SZAKDOLGOZAT FELADAT

Marussy Kristóf
mérnök informatikus hallgató részére

Konfigurálható numerikus módszerek sztochasztikus modellekhez

A kritikus rendszerek – biztonságkritikus, elosztott és felhő-alapú alkalmazások – 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 és leginkább eltérő karakterisztikája 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 hallgató feladata áttekinteni az irodalmat és megvizsgálni az ismert algoritmusokat.

A feladat megoldása a következő lépésekből áll:

1. Mutassa be az irodalomban ismert, markovi sztochasztikus rendszerek állandósult állapotbeli és tranziens viselkedésének vizsgálatára alkalmas numerikus algoritmusokat.
2. Az irodalom alapján implementáljon kiválasztott tranziens és állandósult állapotbeli analízis algoritmusokat.
3. Hasonlítsa össze futási idő és tárhely komplexitás szempontjából az implementált algoritmusokat.
4. Értékelje a megoldást és vizsgálja meg a továbbfejlesztési lehetőségeket.

Tanszéki konzulens: Vörös András, tudományos segédmunkatárs
Molnár Vince, doktorandusz
Külső konzulens: dr. Telek Miklós, egyetemi tanár

Budapest, 2015. október 7.

Dr. Jobbágy Ákos
egyetemi tanár
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

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

Kristóf Marussy

Configurable Numerical Solutions for Stochastic Models

BSc Thesis

Supervisors:

dr. Miklós Telek
Vince Molnár
András Vörös

Budapest, 2015

Contents

Kivonat 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 és leginkább eltérő karakterisztikája 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 dolgozatban bemutatjuk az irodalomban ismert, markovi sztochasztikus rendszerek állandósult állapotbeli és tranziens viselkedésének vizsgálatára alkalmas numerikus algoritmusokat. Az algoritmusokat konfigurálható adatstruktúrával és lineáris algebrai műveletekkel valósítottuk meg.

A bevezetett konfigurálható sztochasztikus analízis keretrendszer lehetővé teszi a sztochasztikus viselkedéseket leíró különböző mátrix-dekompozíciók és az analízis algoritmusok használatát állandósult állapotbeli, tranziens, első hiba várható idő és érzékenységvizsgálatok elvégzésére. Az elkészített eszközt integráltuk a PetriDotNet modellező szoftverrel.

Módszerünk gyakorlati alkalmazhatóságát szintetikus és ipari modelleken végzett mérésekkel igazoljuk.

Kulcsszavak aszinkron rendszerek, teljesítményvizsgálat, sztochasztikus modell, numerikus módszerek, érzékenységvizsgálat

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.

In this thesis we present numerical solution algorithms for the steady state and transient analysis of Markovian models. Various algorithms were implemented with configurable data structure and linear algebra operations.

Our framework provides configurable stochastic analysis: an approach is introduced to combine different matrix representations of stochastic behaviors with numerical solution algorithms for steady-state, transient, mean-time-to-first-failure and sensitivity problems.

The goal of our work is to introduce a framework that facilitates the analysis of complex, stochastic systems by combining the advantages of compact matrix representations and various numerical algorithms. The analysis tool is integrated into the PetriDotNet modeling application.

Benchmarks and industrial case studies are used to evaluate the applicability of our approach.

Keywords asynchronous systems, performance analysis, stochastic model, numeric methods, sensitivity analysis

Hallgatói nyilatkozat

Alulírott **Marussy Kristóf** szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2015. december 11.

.....
Marussy Kristóf

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 numerical backed for the solution of the various problems occurring in stochastic analysis of complex systems.

In [49] we introduced 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 behavior 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. The approach consists of a flexible data structure, which is complemented by a set of linear algebra operations configurable at runtime.

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. 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 algorithm development and optimization includes preliminary work on integrating the IDR(s)STAB(ℓ) numerical linear equations solver into our stochastic analysis framework, including modifications and tuning for matrices arising in steady-state Markovian analysis problems. To our best knowledge, ours is the first preliminary result in this area.

The analysis framework is integrated into the PetriDotNet modeling application for stochastic models in the *Stochastic Petri Net* (SPN) formalism.

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. More than 150 benchmark runs were performed with large models and industrial case studies to gather information about the performance and memory utilization characteristics of the analysis tool. In addition, 10 000 shorter benchmark runs were used to study the convergence behavior of our modification of the IDR(s)STAB(ℓ) algorithm.

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 and its numerical backend. Chapter 4 describes the developed data structure and configurable linear algebra operations. Chapter 5 presents numerical steady-state and transient analysis algorithms and their implementations in our framework, with special attention to the homogeneous linear equation systems arising from steady-state analysis. In Section 5.1.4 on page 52, we present the Krylov subspace solvers utilized, including our preliminary work integrating IDR(s)STAB(ℓ) on page 54. After describing the testing and validation methodologies applied to our framework in Chapter 6 as well as the benchmark results, we conclude our thesis in Chapter 7.

Chapter 2

Background

In this section we overview the basic formalisms and scope of our work. First, continuous-time Markov chains and Markovian reward analysis tasks are introduces. We also recall some foundations of Kronecker algebra. Lastly, stochastic automata networks are introduced, which are commonly used hierarchical formalism for Markovian models that are especially amenable to decomposition by Kronecker products.

2.1 Continuous-time Markov chains

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

Definition 2.1 A *Continuous-time Markov Chain* (CTMC) $X(t) \in S, t \geq 0$ over the finite state space $S = \{0, 1, \ldots, n-1\}$ is a continuous-time random process with the *Markovian* or memoryless property:

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

where $t_0 \leq t_1 \leq \cdots \leq t_k$ and $X(t_k)$ is a random variable denoting the current state of the CTMC at time t_k. A CTMC is said to be *time-homogeneous* 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-homogeneous CTMCs over finite state spaces. The state probabilities of these stochastic processes at time t form a

finite-dimensional vector $\boldsymbol{\pi}(t) \in \mathbb{R}^n$,

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

that satisfies the differential equation

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

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* $h_x = \inf\{h > 0 : X(t) = x, X(t+h) \neq x\}$ 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*.

- 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 exponentially 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 [64] and Stewart [83].

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

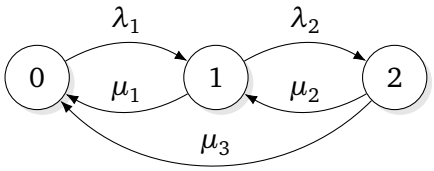
$$x = z_1, z_2, z_3, \ldots, z_{k-1}, z_k = y$$

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 satisfies the linear equation

$$\frac{\mathrm{d}\boldsymbol{\pi}}{\mathrm{d}t} = \boldsymbol{\pi}Q = \mathbf{0}, \quad \boldsymbol{\pi}\mathbf{1}^{\mathrm{T}} = 1. \tag{2.1}$$

Example 2.1 Figure 2.1 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.



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

Figure 2.1 Example CTMC with 3 states and its generator matrix.

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 from state 1 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 $\boldsymbol{\pi}_0$.

2.1.1 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 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 2.2 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 [67, 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

$$\mathbb{E}R(t) = \sum_{i=0}^{n-1} \pi(t)[i]r[i] = \boldsymbol{\pi}(t)\mathbf{r}^{\mathrm{T}}, \tag{2.2}$$

which requires the solution of the initial value problem [39, 70]

$$\frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t} = \boldsymbol{\pi}(t)Q, \quad \boldsymbol{\pi}(0) = \boldsymbol{\pi}_0 \tag{2.3}$$

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 (2.1) should be solved instead of eq. (2.3) in order to acquire the steady-state probability vector $\boldsymbol{\pi}$. The computation of the reward value proceeds by eq. (2.2) 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 [70]

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

Example 2.2 Let c_0, c_1 and c_2 denote operating costs per unit time associated with the states of the CTMC in Figure 2.1. 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.

2.1.2 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 $Q(\theta)$

and the reward rate vector $\mathbf{r}(\theta)$ may depend on some *parameters* $\theta \in \mathbb{R}^m$. 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 2.3 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 sensitivity of $\mathbb{E}R(t)$, the partial derivative of both sides of eq. (2.2) is taken, yielding

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

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 $V_i = \partial Q(\boldsymbol{\theta})/\partial\, \theta[i]$ is the partial derivative of the generator matrix [68]. 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

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

is solved [10].

Another type of sensitivity analysis considers *unstructured* small perturbations of the infinitesimal generator matrix Q instead of dependencies on parameters [36, 46]. This latter, unstructured analysis may be used to study the numerical stability and conditioning of the solutions of the Markov chain.

2.1.3 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 $(\boldsymbol{\pi}_U, Q_{UU})$ [62], 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:

$$MTFF = \mathbb{E}[TFF] = -\boldsymbol{\pi}_U Q_{UU}^{-1} \mathbf{1}^{\mathrm{T}}. \tag{2.4}$$

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

$$\mathbb{P}(X(TFF_{+0}) = y) = -\boldsymbol{\pi}_U Q_{UU}^{-1} \mathbf{q}_{UD'}^{\mathrm{T}}, \tag{2.5}$$

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'.

2.2 Kronecker algebra

In this section, we review the fundamentals of Kronecker algebra.

Definition 2.4 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 Kronecker 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

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

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

Definition 2.5 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 $C=A\oplus B\in\mathbb{R}^{n_1n_2\times m_1m_3}$, 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 2.3 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

$$A\otimes B=\begin{pmatrix}1\cdot0&1\cdot1&2\cdot0&2\cdot1\\1\cdot2&1\cdot0&2\cdot2&2\cdot0\\3\cdot0&3\cdot1&4\cdot0&4\cdot1\\3\cdot2&3\cdot0&4\cdot2&4\cdot0\end{pmatrix}=\begin{pmatrix}0&1&0&2\\2&0&4&0\\0&3&0&4\\6&0&8&0\end{pmatrix},$$

while their Kronecker sum is

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

2.3 Continuous-time stochastic automata networks

In this section, we present Continuous-time Stochastic Automata Networks (SAN), which are a common theoretical foundation for the study of structure continuous-time Markovian processes [13, 42], including Stochastic Petri Nets (SPN) [52], Generalized Stochastic Petri Nets (GSPN) [15] and stochastic activity networks [55].

For additional details on the study of decomposed Markov chains, we refer the reader to Dayar [30].

Definition 2.6 A *Continuous-time stochastic automata network* is a triple $SAN = \left(E, \left(A^{(j)}\right)_{j=0}^{J-1}, \lambda\right)$, where

- E is a finite set of synchronizing *events*,
- $A^{(j)} = (S^{(j)}, x_0^{(j)}, E^{(j)}, T^{(j)})$ is a *stochastic automaton*, such that $E^{(j)} \subseteq E$ and $E = \bigcup_{j=0}^{J-1} E^{(j)}$,
- $\lambda : E \to \mathbb{R}^+$ is an *event rate function*.

Definition 2.7 A *stochastic automaton* is a 4-tuple $A = (S, x_0, E, T)$, where

- S is a finite set of *states*,
- $x_0 \in S$ is the *initial state*,
- E is a finite set of synchronizing events,
- $T \subset E \times S \times S \times \mathbb{R}^+$ is the *local transition relation*, such that $(e, x, y, \mu) \in T$, written as $x \xrightarrow{e,\mu} y$, denotes a transition from x to y with rate μ synchronized on the event e. It is required that $x \xrightarrow{e,\mu} y, x \xrightarrow{e,\nu} y \implies \mu = \nu$, i.e. the rate of a transition is a (partial) function of its start and end states and synchronizing event.

Parenthesized superscripts will be used to denote elements of automatons of a SAN, e.g. $A^{(j)} = (S^{(j)}, x_0^{(j)}, E^{(j)}, T^{(j)})$ is the jth automaton of *SAN* with $|S^{(j)}| = n^{(j)}$ states.

The set of *potential states* of *SAN* is

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

i.e. the Cartesian product of the state spaces of the automata. Thus, *global states* are vectors $\mathbf{x} = (x^{(0)}, x^{(1)}, \ldots, x^{(J-1)})$. The initial global state is $\mathbf{x}_0 = (x^{(0)}, x^{(1)}, \ldots, x^{(J-1)})$.

The global state changes from $\mathbf{x} \in PS$ to $\mathbf{y} \in PS$ when the event $e \in E$ occurs,

$$\mathbf{x}\,[e\rangle\,\mathbf{y} \quad \iff \quad \text{for all } 0 \le j \le J-1 \begin{cases} (e, x^{(j)}, y^{(j)}, \mu^{(j)}) \in T^{(j)} & \text{if } e \in E^{(j)}, \\ x^{(j)} = y^{(j)} & \text{if } e \notin E^{(j)}. \end{cases}$$

The *support* of the event e is the set of automata which respond to it, $\operatorname{supp} e = \{j : e \in E^{(j)}\}$. If $\operatorname{supp} e = \{j\}$, e is *local* to $A^{(j)}$.

