{q} \leftarrow \mathbf{p}Q$ // Vector-matrix product
    $\mathbf{q} \leftarrow \alpha^{-1} \cdot \mathbf{q}$ // In-place vector scaling
    $\mathbf{q} \leftarrow \mathbf{q} + \mathbf{q}$ // In-place vector addition
    if $\|\mathbf{q} - \mathbf{p}\| \leq \tau$ **then**
        $\mathbf{x} \leftarrow \mathbf{x} + \left(\sum_{l=k}^{k_{\text{right}}} w[l - k_{\text{left}}]\right) \cdot \mathbf{q}$ // In-place scaled vector addition
        break
    if $k < k_{\text{left}} \wedge w_{\text{left}} \neq 0$ **then** $\mathbf{x} \leftarrow \mathbf{x} + w_{\text{left}} \cdot \mathbf{q}$ // In-place scaled vector addition
    else if $k \geq k_{\text{left}}$ **then** $\mathbf{x} \leftarrow \mathbf{x} + w[k - k_{\text{left}}] \cdot \mathbf{q}$ // In-place scaled vector addition
    Swap the references to $\mathbf{p}$ and $\mathbf{q}$
$\mathbf{x} \leftarrow W^{-1} \cdot \mathbf{x}$ // In-place vector scaling
return $\mathbf{x}$

Algorithm 5.11 Burak's algorithm for calculating the Poisson weights.

Input: Poisson rate $\lambda = \alpha t$, tolerance $\tau > 10^{-50}$
Output: truncation parameters $k_{\text{left}}, k_{\text{right}} \in \mathbb{N}$, weights $\mathbf{w} \in \mathbb{R}^{k_{\text{right}} - k_{\text{left}}}$, scaling constant $W \in \mathbb{R}$

$M_w \leftarrow 30, M_a \leftarrow 44, M_s \leftarrow 21$
$m \leftarrow \lfloor \lambda \rfloor, tSize \leftarrow \lfloor M_w \sqrt{\lambda} + M_a \rfloor, tStart \leftarrow \max\{m + M_s - \lfloor tSize/2 \rfloor, 0\}$
allocate tWeights $\in \mathbb{R}^{tSize}$
$tWeights[m - tStart] \leftarrow 2^{176}$
for $j \leftarrow m - tStart$ **downto** 1 **do**
    $tWeights[j-1] = (j + tStart)\, tWeights[j]/\lambda$
for $j \leftarrow m - tStart + 1$ **to** $tSize$ **do**
    $tWeights[j+1] = \lambda\, tWeights[j]/(j + tStart)$
$W \leftarrow 0$
for $j \leftarrow 0$ **to** $m - tStart - 1$ **do**
    $W \leftarrow W + tWeights[j]$
$sum1 \leftarrow 0$ // Avoid adding small numbers to larger numbers
for $j \leftarrow tSize - 1$ **downto** $m - tStart$ **do**
    $sum1 \leftarrow sum1 + tWeights[j]$
$W \leftarrow W + sum1, threshold \leftarrow W\tau/2, cdf \leftarrow 0, i \leftarrow 0$
while $cdf < threshold$ **do**
    $cdf \leftarrow cdf + tWeights[i]$
    $i \leftarrow i + 1$
$k_{\text{left}} \leftarrow tStart + i, cdf \leftarrow 0, i \leftarrow tSize - 1$
while $cdf < threshold$ **do**
    $cdf \leftarrow cdf + tWeights[i]$
    $i \leftarrow i - 1$
$k_{\text{right}} \leftarrow tStart + i$
allocate $\mathbf{w} \in \mathbb{R}^{k_{\text{right}} - k_{\text{left}}}$
for $j \leftarrow k_{\text{left}}$ **to** k_{right} **do**
    $w[j - k_{\text{left}}] \leftarrow tWeights[j - tStart]$
return $k_{\text{left}}, k_{\text{right}}, \mathbf{w}, W$

Algorithm 5.12 TR-BDF2 for transient analysis.

Input: infinitesimal generator $Q \in \mathbb{R}^{n\times n}$, initial distribution $\boldsymbol{\pi}_0$, mission time $t > 0$, tolerance $\tau > 0$
Output: transient distribution $\boldsymbol{\pi}(t)$

allocate $\boldsymbol{\pi}_k, \boldsymbol{\pi}_{k+\gamma}, \boldsymbol{\pi}_{k+1}, \mathbf{d}_k, \mathbf{d}_{k+1}, \mathbf{y} \in \mathbb{R}^n$
$maxIncrease \leftarrow 10, leastDecrease \leftarrow 0.9$
$timeLeft \leftarrow t, h \leftarrow 1/\max_i |a[i,i]|, \gamma \leftarrow 2-\sqrt{2}, C \leftarrow \left|\frac{-3\gamma^4+4\gamma-2}{24-12\gamma}\right|, errorSum \leftarrow 0$
$\boldsymbol{\pi}_k \leftarrow \boldsymbol{\pi}_0$
$\mathbf{d}_k \leftarrow \boldsymbol{\pi}_k Q$ // Vector-matrix product
while $timeLeft > 0$ **do**
  $stepFailed \leftarrow$ false, $h \leftarrow \min\{h, timeLeft\}$
  while true **do**
    /* TR step */
    $\mathbf{y} \leftarrow 2 \cdot \boldsymbol{\pi}_k$ // Vector scaling
    $\mathbf{y} \leftarrow \mathbf{y} + \gamma h \cdot \mathbf{d}_k$ // In-place vector addition
    Solve $\boldsymbol{\pi}_{k+\gamma}(2I + -\gamma h Q) = \mathbf{y}$ for $\boldsymbol{\pi}_{k+\gamma}$ with initial guess $\boldsymbol{\pi}_k$
    /* BDF2 step */
    $\mathbf{y} \leftarrow -\frac{(1-\gamma)^2}{\gamma} \cdot \boldsymbol{\pi}_k$ // Vector scaling
    $\mathbf{y} \leftarrow \frac{1}{\gamma} \cdot \boldsymbol{\pi}_{k+\gamma}$ // In-place scaled vector addition
    Solve $\boldsymbol{\pi}_{k+1}((2-\gamma)I + (\gamma-1)hQ) = \mathbf{y}$ for $\boldsymbol{\pi}_{k+1}$ with initial guess $\boldsymbol{\pi}_{k+\gamma}$
    /* Error control and step size estimation */
    $\mathbf{y} \leftarrow -\frac{1}{\gamma}\mathbf{d}_k$ // Vector scaling
    $\mathbf{y} \leftarrow \mathbf{y} + \frac{1}{\gamma(1-\gamma)}\boldsymbol{\pi}_{k+\gamma} Q$ // In-place scaled addition of vector-matrix product
    $\mathbf{d}_{k+1} \leftarrow \boldsymbol{\pi}_{k+1} Q$ // Vector-matrix product
    $\mathbf{y} \leftarrow \mathbf{y} + \left(-\frac{1}{1-\gamma}\right)\mathbf{d}_{k+1}$ // In-place scaled vector addition
    $LTE \leftarrow 2Ch\|\mathbf{y}\|, localTol \leftarrow (\tau - errorSum)/timeLeft \cdot h$
    if $LTE < localTol$ **then** // Successful step
      $timeLeft \leftarrow timeLeft - h, errorSum \leftarrow errorSum + LTE$
      // Do not try to increase h after a failed step
      if $\neg stepFailed$ **then** $h \leftarrow h \cdot \min\{maxIncrease, \sqrt[3]{localTol/LTE}\}$
      break
    $stepFailed \leftarrow$ true, $h \leftarrow h \cdot \min\{leastDecrease, \sqrt[3]{localTol/LTE}\}$
  Swap the references to $\boldsymbol{\pi}_k, \boldsymbol{\pi}_{k+1}$ and $\mathbf{d}_k, \mathbf{d}_{k+1}$
return $\boldsymbol{\pi}_k$

Algorithm 5.13 Parallel block vector-matrix product.