The events local to $A^{(j)}$ are $E_L^{(j)} = \{e : \operatorname{supp} e = \{j\}\}$. Events which affect other automata are $E_S^{(j)} = E^{(j)} \setminus E_L^{(j)}$ synchronizing events of $A^{(j)}$. The set of all local events is $E_L = \bigcup_{j=0}^{J-1} E_L^{(j)}$, while the set of all synchronizing events is $E_S = \bigcup_{j=0}^{J-1} E_S^{(j)} = E \setminus E_L$.

A state $\mathbf{y}$ is *reachable* from the state $\mathbf{x}$ (written as $\mathbf{x} \rightsquigarrow \mathbf{y}$) if there exists a sequence of states and events for some finite k such that

$$\mathbf{x} = \mathbf{x}_1 \, [e_{i_1}\rangle \, \mathbf{x}_2 \, [e_{i_2}\rangle \, \mathbf{x}_3 \, [e_{i_3}\rangle \cdots [e_{i_{k-1}}\rangle \, \mathbf{x}_{k-1} \, [e_{i_k}\rangle \, \mathbf{x}_k = \mathbf{y}.$$

The state $\mathbf{y} \in PS$ is in the *reachable state space* of *SAN* if $\mathbf{x}_0 \rightsquigarrow \mathbf{y}$, hence the reachable state space is

$$RS = \{\mathbf{y} \in PS : \mathbf{x}_0 \rightsquigarrow \mathbf{y}\} \subseteq PS.$$

The term *state space explosion* refers to the phenomenon that even small models may have a very large number of states. For example, if $n^{(j)} = c$ for all $0 \leq j \leq J-1$, $|PS| = c^J$, hence RS may contain $O(c^J)$ elements.

We will assume a bijection $RS \leftrightarrow \{0, 1, \ldots, n-1\}$ between the reachable state space and the natural numbers such that $\mathbf{x}_0 \mapsto 0$. Moreover, we will assume a bijection $S^{(j)} \leftrightarrow \{0, 1, \ldots, n_j - 1\}$ such that $x_0^{(j)} \mapsto 0$. From now on, we will use natural number state indices and abstract state vectors interchangeably.

2.3.1 Stochastic automata networks as Markov chains

We associate a Markov chain $X(t)$ with a SAN as follows:

- The state space of the Markov chain is $S = \{0, 1, \ldots, n-1\}$, i.e. the reachable states RS of *SAN* according to the assumed bijection.
- The transition rate from $\mathbf{x}$ to $\mathbf{y}$ due to the event e is $\lambda(e) \cdot \prod_{j \in \operatorname{supp} e} \mu^{(j)}$, where $x^{(j)} \xrightarrow{e,\mu^{(j)}} y^{(j)}$. Thus, the infinitesimal generator matrix Q matrix of the $X(t)$ is formed by off-diagonal (Q_O) diagonal (Q_D) parts as

$$Q = Q_O + Q_D,$$

$$q_O[x, y] = \begin{cases} 0 & \text{if } x = y, \\ \displaystyle\sum_{e \in E, \mathbf{x}[e)\mathbf{y}} \lambda(e) \cdot \prod_{j=0}^{J-1} \mu_e^{(j)} & \text{if } x \neq y, \text{ where } x^{(j)} \xrightarrow{e,\mu_e^{(j)}} y^{(j)}, \end{cases}$$

$$Q_D = -\operatorname{diag}\{Q_O \mathbf{1}^{\mathrm{T}}\}.$$

- The initial distribution concentrates all the probability mass at $\mathbf{x}_0$,

$$\pi_0 = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \end{pmatrix},$$

that is, $\pi_0[x] = \delta_{0,x}$.

The generator matrix requires $O(n^2)$ memory if a two-dimensional dense array format is used.

Suppose that for each event e and source state $\mathbf{x}$, $\mathbf{x}\,[e\rangle\,\mathbf{y}$ holds only for a number of different target states $\mathbf{y}$ bounded from above by $k \in \mathbb{N}$. Therefore, each row of Q contains up to $k|E|+1$ nonzero elements including the diagonal element. This means Q requires $O(nk|E|)$ memory if a sparse format is chosen, which is preferable over dense arrays for larger models.

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 amount of time.

2.3.2 Kronecker generator matrices

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

$$Q = Q_O + Q_D, \quad Q_O = \bigoplus_{j=0}^{J-1} Q_L^{(j)} + \sum_{e\in E_S} \lambda(e) \bigotimes_{j=0}^{J-1} Q_e^{(j)}, \quad Q_D = -\operatorname{diag}\{Q_O \mathbf{1}^{\mathrm{T}}\}, \tag{2.6}$$

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

$$Q_L^{(j)} = \sum_{e\in E_L^{(j)}} \lambda(e)\, Q_e^{(j)}$$

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

$$Q_e^{(j)} \in \mathbb{R}^{n_j \times n_j}, \quad q_e^{(j)}[x^{(j)}, y^{(j)}] = \begin{cases} \mu & \text{if } x^{(j)} \xrightarrow{e,\mu} y^{(j)}, \\ 0 & \text{otherwise} \end{cases}$$

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

The matrices $Q_L^{(j)}$ and $Q_e^{(j)}$ 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 [18] 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 necessarily irreducible even if it is irreducible over RS.

In contrast, *actual Kronecker methods* [8, 18, 48] 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 [6, 15, 19].

2.3.3 Block Kronecker matrix composition

A *hierarchical decomposition* of the reachable state space expresses RS as

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

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 $A^{(j)}$, and $RS^{(j)}_x = \{0^{(j)}_x, 1^{(j)}_x, \ldots, (n_{j,x}-1)^{(j)}_x\}$ are the *local micro states* in the local macro state $\tilde{x}^{(j)}$. The product symbol denotes the composition of local states into a global state vector.

The decomposition of the state space into global macro states allows Q to be expressed as a block matrix, where each matrix block is expressed using Kronecker decomposition.

The matrices $Q^{(j)}_e[\tilde{x}^{(j)}, \tilde{x}^{(j)}]$ and $Q^{(j)}_L[\tilde{x}^{(j)}, \tilde{x}^{(j)}] \in \mathbb{R}^{n_{j,x} \times n_{n,y}}$ describe the effects of a single event $e \in E$ and the aggregated effects of local transitions on $A^{(j)}$ as its state changes from the local macro state $\tilde{x}^{(j)}$ to $\tilde{y}^{(j)}$, respectively. Formally,

$$q^{(j)}_e[\tilde{x}^{(j)}, \tilde{y}^{(j)}][a^{(j)}_x, b^{(j)}_y] = \begin{cases} \mu & \text{if } a^{(j)}_x \xrightarrow{e,\mu} b^{(j)}_y, \\ 0 & \text{otherwise,} \end{cases}$$
$$Q^{(j)}_L[\tilde{x}^{(j)}, \tilde{y}^{(j)}] = \sum_{e \in E^{(j)}_L} \lambda(e) Q^{(j)}_e[\tilde{x}^{(j)}, \tilde{y}^{(j)}].$$

In the case $j \notin \operatorname{supp} e$, we define $Q^{(j)}_e[\tilde{x}^{(j)}, \tilde{y}^{(j)}]$ 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

$$Q_O[\tilde{\mathbf{x}}, \tilde{\mathbf{y}}] = \begin{cases} \bigoplus_{j=0}^{J-1} Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] \\ \qquad + \sum_{e \in E_S} \lambda(e) \bigotimes_{j=0}^{J-1} Q_e^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] & \text{if } \tilde{\mathbf{x}} = \tilde{\mathbf{y}}, \\ I_{N_1 \times N_1} \otimes Q_L^{(h)}[\tilde{x}^{(h)}, \tilde{x}^{(h)}] \otimes I_{N_2 \times N_2} \\ \qquad + \sum_{e \in E_S} \lambda(e) \bigotimes_{j=0}^{J-1} Q_e^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] & \text{if } (\tilde{\mathbf{x}}, \tilde{\mathbf{y}}) \text{ slmst. at } h, \\ \sum_{e \in E_S} \lambda(e) \bigotimes_{j=0}^{J-1} Q_e^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] & \text{otherwise,} \end{cases} \tag{2.7}$$

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 $A^{(h)}$. In every other case, only synchronizing transitions may occur.

This expansion of block matrices is equivalent to eq. (2.6) on page 12 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. To further reduce storage requirements, identity matrices adjacent in Kronecker products may be merged if there is an $O(1)$ storage method for identity matrices.

Chapter 3

Overview

3.1 General stochastic analysis 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 [45] 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.

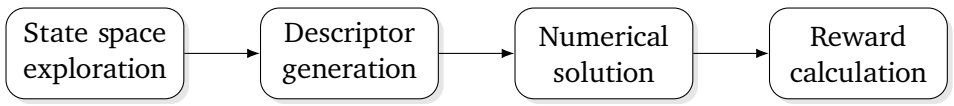


Figure 3.1 The general stochastic analysis workflow.

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 [2, 9], these analytic steps are called as subroutines to evaluate propositions. In the synthesis and optimization of stochastic models [22], 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 [25] 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 necessitates 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 performed 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 implementations are also possible, albeit they are missing from the current version of our framework.

3.2 Our workflow in PetriDotNet

"PetriDotNet is a framework for the editing, simulation and analysis of Petri nets. The framework is developed by the Fault Tolerant Systems Research Group at the Budapest University of Technology and Economics." [33]

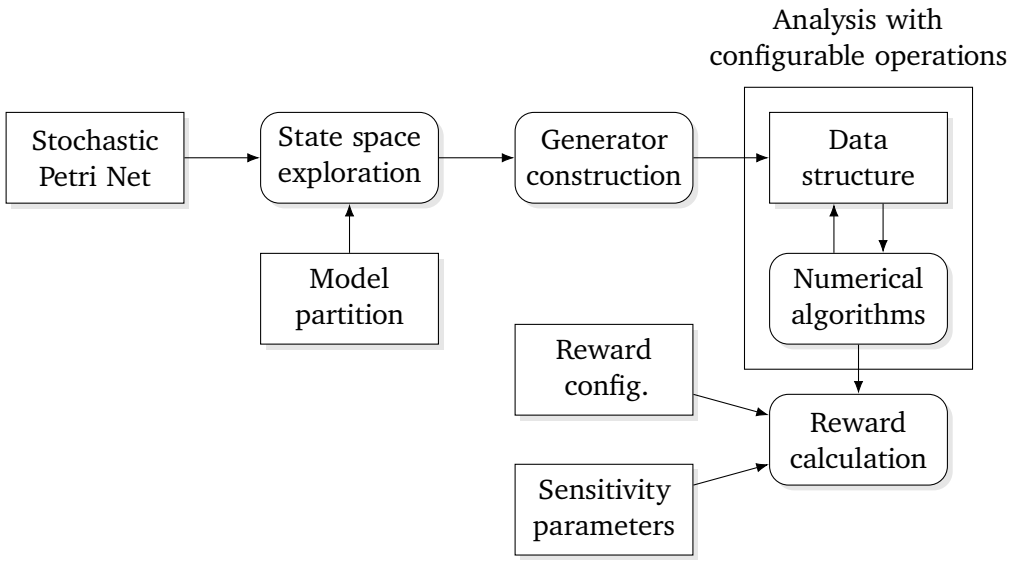


Figure 3.2 Configurable stochastic analysis workflow.

The implementation of the general stochastic analysis workflow in PetriDotNet is illustrated in Figure 3.2. The models are specified using the stochastic Petri net (SPN) formalism [52, 60], while engineering measures to be calculated are expressed as SPN performance measures. Both explicit and symbolic state space exploration and storage is supported, including symbolic hierarchical state space decomposition for block Kronecker generator matrices.

The workflow is fully *configurable*, which means that the modeler may combine the available algorithms for the analysis steps arbitrarily. In addition, implementations of the linear algebra operations performed by the algorithms may be replaced at runtime.

3.3 Architecture

Figure 3.3 shows the architecture of the configurable stochastic analysis module.

- The user interacts with the stochastic *analysis workflow* runner.

  The model, its parameters and its stochastic behavior as transition rates of timed transitions is specified and engineering measures of interest (e.g. performability, availability, reliability, dependability) are defined with SPN rewards. Afterwards, the analysis workflow can be initiated by selecting the analysis type (steady-state, sensitivity, transient, MTFF), the used algorithms and the engineering measures to compute. The workflow runner instantiates and executes the components which are required to complete the analysis and displays the results.