Input: block vector $\mathbf{b} \in \mathbb{R}^{n_0+n_1+\cdots+n_{k-1}}$,
block matrix $A \in \mathbb{R}^{(n_0+n_1+\cdots+n_{k-1})\times(m_0+m_1+\cdots+m_{l-1})}$
Output: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^{m_0+m_1+\cdots+m_{l-1}}$

allocate $\mathbf{c} \in \mathbb{R}^{m_0+m_1+\cdots+m_{l-1}}$
parallel for $j \leftarrow 0$ **to** $l-1$ **do**
  $\mathbf{c}[j] \leftarrow \mathbf{0}$
  for $i \leftarrow 0$ **to** $k-1$ **do**
    $\mathbf{c}[j] \leftarrow \mathbf{c}[j] + \mathbf{b}[i]A[i,j]$ // Scaled addition of vector-matrix product

Algorithm 5.14 Product of a vector with a linear combination matrix.

Input: $\mathbf{b} \in \mathbb{R}^n$, $A = \nu_0 A_0 + \nu_1 A_1 + \cdots + \nu_{k-1} A_{k-1}$, where $A_h \in \mathbb{R}^{n\times m}$
Output: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^m$

allocate $\mathbf{c} \in \mathbb{R}^m$ if no target buffer is provided
$\mathbf{c} \leftarrow \mathbf{0}$
for $h \leftarrow 0$ **to** $k-1$ **do**
  $\mathbf{c} \leftarrow \nu_h \cdot \mathbf{b}A_h$ // In-place scaled addition of vector-matrix product
return $\mathbf{c}$

Algorithm 5.15 The SHUFFLE algorithm for vector-matrix multiplication.

Input: $\mathbf{b} \in \mathbb{R}^{n_0 n_1 \cdots n_{k-1}}$, $A = A^{(0)} \otimes A^{(1)} \otimes \cdots \otimes A^{(k-1)}$, where $A^{(h)} \in \mathbb{R}^{n_h \times m_h}$
Output: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^{m_0 m_1 \cdots m_{k-1}}$

$n \leftarrow n_0 n_1 \cdots n_{k-1}, \quad m \leftarrow m_0 m_1 \cdots m_{k-1}$
$tempLength \leftarrow \max_{h=-1,0,1,\ldots,k-1} \prod_{f=0}^{h} m_f \prod_{f=h+1}^{k-1} n_f$
allocate $\mathbf{x}, \mathbf{x}'$ with at least $tempLength$ elements
$\mathbf{x}[0{:}1{:}n] \leftarrow \mathbf{b}, \quad i_{\text{left}} \leftarrow 1, \quad i_{\text{right}} \leftarrow \prod_{h=1}^{k-1} n_h$
for $h \leftarrow 0$ **to** $k-1$ **do**
  if $A^{(h)}$ is not an identity matrix **then**
    $i_{\text{base}} \leftarrow 0, j_{\text{base}} \leftarrow 0$
    for $il \leftarrow 0$ **to** $i_{\text{left}} - 1$ **do**
      for $ir \leftarrow 0$ **to** $i_{\text{right}} - 1$ **do**
        $\mathbf{x}'[j_{\text{base}}{:}m_h{:}i_{\text{right}}] \leftarrow \mathbf{x}[i_{\text{base}}{:}n_h{:}i_{\text{right}}]A^{(h)}$
        $i_{\text{base}} \leftarrow i_{\text{base}} + n_h i_{\text{right}}, \quad j_{\text{base}} \leftarrow j_{\text{base}} + m_h i_{\text{right}}$
    Swap the references to $\mathbf{x}$ and $\mathbf{x}'$
  $i_{\text{left}} \leftarrow i_{\text{left}} \cdot m_h$
  if $h \neq k-1$ **then** $i_{\text{right}} \leftarrow i_{\text{right}}/n_{h+1}$
return $\mathbf{c} = \mathbf{x}[0{:}1{:}m]$

Chapter 6

Evaluation

6.1 Testing

When developing an algorithm library for formal analysis of safety critical systems it is vital to verify the correctness of the implementation. Since the complexity of the code base makes formal verification difficult we confined ourselves to rigorously testing the functionalities provided by the library.

6.1.1 Combinatorial testing

As described in Chapter 5 algorithms use the common vector and matrix interfaces to perform various operations. This makes the used storage techniques transparent which in turn makes the code base more concise, reusable and less prone to errors.

The most important requirement against the datastructure operations is mathematical correctness regardless of the storage technique used. Considering the number of implementations for a given interface and the previous requirement we used a simple unit testing pattern (sometimes refered to as interface testing pattern **TODO: reference**) as the core building block for the datastructure testing.

The basic idea behind this pattern is to write unit tests for interface operations without any knowledge about the concrete implementation. Hiding implemetation details can be achieved in a number of ways. Some unit testing frameworks (like *NUnit*) support the usage of generic test classes and running them for multiple concrete types.

Since most of the time multiple instances of different types of interface implementations are needed in a single unit test we choose a more flexible approach for hiding implementation details. This approach is based on class inheritance and abstract factory methods. Whenever we need an instance for a given interface we delegate the instantiation to an abstract factory method in the test class.

A drawback of this approach is that the test class itself becomes abstract so we can't

run the tests inside it directly. However we can easily inherit from the base test class and implement the abstract factory methods in any way we'd like. But the most important advantage of this approach manifests itself when we apply the virtual modifier to one or more unit tests in the base class. This way we can completely override tests in the derived classes if needed based on the types of the interface implementations. So the first step in testing the datastructure library was to implement these abstract unit tests that operate on an interface level.

Abstract tests

In order to make sure we cover the most possible usage scenarios of the datastructure we followed some common testing techniques. As a first step we used equivalence partitioning to identify the valid and invalid ranges of the parameters of the operations. Next we implemented the parameter value checks in interface code contract classes using Microsoft's Code Contract library. This enabled us to implement the parameter check logic in one place for an operation making the code more maintainable. Moreover every class implementing a datastructure interface and it's operations will automatically contain these logics if code contracts are enabled. Code contracts can be disabled if needed resulting in a perfomance boost for the datastructure library since the parameter checks are skipped.

Writing unit tests for valid parameter values was straighforward since it's possible to cover multiple valid parameter ranges with a single unit test. However testing for invalid parameter values requires some care. We must ensure that there is only one invalid parameter per unit test so one error doesn't obscure the other. This significantly increases the number of unit tests and the possibility that we forget to test an invalid parameter range. Therefore we aimed to gather every possible invalid parameter range automatically.

For this purpose we used Microsoft's IntelliTest tool which assists in automating white-box and unit-testing. IntelliTest automatically generates unit tests using constraint satisfaction problem solving based on the source code of the method under test. Using IntelliTest on our interface code contract classes provided us with many invalid parameter values which we could use in our abstract unit tests.

Concrete tests

Once the abstract unit tests were implemented the next step was to create the derived classes for every storage combination and implement the abstract factory methods. Since the number of possible combinations were too many to implement manually we used Microsoft's Text Template Transformation Toolkit (T4) to generate the derived classes. The created template files provide ways to modify the behavior of abstract tests (through simple regular expression based configuration files) and to decrease the

number of generated test by using pairwise testing instead of full combinatorial testing of implementation combinations. To generate the combinations for pairwise testing we used the ACTS tool.

As a result of this testing process more than 78 000 unit tests were generated using full combinatorial testing (more than 18 000 with pairwise testing) which together with the behavior configuration files serve as a quasi-formal specification for the expected behavior of future and modified implementations (e.g. perfomance optimization). Breaking changes in implementation should either be rejected or the test suite and configuration files should be revised as specification change. Every unit test was executed sucessfully for both sequential and parallel operation implementations.

6.1.2 Software redundancy based testing

Apart from testing the datastructure operation implementations it is vital to test the correctness of higher level algorithms used in the analysis workflow, e.g. the linear equation solver algorithms. Testing every implemented algorithm one by one with unit tests would be tremendous work and it can't be easily automated (or maintained in case of manual testing). Moreover every algorithm is used as part of a bigger workflow which raises the question of compatibility of algorithms during an analysis.