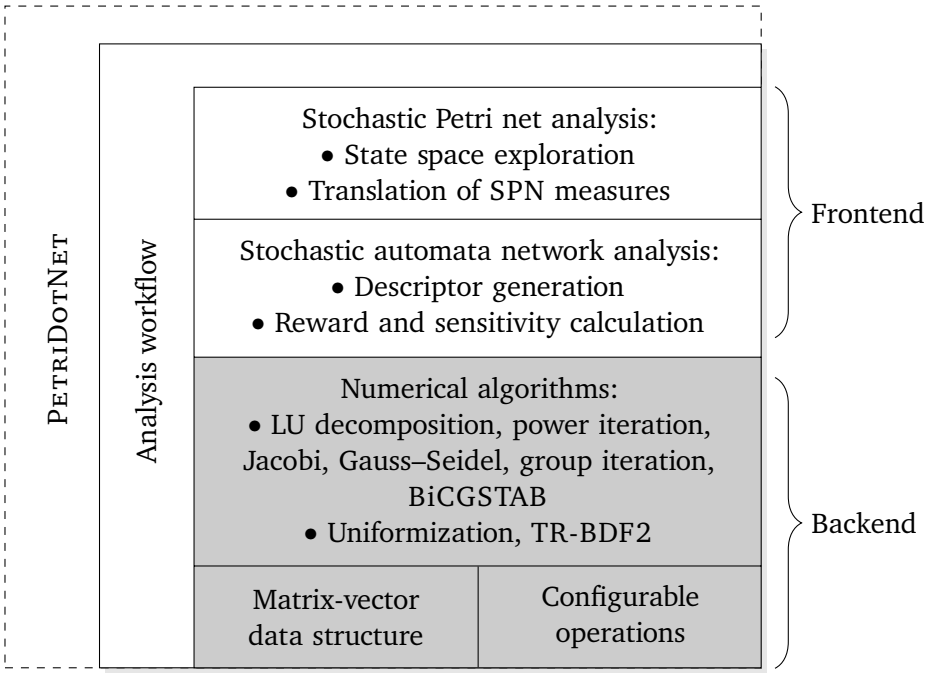


Figure 3.3 Layered architecture for configurable stochastic analysis.

Numerical analysis algorithms most suitable for the analyzed model and executing hardware may be selected by the user. Moreover, low-level linear algebra operations, for example, parallel or sequential algorithms for matrix products, may be also selected for every step in the workflow.

The stochastic analysis problem is translated into numerical problems by the "frontend" part of the analysis module:

- The *stochastic Petri net analysis* modules translate the stochastic behavior of Petri net into generic data structures. The partition of the model defines the stochastic automata of the SAN representation of the model. The algebraic expressions that specify transition rates and rewards are evaluated, thus lower level components only work with transition rates and their derivatives.

  Symbolic state space exploration is performed by the *saturation* algorithm [24], which is provided by the symbolic analysis component of PetriDotNet [29]. Petri nets with inhibitor arcs are supported, but transitions with priority (including non-timed transitions) cannot be used/

  Reward expressions that refer to subsets of the reachable state space defined by Computation Tree Logic (CTL) [44] are also evaluated by the symbolic analysis

component. Therefore, CTL rewards cannot be used with explicit state space representation algorithms.

- The *stochastic automata network analysis* module implements explicit and symbolic procedures for infinitesimal generator matrix composition and reward calculation. This component does not depend on the Petri net formalism and may be reused for different formalisms.

  The matrices Q and V_i, that is, the generator matrix and its partial derivatives may be stored as a dense or sparse array or a block Kronecker matrix using the object model defined by the matrix-vector data structure. Linear algebra operations during the generator composition, for example, calculation of the diagonal entries of Q, are performed by the operation framework supporting the data structure.

  Numerical solution algorithms, such as linear equation solvers and transient integrators are called to derive the steady state and transient distributions of the Markov chain and its sensitivities.

  The final task performed by the frontend is the calculation of the reward values, which uses both linear algebra operations and symbolic iteration over the results of the CTL evaluator.

The analysis "backend" serves as a library of matrix–vector data structured, linear algebra operations and numerical solution algorithms:

- *Numerical algorithms* implement solution finding for linear equations and Markovian transient initial value problems. The algorithms are implemented generically whenever possible, so that no assumptions are made about the structure of matrices unless necessary due to mathematical or performance reasons. This is achieved by the definition of a (non-orthogonal) set of operations on the matrix-vector data structure. The operations may be replaced at runtime for flexibility, for example, different implementations of operations may be used for different algorithms in the same workflow.

- The *matrix-vector data structure* provides an interface for storing various linear algebra objects.

  In addition to dense and sparse arrays, wrappers are provided to access parts of matrices and vectors and to build expression trees out of smaller matrices. Hence, matrices such as block Kronecker infinitesimal generators (eq. (2.7) on page 14) can be stored as a collection of small sparse matrices in a nested expression tree.

  While current the frontend only generates simple arrays and block Kronecker matrices, and descriptor format may be used as long as it can be expressed as expression trees.

The data structure only provides storage, any calls to linear algebra operations are delegated to the configurable operation context.

- The *configurable operation context* provides and dispatches the implementations of linear algebra operations, such as matrix-vector product or vector addition.

  Operations are specific to the data structure and may use multiple dispatch call semantics. For example, an operation can be defined that handles the multiplication of a block matrix and a constant vector, and stores the result in vector backed by a linear array. In addition to type information, the dispatch may use addition runtime properties, such as the length of a vector to select the appropriate implementation.

  The dispatch rules may be modified at runtime. For example, parallel execution may be replaced with sequential during the execution of algorithms that achieve parallelization through other means.

  Contrast operations with numerical algorithms, which are higher level procedures that solve a particular numerical problem on a wide range of data structures by delegation to a non-orthogonal set of specific operations.

The stochastic analysis backed, which was developed by the author, comprise the topic of present work. Figure 3.3 shows its three components shaded in gray.

Manipulations performed by the frontend components on the input Petri net models the generated descriptors are discussed in this thesis only briefly. We refer the interested reader to [49] for an overview of the whole PetriDotNet stochastic analysis component.

3.4 Current status

In this section we briefly summarize the results of the backend development effort.

3.4.1 Data structures

The data structure can represent infinitesimal generator matrices of continuous-time Markov chains with possible stochastic automata network structure as dense or sparse arrays, sums of Kronecker products and block Kronecker matrices.

Additional matrix decompositions may be created as algebraic expression trees using from the provided expression primitives of block matrices, linear combinations and Kronecker products. Moreover, the framework may be extended with further primitives simply by implementing some interfaces.

The linear algebra operations framework provides the most commonly used vector and matrix operations, including addition and scaling, scalar products and vector-matrix

Table 3.1 Linear equation solvers supported by our framework.

	see	memory usage	parallel impl.	uses inner solver	block matrix
LU decomposition	p. 44	very high	–	–	–
Power method	p. 46	moderate	✓	–	✓
Jacobi over-relaxation	p. 47	moderate	✓	–	✓
Gauss–Seidel over-relaxation	p. 47	very low	–	–	✓
Group Jacobi	p. 50	moderate	✓	✓	required
Group Gauss–Seidel	p. 50	low	–	✓	required
BiCGSTAB	p. 53	high	✓	–	✓
IDR(s)STAB(l)	p. 54	Work in progress			

Table 3.2 Transient solvers supported by our framework.

	see	instantenous distribution	accumulated distribution	uses inner solver	block matrix
Uniformization	p. 63	✓	✓	–	✓
TR-BDF2	p. 64	✓	not impl.	✓	not impl.

products. Matrix-matrix products are not provided due to the impossibility of fast and compact evaluation of such products with matrices of general decomposed forms.

All provided operations are listed in Table 4.1 on page 37 in Chapter 4, where the data structure and operations are discussed.

3.4.2 Numerical algorithms

Seven linear equation solver algorithms were implemented for steady-state, sensitivity and MTFF problems: LU decomposition, power iteration, Jacobi over-relaxation, Gauss–Seidel over-relaxation, group Jacobi, group Gauss–Seidel and BiCGSTAB. Group Jacobi and Gauss–Seidel require a block matrix, while the other algorithms may run on any matrix.

Special attention is paid to the root finding of singular systems with zero right vectors, i.e. the determination of the nullspace of a matrix for systems arising from Markovian steady-state problems. Nonsingular problems are solved in steady-state sensitivity analysis and mean-time-to-failure analysis.

The current research and development effort focuses on the integration of a solver based on IDR(s)STAB(l) [78], a Krylov subspace algorithm which generalizes the

BiCGSTAB algorithm. As the algorithm needs adaptation for singular matrices, it is currently not suitable for production use in Markovian analysis due to numerical breakdowns and instability.

Our modifications to IDR(s)STAB(l) are shown in Algorithms 5.10 to 5.12 on pages 57–59, which were found to improve convergence behavior in steady-state analysis problems. However, the stability is still lacking, as we observed in Section 6.3 on page 79.

Two solution algorithms, uniformization and TR-BDF2 are available for transient analysis. Accumulated rewards can be calculated by uniformization only, while TR-BDF2 provides robustness for otherwise difficult to handle stiff Markov chains.

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 further discussed in Chapter 5. Linear equations solvers for steady-state CTMC analysis are shown in Table 3.1, while transient solvers are shown in Table 3.2.

Chapter 4

Configurable data structure and operations

In this chapter, we present the linear algebra library that was developed as a foundation for configurable stochastic analysis.

The library is composed of a data structure and its related operations. The *data structure* provides abstraction for the numerical solution algorithms over the used matrix and vectors storage formats. Matrices stored as dense or sparse arrays, and even complex expression involving sums and Kronecker products that arise from matrix decompositions can be handled in a general way.

While direct read write access to elements is supported for most matrices and vectors, the majority of manipulations, such as vector–matrix products or vector additions, structure are performed as *operations*. Instead of being implemented as methods of the data structure classes, operations are decoupled into separate entities. This allows operation execution with multiple dispatch, selecting optimized implementations according to dynamic types of all operation arguments and other runtime properties, e.g. the number of elements in the vector.

Another advantage of the decoupled operations framework is runtime configurability. The dispatch logic may be replaced between the execution of algorithms in the stochastic analysis workflow, therefore low level linear algebra operations may customized to suit the algorithm and the matrix decomposition in use, as well as the hardware. For example, parallel and sequential execution may be switched as neccessary.

Existing linear algebra and matrix libraries, such as [11, 32, 41, 53, 74], usually have unsatisfactory support for operations required in stochastic analysis algorithms with decomposed matrices, such as multiplications with Kronecker and block Kronecker matrices. Therefore, we have decided to develop out linear algebra framework from scratch in C#.NET specifically for stochastic algorithms as a basis of our stochastic analysis framework.

4.1 Data structure

The data structure library contains matrix and vector classes for stochastic analysis.

Client code interacts with the data structure through interfaces only, no classes are exposed on the public API. The main interfaces are `IVector` and `IMatrix` for vectors and matrices, respectively. The instances are created through an exposed static factory.

The interfaces are generic in the element type. For example `IVector<double>` and `IMatrix<double>` are used to work with double-precision floating point arithmetic. Due to language limitations, some classes must be implemented without genericity. In these cases, only **`double`** is currently supported, although re-implementation for single-precision floating point or other numeric types is trivial. The static factory handles selection of the appropriate non-generic type to instantiate if generic behavior is impossible.

There also exist *block* versions of these interfaces, IBlockVector and IBlockMatrix. A block object is conceptually a container of objects with scalar elements. For example, if $\mathbf{v} \in \mathbb{R}^{n_0+n_1+\cdots+n_{k-1}}$ is a block vectors with k blocks, $\mathbf{v}[i] \in \mathbb{R}^{n_i}$ $(0 \leq i < k)$ is a vector of real numbers with n_i elements, while $v[i][j]$ is the jth element of the ith block of $\mathbf{v}$. However, block interfaces do not extend from `IVector<IVector<T>>` and `IMatrix<IMatrix<T>>`, but a facility separate from ordinary indexing is provided for block access. This allows passing `IBlockVector<double>` and `IBlockMatrix<double>` objects to procedures consuming ordinary `IVector<double>` and `IMatrix<double>`.

4.1.1 Partials, splitting and composition

Manipulations of subsequences of vector and matrix elements, as well as conversion between flat and block object are performed by partial object wrappers.

Definition 4.1 A *partial vector* $\mathbf{v}[s{:}t{:}m]$ of a vector $\mathbf{v} \in \mathbb{R}^n$ is

$$\mathbf{v}[s{:}t{:}m] \in \mathbb{R}^m : v[s{:}t{:}m][i] = v[s + t \cdot i] \text{ for } i = 0, 1, \ldots, m,$$

where $0 \leq s \leq n, 1 \leq t, s + t(m-1) \leq n$.