As described in Section 3.2 for almost every step of the workflow numerous algorithms are available.

> **Observation 6.1** The result of a performance analysis (e.g. reward calculation) is mathematically independent of the used analysis workflow. It only depends on the possible behaviors of the system and the definition of the required performance measure. Two results calculated by using two different analysis methods can only differ from eachother due to the numercial precision properties of the used algorithms.

Combining our fully configurable workflow with Observation 6.1 presents a new approach for testing the algorithm implementations in a maintainable and almost automatic manner. We can take advantage of the concept of software redundancy commonly used in safety critical applications. The main idea behind software redundancy is to perform a calculation multiple times with usually fundamentally different algorithms (often developed by independent teams) thus minimizing the possibility of common mode failures. After the calculations a voting component examines whether every algorithm calculated the same result. If that's not the case then one or more of the algorithms are incorrect.

The building block for this testing phase consists of running our analysis workflow for a given configuration and saving the calculated results (reward and sensitivity values). We generated 588 mathematically consistent configurations in total, executed

them for our running example (Figure 2.4), multiple benchmark models and case studies. Finally we examined the maximum absolute difference of the calculated results as an error indicator for each performance measure in each model as presented in the next sections.

6.2 Measurements

6.2.1 Benchmark models

Resource sharing

Kanban

Dining philosophers

6.2.2 Case studies

Performability of clouds

6.3 Baselines

6.3.1 PRISM

6.3.2 SMART

6.4 Results

Chapter 7

Conclusion

7.1 Future work

References

[1] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. “Approximative symbolic model checking of continuous-time Markov chains”. In: *CONCUR’99 Concurrency Theory*. Springer, 1999, pp. 146–161.

[2] Randolph E. Bank, William M. Coughran Jr., Wolfgang Fichtner, Eric Grosse, Donald J. Rose, and R. Kent Smith. “Transient Simulation of Silicon Devices and Circuits”. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 4.4 (1985), pp. 436–451. DOI: 10.1109/TCAD.1985.1270142.

[3] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. Vol. 43. Siam, 1994.

[4] Falko Bause, Peter Buchholz, and Peter Kemper. “A Toolbox for Functional and Quantitative Analysis of DEDS”. In: *Computer Performance Evaluation: Modelling Techniques and Tools, 10th International Conference, Tools ’98, Palma de Mallorca, Spain, September 14-18, 1998, Proceedings*. Vol. 1469. Lecture Notes in Computer Science. Springer, 1998, pp. 356–359. DOI: 10.1007/3-540-68061-6_32.

[5] Anne Benoit, Brigitte Plateau, and William J Stewart. “Memory efficient iterative methods for stochastic automata networks”. In: (2001).

[6] Anne Benoit, Brigitte Plateau, and William J. Stewart. “Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems”. In: *Future Generation Comp. Syst.* 22.7 (2006), pp. 838–847. DOI: 10.1016/j.future.2006.02.006.

[7] Andrea Bianco and Luca De Alfaro. “Model checking of probabilistic and nondeterministic systems”. In: *Foundations of Software Technology and Theoretical Computer Science*. Springer. 1995, pp. 499–513.

[8] James T. Blake, Andrew L. Reibman, and Kishor S. Trivedi. “Sensitivity Analysis of Reliability and Performability Measures for Multiprocessor Systems”. In: *SIGMETRICS*. 1988, pp. 177–186. DOI: 10.1145/55595.55616.

[9] BlueBit Software. *.NET Matrix Library 6.1*. Accessed October 26, 2015. URL: http://www.bluebit.gr/NET/.

[10] Peter Buchholz. "Hierarchical Structuring of Superposed GSPNs". In: *IEEE Trans. Software Eng.* 25.2 (1999), pp. 166–181. DOI: 10.1109/32.761443.

[11] Peter Buchholz, Gianfranco Ciardo, Susanna Donatelli, and Peter Kemper. "Complexity of Memory-Efficient Kronecker Operations with Applications to the Solution of Markov Models". In: *INFORMS Journal on Computing* 12.3 (2000), pp. 203–222. DOI: 10.1287/ijoc.12.3.203.12634.

[12] Peter Buchholz and Peter Kemper. "Kronecker based matrix representations for large Markov models". In: *Validation of Stochastic Systems*. Springer, 2004, pp. 256–295.

[13] Peter Buchholz and Peter Kemper. "On generating a hierarchy for GSPN analysis". In: *SIGMETRICS Performance Evaluation Review* 26.2 (1998), pp. 5–14. DOI: 10.1145/288197.288202.

[14] Maciej Burak. "Multi-step Uniformization with Steady-State Detection in Non-stationary M/M/s Queuing Systems". In: *CoRR* abs/1410.0804 (2014). URL: http://arxiv.org/abs/1410.0804.

[15] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Rohit Singh. "Measuring and Synthesizing Systems in Probabilistic Environments". In: *J. ACM* 62.1 (2015), 9:1–9:34. DOI: 10.1145/2699430.

[16] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. "Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications". In: *IEEE Trans. Computers* 42.11 (1993), pp. 1343–1360. DOI: 10.1109/12.247838.

[17] Piotr Chrzastowski-Wachtel. "Testing Undecidability of the Reachability in Petri Nets with the Help of 10th Hilbert Problem". In: *Application and Theory of Petri Nets 1999, 20th International Conference, ICATPN '99, Williamsburg, Virginia, USA, June 21-25, 1999, Proceedings*. Vol. 1639. Lecture Notes in Computer Science. Springer, 1999, pp. 268–281. DOI: 10.1007/3-540-48745-X_16.

[18] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. *Saturation: an efficient iteration strategy for symbolic state—space generation*. Springer, 2001.

[19] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. "The saturation algorithm for symbolic state-space exploration". In: *Int. J. Softw. Tools Technol. Transf.* 8.1 (2006), pp. 4–25. DOI: http://dx.doi.org/10.1007/s10009-005-0188-7.

[20] Gianfranco Ciardo, Jogesh K. Muppala, and Kishor S. Trivedi. "On the Solution of GSPN Reward Models". In: *Perform. Eval.* 12.4 (1991), pp. 237–253. DOI: 10.1016/0166-5316(91)90003-L.

[21] Ricardo M. Czekster, César A. F. De Rose, Paulo Henrique Lemelle Fernandes, Antonio M. de Lima, and Thais Webber. “Kronecker descriptor partitioning for parallel algorithms”. In: *Proceedings of the 2010 Spring Simulation Multiconference, SpringSim 2010, Orlando, Florida, USA, April 11-15, 2010*. SCS/ACM, 2010, p. 242. ISBN: 978-1-4503-0069-8. URL: http://dl.acm.org/citation.cfm?id=1878537.1878789.

[22] Dániel Darvas. *Szaturáció alapú automatikus modellellenőrző fejlesztése aszinkron rendszerekhez [in Hungarian]*. 1st prize. 2010. URL: http://petridotnet.inf.mit.bme.hu/publications/OTDK2011_Darvas.pdf.

[23] Susanna Donatelli. “Superposed Generalized Stochastic Petri Nets: Definition and Efficient Solution”. In: *Application and Theory of Petri Nets 1994, 15th International Conference, Zaragoza, Spain, June 20-24, 1994, Proceedings*. Vol. 815. Lecture Notes in Computer Science. Springer, 1994, pp. 258–277. DOI: 10.1007/3-540-58152-9_15.

[24] Susanna Donatelli. “Superposed stochastic automata: a class of stochastic Petri nets with parallel solution and distributed state space”. In: *Performance Evaluation* 18.1 (1993), pp. 21–36.

[25] Extreme Optimization. *Numerical Libraries for .NET*. Accessed October 26, 2015. URL: http://www.extremeoptimization.com/VectorMatrixFeatures.aspx.

[26] Fault Tolerant Systems Research Group, Budapest University of Technology and Economics. *The PetriDotNet webpage*. Accessed October 23, 2015. URL: https://inf.mit.bme.hu/en/research/tools/petridotnet.