The index s is the start of partial, t is the stride and m is the partial length. Matrix partials $A[s_1{:}t_1{:}m_1, s_2{:}t_2{:}m_2] \in \mathbb{R}^{m_1 \times m_2}$ are defined analogously.

The method `GetPartial` forms partial matrices and vectors. The returned object is always a wrapper which passes through and read and write indices to the original object after index manipulation. However, partial manipulation of large vectors, which was found to be a performance bottleneck upon profiling, is implemented with pointer arithmetic instead.

The `GetPartial` method itself is also passed through. This means forming a partial of partial ($\mathbf{v}[s_1{:}t_1{:}m_1][s_2{:}t_2{:}m_2]$) does not result in a chain of wrappers being created, but only a single wrapper object is places around the original after the necessary index manipulations.

Block vectors and matrices may be be formed by splitting flat objects into blocks with partials, or by composition from unrelated objects.

Definition 4.2 A *split* of a vector $\mathbf{v} \in \mathbb{R}^n$ at $(n_0, n_1, \ldots, n_{k-1})$ is a block vector

$$\mathbf{v}_S \in \mathbb{R}^{n_0+n_1+\cdots n_{k-1}} : \mathbf{v}_S[i] = \mathbf{v}[N_i{:}1{:}n_i], \quad N_i = \sum_{j=0}^{i-1} n_j, \tag{4.1}$$

where $N_0 = 0$ and $n = N_{k+1} = n_0 + n_1 + \cdots n_{k-1}$.

Split matrices are defined analogously.

Definition 4.3 If $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{k-1}$ are real vectors of length $n_0, n_1, \ldots, n_{k-1}$, respectively, their *composition* $\mathbf{v}_C \in \mathbb{R}^{n_0+n_1+\cdots+n_{k-1}}$ is block vector $\mathbf{v}_C[i] = \mathbf{v}_i$.

Definition 4.4 If $A_{0,0}, A_{0,1}, \ldots, A_{0,l-1}, A_{1,0}, \ldots A_{k-1,l-1}$ are matrices such that $A_{i,j} \in \mathbb{R}^{n_i, m_j}$, their *composition* $A_C \in \mathbb{R}^{(n_0+n_1+\cdots+n_{k-1})\times(m_0+m_1+\cdots m_{l-1})}$ is block matrix $A_C[i,j] = A_{i,j}$.

The `Split` method builds block vectors and matrices from flat objects for blockwise access. Objects formed by `Split` can be split again arbitrarily, where the split command is forwarded to the original flat object.

In contrast, `Split` can only be applied to a composite object if it does not result in the creation of new partials. That is, a composite vector $\mathbf{v} \in \mathbb{R}^{n_0+n_1+\cdots+n_{k-1}}$ may only be split at $(m_0, m_1, \ldots, m_{l-1})$ if $k = l$ and $n_i = m_i$ for all $0, 1, \ldots, k-1$. Because composite objects are usually very large and are used in performance critical parts of algorithms, we decided to throw an exception instead of splitting even though arbitrary splitting of composite vectors and matrices would have been implemented easily.

4.1.2 Vectors

Vector data structures are used to store probability distributions of Markovian models, as well as intermediate results of numerical algorithms.

The class hiearchy of vectors in our library is shown in Figure 4.1.

The abstract base classes `AbstractVector`, `AbstractBlockVector` are at the root of the inheritance hierarchies. The data structure may be extended by inheriting from these classes, or by implementing the publicly exposed interfaces directly.

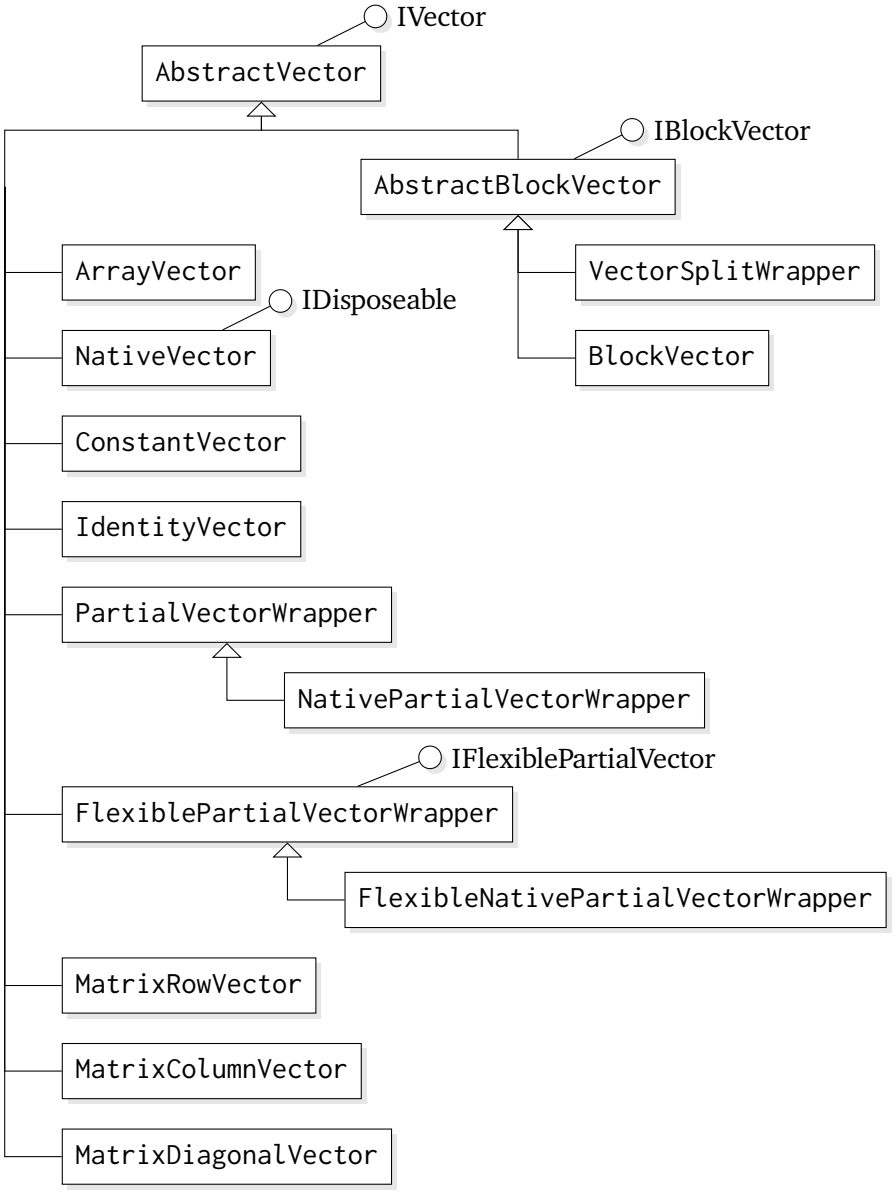


Figure 4.1 Inheritance hierarchy of vectors.

Vector data types are available for the storage of general vectors, as well as for some special cases.

ArrayVector The basic vector datatype provided is `ArrayVector`, which stores vector elements in a Common Language Runtime (CLR) array. This class is completely generic, i.e. any CLR value or reference type may be used as an element.

The Microsoft .NET implementation of the CLR allows arrays of size up to 2 GiB even on 60-bit platforms. While on .NET 4.5, this limitation may be lifted with the `gcAllowVeryLargeObjects` configuration directive[1], this setting is cumbersome to use. Therefore, no vectors larger than 2 GiB should be stored as array vectors.

NativeVector To work around the 2 GiB memory limitation on CLR arrays, we implemented `NativeVector` which stores vector elements on the unmanaged heap. We also found unmanaged allocation reduce the pressure on the garbage collector, therefore provide the benefit of faster allocations.

Native vectors utilize the `unsafe` facilities[2] provided by the C# language, including the access to memory through pointers and direct memory management through `AllocHGlobal` and `FreeHGlobal`. Therefore, the linear algebra library must be compiled with unsafe language features enabled. As an alternative, `NativeVector` may be disabled with conditional compilation directive and replaced by a wrapper around `ArrayVector`, forgoing the benefits of unmanaged allocation.

Due to language limitations, `NativeVector` must be implemented for any primitive type desired to be used as vector elements. Currently, only **`double`** is supported.

The use of unmanaged memory requires manual deallocation to avoid memory leaks. Because the C# language does not provide deterministic destructors, the `IDisposeable` pattern[3] must be used.

As an alternative means of memory management, an interface `IBufferProvider` may be used to allocate and track multiple vectors. A `IBufferProvider` itself implements `IDisposeable`, a single C# `using` block may free several vectors in the same scope, easing the burden of manual disposal. This approach is illustrated in Listing 4.1.

ConstantVector A constant vector is a vector with equal elements.

Two important special vector may be realized as `ConstantVector` instances in stochastic analysis, the vector of all zeroes $\mathbf{0}$, and the vector of all ones $\mathbf{1}$,

$$\texttt{Vectors.Constant<double>}(n,\ 0) = \mathbf{0} \in \mathbb{R}^n,$$
$$\texttt{Vectors.Constant<double>}(n,\ 1) = \mathbf{1} \in \mathbb{R}^n.$$

[1] `https://msdn.microsoft.com/en-us/library/hh285054(v=vs.110).aspx`
[2] `https://msdn.microsoft.com/en-us/library/chfa2zb8.aspx`
[3] `https://msdn.microsoft.com/en-us/library/system.idisposable(v=vs.110).aspx`

Listing 4.1 Manual memory management for `NativeVector`.

```
// Create and dispose a NativeVector of length 100.
using (var vector = Vectors.NewDisposeableVector<double>(100))
{
    vector[0] = 1.0;
}

// Dispose using IBufferProvider.
var factory = new DisposingBufferProviderFactory();
using (var bufferProvider = factory.Make())
{
    var v1 = bufferProvider.GetVector<double>(100);
    var v2 = bufferProvider.GetVector<double>(100);
}
// Both v1 and v2 are disposed here.
```

Because constant vectors require only $O(1)$ storage space instead of $O(n)$, this is an important optimization in equations involving $\mathbf{0}$, $\mathbf{1}$ and its scalar multiples.

IdentityVector An identity vector is vector with all but one zero elements and a single 1 element. Formally,

$$\texttt{Vectors.Identity<double>}(n,\ i) = \mathbf{e}_i \in \mathbb{R}^n, \quad e_i[j] = \delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

`IdentityVector` is an $O(1)$ space optimization for storing special vectors, similar to `ConstantVector`.

PartialVectorWrapper and FlexiblePartialVectorWrapper Taking a partial vector of a vector results in the creation of `PartialVectorWrapper` object, which passes through read and write actions to the underlying vector after the necessary index manipulations. Hence a new object is allocated at every call to `GetPartial`.

Long delegation chains are eliminated by *collapsing* partial vectors. When the method `GetPartial` is invoked on `PartialVectorWrapper`, it performs index manipulations and passes delegates to the `GetPartial` method of the original vector. Thus, further partials do not have reference to the partial vector they were created from, but only to the underlying vector.

`FlexiblePartialVectorWrapper` alleviates allocation costs in inner loops by providing a partial vector whose indices can be changed after construction. For example, if

Listing 4.2 Ambigous use of `FlexiblePartialVectorWrapper`.

```
1  var vector = Vectors.NewArray<double>(100);

3  var part = vector.GetPartial(5, 2, 20);
4  var subPart = part.GetPartial(2, 1, 5);
5  // Unambigous, subPart = vector[5:2:20][2:1:5].

7  var flexible = vector.GetFlexiblepartial(5, 2, 20);
8  var subFlexible = vector.GetPartial(2, 1, 5);
9  flexible.SetPartial(6, 2, 20);
10 // Ambigous, is subFlexible = vector[5:2:20][2:1:5] or vector[6:2:20][2:1:5]?
```

a flexible partial vector $\mathbf{v}[s{:}t{:}m]$ is available, it can be changed to $\mathbf{v}[s'{:}t'{:}m']$ without allocating a new instance whenever the need arises. The functionality is exposed to consumers through an interface.

Flexible partials cannot have their partials taken, because the collapse of the delegation chain makes the propagation of index changes impossible. This problem is illustrated in Listing 4.2.

Another set of partial wrappers handle partials of `NativeVector` instances. In these cases, a (*base pointer*, *stride*, *length*) triple amy be queried for use in low-level operations. Hence indexing logic may be skipped in favor of direct access.

To create a unified interface, the same triple may be queried from `NativeVector` instances themselves, where *base pointer* is the pointer to the allocated buffer, $\mathit{stride} = 1$ and *length* is the length of the vector itself.

Matrix vector wrappers To facilitate common manipulations of matrices, our library provides wrappers to for read and write access of parts of matrices as vectors.

`MatrixRowVector` and `MatrixColumnVector` accesses a row or a column of matrix, respectively. If $A \in \mathbb{R}^{n \times m}$,

$$\begin{aligned} A.\texttt{GetRow}(i) = \mathbf{r} \in \mathbb{R}^m, \quad r[j] = a[i,j], \\ A.\texttt{GetColumn}(j) = \mathbf{c} \in \mathbb{R}^n, \quad c[i] = a[i,j] \end{aligned}$$

for $0 \leq i < n$, $0 \leq j < m$.

If $n = m$, i.e. A is square, `MatrixDiagonalVector` may provide access to the diagonal of the matrix,

$$A.\texttt{GetDiagonal}() = \mathbf{d} \in \mathbb{R}^n = \mathbb{R}^m, \quad d[i] = a[i,j].$$

Block vectors `VectorSplitWrapper` reifies vector splitting according to eq. (4.1) on page 25. The split vector is backed by a composition of partial vectors, thus it acts as a composite vector. However, when the split wrapper is used as an instance of `IVector`, commands, including re-splitting, are delegated to the underlying vector instead.

Composition of vectors according to Definition 4.3 on page 25 is represented by `BlockVector`. The constructor of `BlockVector` is passed a sequence of vectors which will constitute the block vector. Because `BlockVector` implements `IVector`, the composite vector may be used as a normal vector, however, splitting is limited to avoid performance penalties associated

4.1.3 Matrices

The class hiearchy of vectors in our library is shown in Figure 4.2. The abstract base classes `AbstractVector`, `AbstractBlockVector` are at the root of the inheritance hierarchies.

ArrayMatrix For smaller dense arrays with items of any value or reference type, `ArrayMatrix` allows storage in two-dimensional CLR array.

The matrix may not be larger than 2 GiB, however, this is not a serious limitation in practice, because processing large dense arrays could take extreme amounts of time.

SparseMatrix and NativeSparseMatrix 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 (Figure 4.3), in order to effectively perform multiplications by vectors from left.

While other sparse matrix formats, such as sliced LAPACK are more amenable to parallel and SIMD processing Kreutzer et al. [50], 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.

The class `SparseMatrix` implements CCS sparse matrices backed by CLR arrays.

Due to state space explosion, extremely large sparse matrices may be needed when block Kronecker decomposition is not use use. In our experiments in Chapter 6, sparse matrices up to 20 GiB were tested. Therefore, sparse matrices backed by unmanaged allocations were also implemented in the class `NativeSparseMatrix`.

`NativeSparseMatrix` is used for all sparse matrices, including Kronecker factor matrices in block Kronecker decompositions. Memory may be freed by the `IDisposeable` pattern or `IBufferProvider` (see Listing 4.1 on page 28).

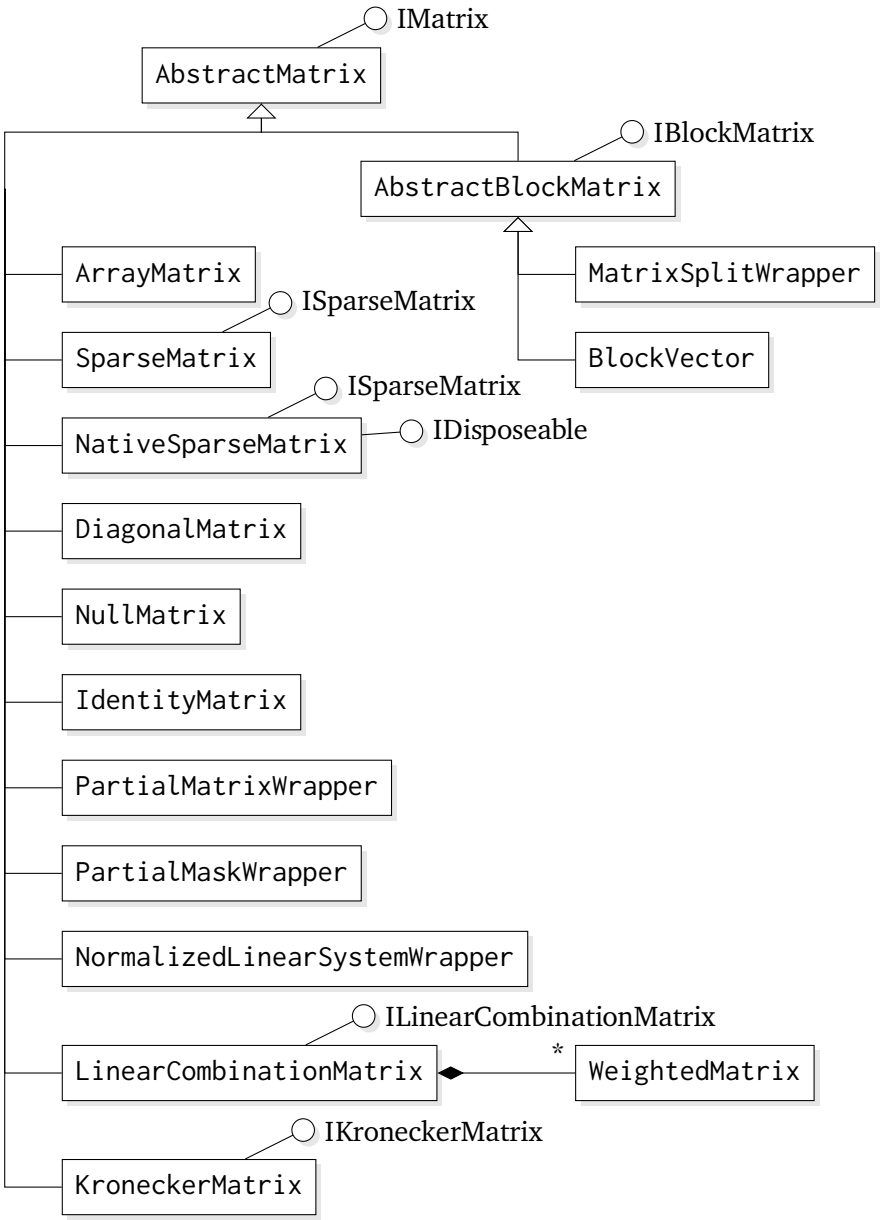


Figure 4.2 Inheritance hierarchy of matrices.

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

Figure 4.3 Compressed Column Storage of a matrix.

DiagonalMatrix For $O(n)$ storage of matrices that that have nonzero elements only along their diagonal, `DiagonalMatrix` provides a wrapper around any `IVector` containing the diagonal elements.

Let $\mathbf{v} \in \mathbb{R}^n$. Then we have

$$\texttt{Matrices.Diagonal}(\mathbf{v}) = \operatorname{diag}\{\mathbf{v}\} = D \in \mathbb{R}^{n\times n}, \quad d[i,j] = \begin{cases} v[i] & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

NullMatrix and IdentityMatrix Two special cases were implemented with $O(1)$ storage, zero matrices and identity matrices, i.e.

$$\texttt{Matrices.Null<}\textbf{double}\texttt{>}(n,\ m) = A \in \mathbb{R}^{n\times m}, \quad a[i,j] = 0,$$

$$\texttt{Matrices.Identity<}\textbf{double}\texttt{>}(n) = B \in \mathbb{R}^{n\times n}, \quad b[i,j] = \delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

PartialMatrixWrapper Partials of matrices written as $A[s_1{:}t_1{:}m_1, s_2{:}t_2{:}m_2]$ are represented as instances of `PartialMatrixWrapper`. The wrapper passes through any access after the necessary index manipulations to the underlying matrix.

There is no support for flexible partial matrices, that is, unlike vectors, the partial indexing of a matrix cannot be updated without creating a new instance of `IPartialMatrixWrapper`.

PartialMaskWrapper The `PartialMaskWrapper` class may be used to access the strictly upper and strictly lower triangular and diagonal parts of a matrix. In addition, the mask flags may be combined arbitrarily to yield upper and lower triangular parts of matrices including the diagonal and also the off-diagonal part.

NormalizedLinearSystemWrapper The normalized linear system wrapper replaces the last column of a matrix with the vector of all ones $\mathbf{1}$. If $A \in \mathbb{R}^{n\times m}$,

$$A\texttt{.ToNormalizedLinearSystem()} = \widehat{A} \in \mathbb{R}^{n\times m}, \quad \hat{a}[i,j] = \begin{cases} 1 & \text{if } j = m-1, \\ a[i,j] & \text{if } j \neq m-1. \end{cases}$$

If A is an $n \times n$ matrix of rank $n-1$ (i.e. its nullity is 1) the system of linear equations

$$\mathbf{x}A = \mathbf{1}, \quad \mathbf{x}\mathbf{1}^{\mathrm{T}} = 1$$

may be replaced with

$$\mathbf{x}\widehat{A} = \mathbf{e}_{n-1},$$

because A contains a redundant column due to its rank deficiency. The vector $\mathbf{e}_{n-1}$ may be realized as an `IdentityVector`.

LinearCombinationMatrix Linear combinations of matrices may be stored as an instance of `LinearCombinationsMatrix` that contains a sequence of `WeightedMatrix` instances, which are pairs of `IMatrix` objects a double-precision floating point scaling factors. In our current implementation, only linear combinations of `IMatrix<`**`double`**`>` are supported.

If $A_0, A_1, \ldots, A_{k-1} \in \mathbb{R}^{n\times m}$ and $w_0, w_1, \ldots, w_{k-1} \in \mathbb{R}$,

$$\texttt{Matrices.LinearCombination}(A_0\texttt{.WithWeight}(w_0),\ A_1\texttt{.WithWeight}(w_1),\ \ldots,\ A_{k-1}\texttt{.WithWeight}(w_{k-1})) = w_0A_0 + w_1A_1 + \cdots + w_{k-1}A_{k-1} \in \mathbb{R}^{n\times m}.$$

As an optimization, a term of type `DiagonalMatrix` in the linear combination may be designated as the *diagonal* if no other term contains diagonal elements. In this case, methods to access the diagonal of the linear combination matrix will return the designated term instead of `MatrixDiagonalVector` and `PartialMaskWrapper` wrapper objects.

KroneckerMatrix `KroneckerMatrix` allows the representations of Kronecker products of matrices as expressions similar to `LinearCombinationMatrix` for linear combinations. More concretely,

$$\texttt{Matrices.KroneckerProduct}(A_0,\ A_1,\ \ldots,\ A_{J-1}) = A_0 \otimes A_1 \otimes \cdots \otimes A_{J-1},$$

where $A_0, A_1, \ldots, A_{J-1}$ are matrices of possibly different sizes.

As the Kronecker product is not explicitly computed, very large matrices may be expressed with this method in relatively little space.

Block matrices The classes `MatrixSplitWrapper` and `BlockMatrix` may be used for blockwise storage and access of matrices in ways similar to `VectorSplitWrapper` and `BlockVector`.

Diagonals and triangular parts of block matrices in $\mathbb{R}^{(n_0+n_1+\cdots+n_{k-1})\times(m_0+m_1+\cdots+m_{l-1})}$ may only be requested if $k = l$ and $n_i = m_i$ for all $i = 0, 1, \ldots, k-1$, i.e. when the rows

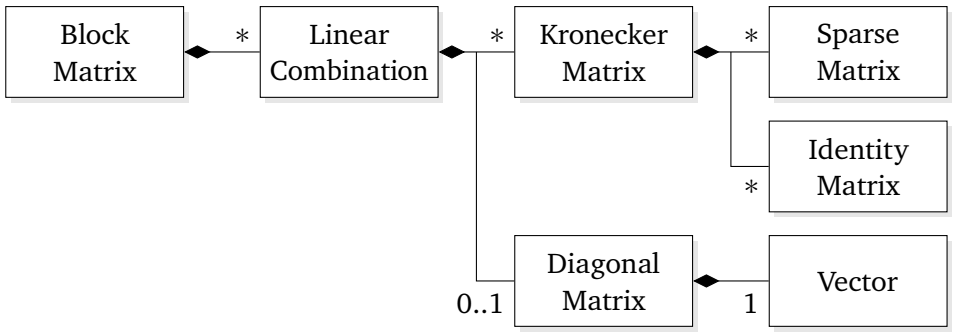


Figure 4.4 Data structure for block Kronecker matrices.

and columns of the matrix have the same blocking partition. This always occurs if the matrices describe transitions between states of a Markov chain, therefore, it poses no practical limitation.