[27] Paulo Fernandes, Brigitte Plateau, and William J. Stewart. “Numerical Evaluation of Stochastic Automata Networks”. In: *MASCOTS ’95, Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, January 10-18, 1995, Durham, North Carolina, USA*. IEEE Computer Society, 1995, pp. 179–183. DOI: 10.1109/MASCOT.1995.378690.

[28] Paulo Fernandes, Ricardo Presotto, Afonso Sales, and Thais Webber. “An Alternative Algorithm to Multiply a Vector by a Kronecker Represented Descriptor”. In: *21st UK Performance Engineering Workshop*. 2005, pp. 57–67.

[29] Bennett L. Fox and Peter W. Glynn. “Computing Poisson Probabilities”. In: *Commun. ACM* 31.4 (1988), pp. 440–445. DOI: 10.1145/42404.42409.

[30] Robert E Funderlic and Carl Dean Meyer. “Sensitivity of the stationary distribution vector for an ergodic Markov chain”. In: *Linear Algebra and its Applications* 76 (1986), pp. 1–17.

[31] Stephen Gilmore and Jane Hillston. “The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling”. In: *Computer Performance Evaluation, Modeling Techniques and Tools, 7th International Conference, Vienna, Austria, May 3-6, 1994, Proceedings*. Vol. 794. Lecture Notes in Computer Science. Springer, 1994, pp. 353–368. DOI: 10.1007/3-540-58021-2_20.

[32] Winfried K. Grassmann. “Transient solutions in markovian queueing systems”. In: *Computers & OR* 4.1 (1977), pp. 47–53. DOI: 10.1016/0305-0548(77)90007-7.

[33] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1997. ISBN: 9781611970937. URL: https://books.google.hu/books?id=IX9rrFe1YLQC.

[34] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. Accessed October 26, 2015. 2010. URL: http://eigen.tuxfamily.org.

[35] Boudewijn R Haverkort. “Matrix-geometric solution of infinite stochastic Petri nets”. In: *Computer Performance and Dependability Symposium, 1995. Proceedings., International*. IEEE. 1995, pp. 72–81.

[36] *International Workshop on Timed Petri Nets, Torino, Italy, July 1-3, 1985*. IEEE Computer Society, 1985. ISBN: 0-8186-0674-6.

[37] Ilse CF Ipsen and Carl D Meyer. “Uniform stability of Markov chains”. In: *SIAM Journal on Matrix Analysis and Applications* 15.4 (1994), pp. 1061–1074.

[38] David N Jansen. “Understanding Fox and Glynn’s “Computing Poisson probabilities”. In: (2011).

[39] Peter Kemper. “Numerical Analysis of Superposed GSPNs”. In: *IEEE Trans. Software Eng.* 22.9 (1996), pp. 615–628. DOI: 10.1109/32.541433.

[40] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. “A unified sparse matrix data format for modern processors with wide SIMD units”. In: *CoRR* abs/1307.6209 (2013). URL: http://arxiv.org/abs/1307.6209.

[41] Amy Nicole Langville and William J. Stewart. “Testing the Nearest Kronecker Product Preconditioner on Markov Chains and Stochastic Automata Networks”. In: *INFORMS Journal on Computing* 16.3 (2004), pp. 300–315. DOI: 10.1287/ijoc.1030.0041.

[42] Francesco Longo and Marco Scarpa. “Two-layer Symbolic Representation for Stochastic Models with Phase-type Distributed Events”. In: *Intern. J. Syst. Sci.* 46.9 (2015), pp. 1540–1571. DOI: 10.1080/00207721.2013.822940.

[43] Marco Ajmone Marsan. “Stochastic Petri nets: an elementary introduction”. In: *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets, held in Venice, Italy in June 1988, selected papers*. Vol. 424. Lecture Notes in Computer Science. Springer, 1988, pp. 1–29. DOI: 10.1007/3-540-52494-0_23.

[44] Marco Ajmone Marsan, Gianfranco Balbo, Andrea Bobbio, Giovanni Chiola, Gianni Conte, and Aldo Cumani. “The Effect of Execution Policies on the Semantics and Analysis of Stochastic Petri Nets”. In: *IEEE Trans. Software Eng.* 15.7 (1989), pp. 832–846. DOI: 10.1109/32.29483.