4.1.4 Expression trees

Decomposed Kronecker and block Kronecker matrices are stored as algebraic expression trees as shown in Figure 4.4. Expression may contain Kronecker products (`KroneckerMatrix`), linear combinations (`LinearCombinationMatrix`) and block compositions (`BlockMatrix`).

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 operational primitives is also straightforward. The data structure forms a flexible basis for the development of stochastic analysis algorithms with decomposed matrix representations.

4.2 Operations

The operation framework achieves decoupling between the data structure and operations by reifying low-level linear algebra manipulations as *operation* objects.

In contrast to numerical solution algorithms, these operations are specific to their argument types. For example, the multiplication of a `NativeVector` with a matrix stored as a Kronecker product might be served by a different operation object that the multiplication by a `SparseMatrix`. In addition, dispatch logic may take other runtime properties into account, such as selecting a parallel implementation for multiplication only if the vector is long enough.

Listing 4.3 DSL for operation declarations (excerpt).

```
1  public enum OperationName
2  {
3      // Operation with base, operand, target and scale factor.
4      [OperationAlias("Add")]
5      [OperationArguments(typeof(IVector<>), typeof(IVector<>),
           typeof(IVector<>), ExtraOperationArguments.ScaleFactor)]
6      VectorAdd,

8      // Operation with only base and extra argument.
9      [OperationAlias("Clear")]
10     [OperationArguments(typeof(IVector<>), Extra =
           ExtraOperationArguments.Length)]
11     VectorClearFirstN,

13     // Operation that returns a value of the vector element type.
14     [OperationAlias("Sum")]
15     [OperationArguments(typeof(IVector<>), Return =
           typeof(TheGenericArgument))]
16     VectorSumElements,
17 }
```

4.2.1 Operation declarations

Operations have a *base* (B) argument, and optionally one or both of *operand* (O) and *target* (T) arguments. Syntactically, operation calls are represented as methods of the base object.

The target of the operation is the vector or matrix that gets modified by the operation. *In-place* execution of operations refers to the usage when the base and the target are equal. It is also possible that the operations has no other argument that its base, for example, the clearing of a vector by filling it with zeroes uses only a base. Then the base of the operation will be modified, similar to in-place use.

Extra (E) operation arguments may include a **`double`** scaling factor λ and various indices for operations manipulating parts of vectors and matrices. Operations that return a value are also possible.

Operations are specified using an Domain Specific Language (DSL) embedded into the C# programming language. An excerpt of the operation declarations is shown in Listing 4.3.

The operation declarations are processed with a Microsoft Text Template Transforma-

Listing 4.4 An example interface generated by the T4 template.

```
1 public interface IVectorAddOperation<T>
2 {
3     void Invoke(PDN.Analysis.Common.Math.Vector.IVector<T> @base,
          PDN.Analysis.Common.Math.Vector.IVector<T> operand,
          PDN.Analysis.Common.Math.Vector.IVector<T> target, double
          scaleFactor);
4 }
```

tion Toolkit (T4) template [58]. The template generates interfaces to be implemented by the operation classes (Listing 4.4). In addition, a dispatch logic stub is created in the `OperationContext`, which is the class that dispatches operation invocations.

The declared operations are summarised in Table 4.1.

Some operations are special cases of others, for example, ElementwiseMultiply may be implemented in terms of Clear and AccumulateElementwiseMultiply. However, a non-orthogonal set operations was defined to handle special cases such as in-place execution. In addition, the non-orthogonality allows more easier overloading in specific bottleneck scenarios identified by profiling.

4.2.2 Binding of operations to the data structure

Operations are declared on the interface of the base object as methods. Therefore, the call to an operation is syntactically equivalent to a method call. This gives the opportunity to use a friendlier naming and parameter order on the interfaces than the strict base-operand-target conventions used by the operation declarations.

The interface is augmented with a *contract class* which describes the pre- and postconditions of the operation. The Microsoft Code Contracts [56] runtime verification engine inserts assertions into the output of the compiler if the library is compiled in Debug mode. However, no assertions are inserted in Release mode, thus performance penalties are averted in production.

The base classes `AbstractVector` and `AbstractMatrix` contain a reference to their `OperationContext`. Operations are executed by delegation to the context, which contains the dispatch logic, partly generated by the T4 template from the operation declaration DSL, partly configured at runtime.

The contract and delegation process is illustrated in Listing 4.5.

Another domain specific language describe the dispatch logic of operations, which is referred to as the *operation configuration*. The calls are dispatched to objects imple-

Table 4.1 Linear algebra operations supported by our framework with their base (B), operand (O), target (T) and extra (E) arguments. Operations marked with a star (*) are syntactic sugars implemented in terms of other operations.

Category and operation	B O T E	Description
Vector		
Add	$\mathbf{b}$ $\mathbf{o}$ $\mathbf{t}$ λ	$\mathbf{t} \leftarrow \mathbf{b} + \lambda\mathbf{o}$
Scale	$\mathbf{b}$ — $\mathbf{t}$ λ	$\mathbf{t} \leftarrow \lambda\mathbf{b}$
Accumulate	$\mathbf{b}$ — $\mathbf{t}$ λ	$\mathbf{t} \leftarrow \mathbf{t} + \lambda\mathbf{b}$
Set*	$\mathbf{b}$ $\mathbf{o}$ — —	$\mathbf{b} \leftarrow \mathbf{o}$
ElementwiseMultiply	$\mathbf{b}$ $\mathbf{o}$ $\mathbf{t}$ —	$t[i] \leftarrow b[i] \cdot o[i]$
AccumulateElementwiseMultiply	$\mathbf{b}$ $\mathbf{o}$ $\mathbf{t}$ λ	$t[i] \leftarrow t[i] + \lambda \cdot b[i] \cdot o[i]$
Clear	$\mathbf{b}$ — — —	$\mathbf{b} \leftarrow \mathbf{0}$
ClearFirstN	$\mathbf{b}$ — — n	$\mathbf{b}[0{:}1{:}n] \leftarrow \mathbf{0}$
ScalarProduct	$\mathbf{b}$ $\mathbf{o}$ — —	return $\mathbf{b} \cdot \mathbf{o}$
Sum	$\mathbf{b}$ — — —	return $\sum_{i=0}^{n} b[i] = \mathbf{b} \cdot \mathbf{1}$
L1Norm	$\mathbf{b}$ — — —	return $\sum_{i=0}^{n} \lvert b[i]\rvert = \lVert\mathbf{b}\rVert_1$
L2Norm*	$\mathbf{b}$ — — —	return $\sqrt{\mathbf{b} \cdot \mathbf{b}} = \lVert\mathbf{b}\rVert_2$
Matrix		
Add	B O T λ	$T \leftarrow B + \lambda O$
Scale	B — T λ	$T \leftarrow \lambda B$
Accumulate	B — T λ	$T \leftarrow T + \lambda B$
Set*	B O — —	$B \leftarrow O$
Clear	B — — —	$B \leftarrow$ the zero matrix
VectorMatrix		
MultiplyFromLeft	B $\mathbf{o}$ $\mathbf{t}$ —	$\mathbf{t} \leftarrow \mathbf{o}B$
AccumulateMultiplyFromLeft	B $\mathbf{o}$ $\mathbf{t}$ λ	$\mathbf{t} \leftarrow \mathbf{t} + \lambda\mathbf{o}B$
MultiplyFromRight	B $\mathbf{o}$ $\mathbf{t}$ —	$\mathbf{t} \leftarrow B\mathbf{o}$
AccumulateMultiplyFromRight	B $\mathbf{o}$ $\mathbf{t}$ λ	$\mathbf{t} \leftarrow \mathbf{t} + \lambda B\mathbf{o}$
ScalarProductWithColumn	B $\mathbf{o}$ — j	return $\mathbf{b}[\cdot, j] \cdot \mathbf{o}$

Listing 4.5 Delegation of operations to the context (excerpt).

```
1  [ContractClass(typeof(VectorContract<>))]
2  public interface IVector<T> : IEnumerable<T>
3  {
4      // Method declaration on the interface with friendlier naming.
5      IVector<T> Add(IVector<T> toAdd, IVector<T> resultTarget,
           double scaleFactor = 1);
6  }

8  [ContractClassFor(typeof(IVector<>))]
9  abstract class VectorContract<T> : IVector<T>
10 {
11     // Declaration of preconditions.
12     IVector<T> IVector<T>.Add(IVector<T> toAdd, IVector<T>
           resultTarget, double scaleFactor)
13     {
14         Contract.Requires(toAdd != null);
15         Contract.Requires(resultTarget != null);
16         Contract.Requires(toAdd.Length ==
               ((IVector<T>)this).Length);
17         Contract.Requires(resultTarget.Length >=
               ((IVector<T>)this).Length);
18         throw new NotImplementedException();
19     }
20 }

22 abstract class AbstractVector<T> : IVector<T>
23 {
24     OperationContext OperationContext { get; set; }

26     // Delegation to the OperationContext.
27     public virtual IVector<T> Add(IVector<T> toAdd, IVector<T>
           resultTarget, double scaleFactor = 1)
28     {
29         OperationContext.Add(this, toAdd, resultTarget,
               scaleFactor);
30         return resultTarget;
31     }
32 }
```

Algorithm 4.1 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: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^{m_0+m_1+\cdots+m_{l-1}}$

```
1 allocate c ∈ ℝ^(m_0+m_1+⋯+m_{l−1})
2 parallel for j ← 0 to l − 1 do
3 |   c[j] ← 0                                          ▷ VectorClear
4 |   for i ← 0 to k − 1 do
5 |   |   c[j] ← c[j] + b[i]A[i, j]      ▷ VectorMatrixAccumulateMultiplyFromLeft
```

menting the interfaces generated from the operations declarations, such as the one shown in Listing 4.4.

The dispatch logic is compiled into .NET IL bytecode for fast execution. The dispatch logic may be changed at runtime between executions of higher level algorithms.

Current, two operation configurations are exposed readily on the public interface of the library.

- The *Parallel* operation configuration uses the thread pool to utilize multiple CPU cores.
- The *Sequential* operation configuration does not use multiple thread, therefore it is suitable for use with algorithms that handle multithreaded execution via other means.

Additional operations configurations may be developed with the public API of the library. Thus, the characteristics of the model and the executing hardware may be considered on the linear algebra operation level in addition to the numerical algorithm level in advanced stochastic analysis scenarios.

The flexible dispatch logic allows the identification of calculation hotspots via profiling, such that a specific operation implementation may be created and used to improve performance. If the specific implementation degrades performance of some algorithms, it can be switched off by replacing the operation configuration.

4.2.3 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.

Algorithm 4.2 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: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^m$

$\mathbf{c} \leftarrow \mathbf{0}$ ▷ VectorClear
for $h \leftarrow 0$ **to** $k-1$ **do**
  $\mathbf{c} \leftarrow \nu_h \cdot \mathbf{b}A_h$ ▷ VectorMatrixAccumulateMultiplyFromLeft
return $\mathbf{c}$

Algorithm 4.3 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
$\mathbf{x}[0{:}1{:}n] \leftarrow \mathbf{b}, i_{\text{left}} \leftarrow 1, i_{\text{right}} \leftarrow \prod_{h=1}^{k-1} n_h$ ▷ VectorSet
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**
        ▷ VectorMatrixMultiplyFromLeft
        $\mathbf{x}'[j_{\text{base}}{:}m_h{:}i_{\text{right}}] \leftarrow \mathbf{x}[i_{\text{base}}{:}n_h{:}i_{\text{right}}]A^{(h)}$
        $i_{\text{base}} \leftarrow i_{\text{base}} + n_h i_{\text{right}}, j_{\text{base}} \leftarrow j_{\text{base}} + m_h i_{\text{right}}$
    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 $\mathbf{c} = \mathbf{x}[0{:}1{:}m]$

In this section, we present the operations developed in our framework for vector-matrix multiplication. In addition, an example of the complex dispatch logic made possible by the operation context mechanism is described.

Implemented matrix multiplication routines for the data structure (see Figure 4.4 on page 34) with a base and a target vector include

- Multiplication of vectors with dense and sparse matrices with and without parallelization.

  If parallel execution is desired, vectors are partitioned into chunks of length equal to a *blocking factor*. Multiplications involving each chunk are executed on the thread pool provided by the .NET Common Language Runtime.

- If one of the vectors is a `VectorSplitWrapper` but the matrix is not a block matrix, the vector must be *unwrapped* first and the dispatch should be repeated.

- Multiplication with block matrices by delegation to the constituent blocks of the matrix (Algorithm 4.1 on page 39). 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. If the operand and target vectors are not block vectors, a `VectorSplitWrapper` must be created first.

- Multiplication by a linear combination of matrices is delegated to the constituent matrices (Algorithm 4.2 on page 40).

- Multiplications $\mathbf{b} \cdot \operatorname{diag}\{\mathbf{a}\}$ by diagonal matrices are executed as elementwise products $\mathbf{b} \odot \mathbf{a}$. 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 [7, 18] as shown in Algorithm 4.3 on page 40.

  The algorithm requires access to partial slices of a vector $\mathbf{x}[s{:}t{:}m]$. As a sliding window of partial vectors is used, the multiplication uses flexible partial vectors to avoid repeated object construction in the inner loop. If both vectors are `NativeVector` and the Kronecker product only contains identity matrices and `NativeSparseMatrix` instances, a specialized subroutine is used which performs pointer arithmetic directly without the use of partial vectors. This is an example of an optimization that was added after profiling the computation of vector-matrix products.

  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 4.3.

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 [34] and SPLIT [28] algorithms, which are more amenable to parallel execution than SHUFFLE. Their implementation is in the scope of our future work.

Multiplication of a matrix with a vector from the right is implemented similarly.

Chapter 5

Algorithms for stochastic analysis

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

In steady-state and sensitivity analysis, linear equations of the form $\mathbf{x}A = \mathbf{b}$ are solved, such as eq. (2.1) on page 4. The steady-state probability vector is the solution of the linear system

$$\frac{\mathrm{d}\boldsymbol{\pi}}{\mathrm{d}t} = \boldsymbol{\pi}Q = \mathbf{0}, \quad \boldsymbol{\pi}\mathbf{1}^{\mathrm{T}} = 1, \qquad \text{(2.1 revisited)}$$

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 computation 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 differential equations 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 developed several linear equation solvers and transient analysis methods. Where it is reasonable, the implementation is independent of the form of the generator matrix Q.

The implementation of low-level linear algebra operations is also decoupled from the numerical algorithms and data structure. This strategy enables further configurability by replacing the operations at runtime, as described in Chapter 4.

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.

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 $i = 0, 1, \ldots, n-1$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** i **do** $a[i,j] \leftarrow a[i,j] - \sum_{k=0}^{j-1} a[i,k]a[k,j]$
  for $j \leftarrow i+1$ **to** $n-1$ **do** $a[i,j] \leftarrow \left(a[i,j] - \sum_{k=0}^{i-1} a[i,k]a[i,j]\right) \big/ a[i,i]$
Let A_L, A_D and A_U refer to the strictly lower triangular, diagonal and strictly upper triangular parts of A, respectively.
$L \leftarrow A_L + A_D$
$U \leftarrow A_U + I$
return L, U

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

We used Crout's LU decomposition [66, 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 analyzed 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.

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

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** $\mathbf{x}, \mathbf{z} \in \mathbb{R}^n$
2 **if** $\mathbf{b} = \mathbf{0}$ **then** $\mathbf{z} \leftarrow \mathbf{0}$ // Skip forward substitution for homogenous equations
3 **else for** $j \leftarrow 0$ **to** $n-1$ **do** $z[j] \leftarrow b[j] \cdot \sum_{i=0}^{j-1} u[i,j]$
4 **if** $l[n-1,n-1] \approx 0$ **then**
5 | **if** $z[n-1] \approx 0$ **then** $x[n-1] \leftarrow 0$ // Set the free parameter to 1
6 | **else error** "inconsistent linear equation system"
7 **else** $x[n-1] \leftarrow z[n-1]/l[n-1,n-1]$
8 **for** $j \leftarrow n-2$ **downto** 0 **do**
9 | **if** $l[j,j] \approx 0$ **then error** "more than one free parameter"
10 | $x[j] \leftarrow \left(z[i] - \sum_{i=j+1}^{n-1} x[i]l[i,j]\right) \Big/ l[j,j]$
11 **return** $\mathbf{x}$

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
1 **allocate** $\mathbf{x}' \in \mathbb{R}^n$ // Previous iterate for convergence test
2 **repeat**
3 | $\mathbf{x}' \leftarrow \mathbf{x}$ // Save the previous vector
4 | $\mathbf{x} \leftarrow f(\mathbf{x}')$
5 **until** $\|\mathbf{x}' - \mathbf{x}\| \le \tau$
6 **return** $\mathbf{x}$ and $\|\mathbf{x}Q - \mathbf{b}\|$

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 homogeneous equations in Markovian 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 on page 45.

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

For a detailed discussion stopping criteria and iterate normalization in steady-state CTMC analysis, we refer to [82, Section 10.3.5].

Power iteration

Power iteration [82, 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 infinitesimal 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 45 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 $\mathbf{x}$.

Algorithm 5.4 Power iteration.

$\alpha^{-1} \leftarrow 1/\max_i |a[i,i]|$
repeat
  $\mathbf{x}' \leftarrow \mathbf{x}A$ ▷ VectorMatrixMultiplyFromLeft
  $\mathbf{x}' \leftarrow \mathbf{x}' + (-1)\cdot\mathbf{x}$ ▷ In-place VectorAdd
  $\epsilon \leftarrow \alpha^{-1}\|\mathbf{x}'\|$ ▷ VectorL1Norm
  $\mathbf{x} \leftarrow \mathbf{x} + \alpha^{-1}\mathbf{x}'$ ▷ In-place VectorAdd
until $\epsilon \leq \tau$

Observation 5.1 If $\mathbf{b} = 0$ and A is an infinitesimal 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 [82, Section 10.3.2–3] repeatedly solve a system of simultaneous equations of a specific form.

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

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

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 iteration, is realized in Algorithm 5.5. The value 1 of the over-relaxation parameter ω corresponds to ordinary

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

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

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

Jacobi iteration. Values $\omega > 1$ may accelerate convergence, while $0 < \omega < 1$ may help diverging Jacobi iteration converge.

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 parallelization 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 entirely handled by the initial guess. We instead transform the equation into the form in eq. (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.

5.1.3 Group iterative methods

Group or *block* iterative methods Stewart [82, 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. Each block of $\mathbf{x}$ corresponds to a group a variables that are simultaneously solved for.

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. 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.

In Jacobi over-relaxation, parallelization 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.

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
    $\mathbf{x}' \leftarrow \mathbf{x}, \mathbf{c} \leftarrow \mathbf{b}$
    $\mathbf{c} \leftarrow \mathbf{c} + (-1) \cdot \mathbf{x}'A_{OB}$ ▷ VectorMatrixAccumulateMultiplyFromLeft
    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 $\epsilon \leq \tau$

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** ▷ VectorMatrixAccumulateMultiplyFromLeft
                $\mathbf{c} \leftarrow \mathbf{c} + (-1) \cdot \mathbf{x}[j]A[i,j]$
        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 $\epsilon \leq \tau$

Gauss–Seidel successive over-relaxation cannot be parallelized easily. However 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 behavior varies widely, we do not perform the transformation for homogeneous 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 behavior of group iterative methods, we refer to Greenbaum [40, Chapter 14] and Courtois and Semal [27].

5.1.4 Krylov subspace methods

Projectional iterative methods are iterative linear equation solvers that produce a sequence of approximate solutions $\mathbf{x}_k$ of the linear equation $\mathbf{x}A = \mathbf{b}$ that satisfy the Petrov–Galerkin conditions [73, Section 5.1.1]

$$\mathbf{x}_k \in \mathcal{K}_k, \quad \mathbf{r}_k = \mathbf{b} - \mathbf{x}_k A \perp \mathcal{L}_k, \tag{5.5}$$

where $\mathcal{K}_k$ and $\mathcal{L}_k$ are two subspaces of $\mathbb{R}^n$ and $\mathbf{r}_k$ is residual in the kth iteration.

Krylov subspace iterative methods correspond to the choice

$$\mathcal{K}_k = \mathcal{K}_k(A, \mathbf{r}_0) = \operatorname{span}\{\mathbf{r}_0, \mathbf{r}_0 A, \mathbf{r}_0 A^2, \ldots, \mathbf{r}_0 A^{k-1}\},$$

where $\mathcal{K}_k(A, \mathbf{r}_0)$ is the *kth Krylov subspace* of A and the initial residual $\mathbf{r}_0 = \mathbf{b} - \mathbf{x}_0 Q$.

The smallest $m \in \mathbb{N}$ such that $\dim \mathcal{K}_m(A, \mathbf{r}_0) = \dim \mathcal{K}_{m+1}(A, \mathbf{r}_0)$ is called the *grade* of A with respect to $\mathbf{r}_0$. Hence $k \leq m$ implies $\dim \mathcal{K}_k(A, \mathbf{r}_0) = k$. Krylov subspace solvers usually suppose that the algorithm terminates at some iteration k^* such that $k^* \leq m$, therefore the dimension of $\mathcal{K}_k$ increases with each iteration. The contrary situation leads to stagnation, because $\mathcal{K}_k \subseteq \mathcal{K}_{k+1}$ together with $\dim \mathcal{K}_k = \dim \mathcal{K}_{k+1}$ $(k \geq m)$ implies $\mathcal{K}_k = \mathcal{K}_{k+1}$.

The subspace $\mathcal{L}_k$ also must be a k-dimensional subspace of $\mathbb{R}^n$. Conceptually, while the Krylov subspace $\mathcal{K}_k$ "expands" in dimensionality every iteration, the subspace $\mathcal{L}_k$ likewise fills the space to make additional residuals forbidden by the Petrov–Galerkin condition (5.5).

If $A \in \mathbb{R}^{n \times n}$ is of full rank and grade, Krylov subspace solvers find the exact solution of the linear equation in at most n iterations with exact arithmetic. The only possible orthogonal residual is the zero vector $\mathbf{0}$ if $\mathcal{L}_n = \mathbb{R}^n$ holds. While n is usually too large for this to be practical, convergence often happens with suitable accuracy after a small number of iterations.

Note that problems may arise when A is singular, which may worsen the convergence behavior. This is the case in CTMC analysis, where the infinitesimal generator matrix Q is of rank $n-1$.

Some Krylov subspace methods for nonsymmetric matrices in wide use are Generalized Minimum Residual (GMRES) [72], Bi-Conjugate Gradient Stabilized (BiCGSTAB) [86], Conjugate Gradient Squared (CGS) [79] and IDR(s) [81].

Generalized Minimal Residual (GMRES)

Generalized Minimal Residual (GMRES) [73, Section 6.5.1; 72] a Krylov subspace method for nonsymmetric linear systems. It is based on the choice

$$\mathscr{L}_k = \mathscr{K}_k A = \{\mathbf{r}_0 A, \mathbf{r}_0 A^2, \ldots, \mathbf{r}_0 A^k\}.$$

With this choice, the Petrov–Galerkin condition (5.5) minimizes the Euclidean norm of the residuals in each iteration, i.e.

$$\mathbf{x}_k \in \mathscr{K}_k \text{ such that } \mathbf{r}_k = \mathbf{b} - \mathbf{x}_k A \perp \mathscr{L}_k \quad \Longleftrightarrow \quad \mathbf{x}_k = \operatorname*{arg\,min}_{\mathbf{x} \in \mathscr{K}_k} \|\mathbf{b} - \mathbf{x}Q\|_2. \tag{5.6}$$

Unfortunately, the solution of eq. (5.6) requires the storage of a basis of $\mathscr{K}_k$, which is a k dimensional subspace of $\mathbb{R}^n$. Thus, each iteration requires the allocation of an additional vector. Solution of a linear system with GMRES requires up to n additional floating-point vectors of n elements each, i.e. $O(n^2)$ floating-point numbers. This property makes GMRES a "long recurrence" algorithm.

The high memory requirements may be alleviated by discarding the basis of $\mathscr{K}_k$ and restarting the iteration from another initial guess $\mathbf{x}_0$ if no solution is obtained after ℓ iterations. The resulting algorithm is called GMRES(ℓ).

The convergence behavior of full GMRES is often excellent. However, due to impractical memory requirements, we did not implement GMRES as a numerical solver in our framework. We instead use BiCGSTAB and IDR(s)STAB(ℓ), Krylov subspace solvers incorporating GMRES(ℓ)-like steps.

Bi-Conjugate Gradient Stabilized (BiCGSTAB)

Bi-Conjugate Gradient Stabilized (BiCGSTAB) [73, Section 7.4.2; 86] is a Krylov subspace method where [75]

$$\mathscr{L}_k = \mathscr{K}_k(A^{\mathrm{T}}, \tilde{\mathbf{r}}_0) \cdot (\Omega_k(A)^{\mathrm{T}})^{-1}, \quad \Omega_k(A) = \begin{cases} \Omega_{k-1}(A) \cdot (I - \omega_k A) & \text{if } k \geq 1, \\ I & \text{if } k = 0. \end{cases} \tag{5.7}$$

The *initial shadow residual* $\tilde{\mathbf{r}}_0$ must satisfy $\mathbf{r}_0 \tilde{\mathbf{r}}_0^{\mathrm{T}} \neq 0$ and must not be an eigenvector of Q^{T}. Usually, $\tilde{\mathbf{r}}_0 = \mathbf{r}_0$, which is the convention we use in our implementation.