[45] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. “A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems”. In: *ACM Trans. Comput. Syst.* 2.2 (1984), pp. 93–122. DOI: 10.1145/190.191.

[46] Math.NET. *Math.NET Numerics webpage*. Accessed October 26, 2015. URL: http://numerics.mathdotnet.com/.

[47] Carl D Meyer. “Stochastic complementation, uncoupling Markov chains, and the theory of nearly reducible systems”. In: *SIAM review* 31.2 (1989), pp. 240–272.

[48] Aad PA van Moorsel and William H Sanders. “Adaptive uniformization”. In: *Stochastic Models* 10.3 (1994), pp. 619–647.

[49] Tadao Murata. “Petri nets: Properties, analysis and applications”. In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.

[50] M. Neuts. “Probability distributions of phase type”. In: *Liber Amicorum Prof. Emeritus H. Florin*. University of Louvain, 1975, pp. 173–206.

[51] RJ Plemmons and A Berman. *Nonnegative matrices in the mathematical sciences*. Academic Press, New York, 1979.

[52] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[53] S. Rácz, Á. Tari, and M. Telek. “MRMSolve: Distribution estimation of Large Markov reward models”. In: *Tools 2002*. Springer, LNCS 2324, 2002, pp. 72–81.

[54] S. Rácz and M. Telek. “Performability Analysis of Markov Reward Models with Rate and Impulse Reward”. In: *Int. Conf. on Numerical solution of Markov chains*. 1999, pp. 169–187.

[55] A. V. Ramesh and Kishor S. Trivedi. “On the Sensitivity of Transient Solutions of Markov Models”. In: *SIGMETRICS*. 1993, pp. 122–134. DOI: 10.1145/166955.166998.

[56] Andrew L. Reibman and Kishor S. Trivedi. “Numerical transient analysis of markov models”. In: *Computers & OR* 15.1 (1988), pp. 19–36. DOI: 10.1016/0305-0548(88)90026-3.

[57] Andrew Reibman, Roger Smith, and Kishor Trivedi. “Markov and Markov reward model transient analysis: An overview of numerical approaches”. In: *European Journal of Operational Research* 40.2 (1989), pp. 257–267.

[58] Pierre Roux and Radu Siminiceanu. “Model Checking with Edge-valued Decision Diagrams”. In: *Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings*. Vol. NASA/CP-2010-216215. NASA Conference Proceedings. 2010, pp. 222–226.

[59] *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets, Oxford, UK, June 1986*. Vol. 266. Lecture Notes in Computer Science. Springer, 1987. ISBN: 3-540-18086-9.

[60] Youcef Saad and Martin H Schultz. “GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems”. In: *SIAM Journal on scientific and statistical computing* 7.3 (1986), pp. 856–869.

[61] Yousef Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[62] Conrad Sanderson. “Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments”. In: (2010).

[63] Peter Sonneveld. “CGS, a fast Lanczos-type solver for nonsymmetric linear systems”. In: *SIAM journal on scientific and statistical computing* 10.1 (1989), pp. 36–52.

[64] Peter Sonneveld and Martin B van Gijzen. “IDR (s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations”. In: *SIAM Journal on Scientific Computing* 31.2 (2008), pp. 1035–1062.

[65] William J Stewart. *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton University Press, 2009.

[66] Williams J Stewart. *Introduction to the numerical solutions of Markov chains*. Princeton Univ. Press, 1994.

[67] Enrique Teruel, Giuliana Franceschinis, and Massimiliano De Pierro. “Well-Defined Generalized Stochastic Petri Nets: A Net-Level Method to Specify Priorities”. In: *IEEE Trans. Software Eng.* 29.11 (2003), pp. 962–973. DOI: 10.1109/TSE.2003.1245298.

[68] Henk A Van der Vorst. “Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems”. In: *SIAM Journal on scientific and Statistical Computing* 13.2 (1992), pp. 631–644.