Equivalently, BiCGSTAB is a Krylov subspace method which produces residuals

$$\mathbf{r}_k \in \mathscr{G}_k, \quad \mathscr{G}_k = \begin{cases} (\mathscr{G}_k \cap \tilde{\mathbf{r}}_0^{\perp})(I - \omega_k A) & \text{if } k \geq 1, \\ \mathbb{R}^n & \text{if } k = 0, \end{cases} \tag{5.8}$$

where $\mathcal{A}^\perp$ is the set of vector orthogonal to $\mathcal{A}$. It can be shown that [78]

$$\mathcal{G}_k = \mathcal{S}(\Omega_k, A, \tilde{\mathbf{r}}_0) = \{\mathbf{v} \cdot \Omega_k(A) : \mathbf{v} \perp \mathcal{H}_k(A^{\mathrm{T}}, \tilde{\mathbf{r}}_0)\},$$

where $\mathcal{S}(\Omega, A, \tilde{\mathbf{r}}_0)$ is the Ωth *Sonneveld subspace* generated by A and $\tilde{\mathbf{r}}_0$) of order $k = \deg \Omega$. Hence $\mathcal{L}_k = \mathcal{G}_k^\perp$, which makes BiCGSTAB equivalent to another Krylov subspace method, Induced Dimensionality Reduction (IDR) [78] in exact arithmetic.

BiCGSTAB is a "short recurrence", that is, the number of allocated intermediate vectors does not depend on the number of variables in equation system.

Implementation We selected BiCGSTAB as the first Krylov subspace solver integrated into our framework because of its good convergence behavior and low memory requirements. BiCGSTAB only requires the storage of 7 vectors, which makes it suitable even for large state spaces with large state vectors.

Algorithm 5.9 shows the pseudocode for BiCGSTAB. Our implementation is based on the MATLAB code[1] by Barrett et al. [5].

The inner loop of BiCGSTAB is the composed of two procedures. The bi-conjugate gradient (Bi-CG) part in lines 9–20 calculates a residual $\mathbf{t} \in \mathcal{G}_{k-1}A$ and its associated approximate solution $\mathbf{x} \in \mathcal{H}_k$. The GMRES(1) part in lines 21–31 selects $\omega_k \in \mathbb{R}$ and calculates a new residual $\mathbf{r} \in \mathcal{G}_k$ such that the Euclidean norm $\|\mathbf{r}\|_2$ is minimized. This part improves convergence over the original Bi-Conjugate Gradient algorithm.

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 of appropriate preconditioner matrices M is not trivial [51], 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 47, 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 47.

Induced Dimensionality Reduction Stabilized (IDRSTAB)

Induced Dimensionality Reduction Stabilized (IDR(s)STAB(ℓ)) [78] is Krylov subspace solver that generalizes BiCGSTAB and IDR techniques to provide converge behaviors closely matching GMRES while maintaining the short recurrence property.

As the algorithm developed relatively recently in 2010, high performance implementations of IDR(s)STAB(ℓ) are not widely available. To our best knowledge,

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

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 $\mathbf{r}, \tilde{\mathbf{r}}_0, \mathbf{v}, \mathbf{p}, \mathbf{s}, \mathbf{t} \in \mathbb{R}^n$
$\mathbf{r} \leftarrow \mathbf{b}$ ▷ VectorSet
$\mathbf{r} \leftarrow \mathbf{r} + (-1) \cdot \mathbf{x}A$ ▷ VectorMatrixAccumulateMultiplyFromLeft
if $\|\mathbf{r}\| \leq \tau$ **then**
    message "initial guess is correct, skipping iteration"
    return $\mathbf{x}$
$\tilde{\mathbf{r}}_0 \leftarrow \mathbf{r}, \mathbf{v} \leftarrow \mathbf{0}, \mathbf{p} \leftarrow \mathbf{0}, \rho' \leftarrow 1, \alpha \leftarrow 1, \omega \leftarrow 1$
while true **do**

Bi-CG step

    $\rho \leftarrow \mathbf{r_0} \cdot \mathbf{r}$ ▷ VectorScalarProduct
    if $\rho \approx 0$ **then error** "breakdown: $\mathbf{r} \perp \tilde{\mathbf{r}}_0$"
    $\beta \leftarrow \rho / \rho' \cdot \alpha / \omega$
    $\mathbf{p} \leftarrow \mathbf{r} + \beta \cdot \mathbf{p}$ ▷ VectorAdd
    $\mathbf{p} \leftarrow \mathbf{p} + (-\beta\omega) \cdot \mathbf{v}$ ▷ In-place VectorAdd
    $\mathbf{v} \leftarrow \mathbf{p}Q$ ▷ VectorMatrixMultiplyFromLeft
    $\alpha \leftarrow \rho / (\tilde{\mathbf{r}}_0 \cdot \mathbf{v})$ ▷ VectorScalarProduct
    $\mathbf{r} \leftarrow \mathbf{s} + (-\alpha) \cdot \mathbf{s}$ ▷ VectorAdd
    if $\|\mathbf{s}\| < \tau$ **then**
        $\mathbf{x} \leftarrow \mathbf{x} + \alpha \cdot \mathbf{p}$ ▷ In-place VectorAdd
        message "early return with vanishing $\mathbf{s}$"
        return $\mathbf{x}$

GMRES(1) step

    $\mathbf{t} \leftarrow \mathbf{s}A$ ▷ VectorMatrixMultiplyFromLeft
    $tLengthSquared \leftarrow \mathbf{t} \cdot \mathbf{t}$ ▷ VectorScalarProduct
    if $tLengthSquared \approx 0$ **then error** "breakdown: $\mathbf{t} \approx \mathbf{0}$"
    $\omega \leftarrow (\mathbf{t} \cdot \mathbf{s}) / tLengthSquared$ ▷ VectorScalarProduct
    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** $\mathbf{x}$
    $\mathbf{s} \leftarrow \mathbf{t} + (-\omega) \cdot \mathbf{r}$ ▷ VectorAdd
    $\rho' \leftarrow \rho$

IDR(s)STAB(ℓ) was not investigated for use in CTMC analysis despite its promising results solving differential equations arising from finite element problems. Therefore, we are currently focusing research and development effort into integrating IDR(s)STAB(ℓ) into our stochastic analysis. Special attention is paid to its behavior on steady-state equations with infinitesimal generator matrices and other linear systems arising from CTMC analysis.

IDR(s)STAB(ℓ) merges two generalizations of BiCGSTAB:

- The first idea comes from IDR(s) [81], a Krylov subspace solver based on Sonneveld subspaces. A block version of eq. (5.8) constraints the residual $\mathbf{r}_k$

  $$\mathbf{r}_k \in \mathcal{G}_k = \mathcal{S}(\Omega_k, A, \widetilde{R}_0) = \{\mathbf{v} \cdot \Omega_k(A) : \mathbf{v} \perp \mathcal{K}_k(A, \tilde{R}_0)\},$$

  where $\mathcal{K}_k(A, \tilde{R}_0)$ is the kth *row* Krylov subspace of $A \in \mathbb{R}^{n \times n}$ with respect to $\tilde{R}_0 \in \mathbb{R}^{s \times n}$

  $$\mathcal{K}_k(A, \widetilde{R}_0) = \text{span}\{\tilde{\mathbf{r}}_0[i], \tilde{\mathbf{r}}_0[i]A, \ldots, \tilde{\mathbf{r}}_0[i]A^{k-1} : i = 0, 1, \ldots, s-1\}$$

  and $\tilde{\mathbf{r}}_0[i]$ is the ith row $\tilde{R}_0$.

  Higher values of s, i.e. higher dimensional initial shadow spaces, may accelerate convergence, at the cost of allocation additional intermediate vectors.

- The second generalization, which is called BiCGSTAB(ℓ) [76], replaces the stabilizer polynomial Ω_k from eq. (5.7) with

  $$\Omega_k(A) = \Omega_{k-1}(A) \cdot (I - \gamma[0]A - \gamma[1]A^2 - \cdots - \gamma[\ell-1]A^\ell),$$

  i.e. degree of the stabilizer polynomial Ω increases by ℓ instead of 1 every iteration. The increase is described by the vector $\vec{\gamma} \in \mathbb{R}^\ell$.

  The higher-order stabilization, also called a GMRES(ℓ) step, improves convergence behavior with unsymmetric matrices that have complex spectrum. However, the number of intermediate vectors, thus the amount of required memory, also grows.

A single dimensional initial shadow space ($s = 1$) and first-order stabilization ($\ell = 1$) make IDR(s)STAB(ℓ) identical to BiCGSTAB. Moreover, $\ell = 1$ results is behavior equivalent to IDR(s), while $s = 1$ results in behavior equivalent to BiCGSTAB(ℓ).

These correspondences make IDR(s)STAB(ℓ) a promising candidate for use in configurable stochastic analysis, as different settings of (s, ℓ) bring the power of multiple algorithms to the modelers' disposal.

Algorithm 5.10 IDR(s)STAB(ℓ).

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 $\mathbf{R} \in \mathbb{R}^{(\ell+1)\times n}, \mathbf{U}, \mathbf{V} \in \mathbb{R}^{((\ell+2)\times s)\times n}, \widetilde{\mathbf{R}}_0 \in \mathbb{R}^{s\times n}$
allocate $\sigma \in \mathbb{R}^{s\times s}, \vec{m} \in \mathbb{R}^s, \vec{\alpha} \in \mathbb{R}^s, \vec{\beta} \in \mathbb{R}^s, G \in \mathbb{R}^{\ell\times\ell}, \vec{\rho} \in \mathbb{R}^\ell, \vec{\gamma} \in \mathbb{R}^\ell$
allocate $\mathbf{C}, \mathbf{D} \in \mathbb{R}^{s\times n}$

Initialize shadow residuals

for $j = 0$ **to** $s-1$ **do**
  Sample $\tilde{\mathbf{r}}_0[j]$ from an n-dimensional standard normal distribution
  $\tilde{\mathbf{r}}_0[j] \leftarrow \tilde{\mathbf{r}}_0[j] + \left(-\sum_{k=0}^{n-1} \tilde{r}_0[j][k]/n\right)\mathbf{1}$ ▷ In-place VectorAdd
  for $q = 0$ **to** $s-1$ **do**
    $\tilde{\mathbf{r}}_0[j] \leftarrow \tilde{\mathbf{r}}_0[j] + (-\tilde{\mathbf{r}}_0[j] \cdot \tilde{\mathbf{r}}_0[q])\,\tilde{\mathbf{r}}_0[q]$ ▷ In-place VectorAdd
  $\tilde{\mathbf{r}}_0[j] \leftarrow (1/\|\tilde{\mathbf{r}}_0[j]\|_2)\,\tilde{\mathbf{r}}_0[j]$ ▷ In-place VectorScale

Initialize residuals and intermediate vectors

$\mathbf{r}[0] \leftarrow \mathbf{b}$, $\mathbf{r}[0] \leftarrow \mathbf{r}[0] + (-1)\,\mathbf{x}A$ ▷ VectorMatrixAccumulateMultiplyFromLeft
for $q \leftarrow 0$ **to** $s-1$ **do**
  if $q = 0$ **then** $\mathbf{c}[0] \leftarrow (-1)\,\mathbf{x}$, $\mathbf{u}[0,0] \leftarrow \mathbf{r}[0]$ ▷ VectorSet
  else $\mathbf{c}[q] \leftarrow \mathbf{u}[0,q-1]$, $\mathbf{u}[0,q] \leftarrow \mathbf{u}[1,q-1]$ ▷ VectorSet
  $U[1,q] \leftarrow U[0,q]A$ ▷ VectorMatrixMultiplyFromLeft
  for $k \leftarrow 0$ **to** $q-1$ **do**
    $proj \leftarrow \mathbf{u}[0,k] \cdot \mathbf{u}[0,q]$ ▷ VectorScalarProduct
    $\mathbf{c}[q] \leftarrow \mathbf{c}[q] + (-proj)\,\mathbf{c}[k]$ ▷ In-place VectorAdd
    $\mathbf{u}[0,q] \leftarrow U[0,q] + (-proj)\,\mathbf{u}[0,k]$, $\mathbf{u}[1,q] \leftarrow U[1,q] + (-proj)\,\mathbf{u}[1,k]$
  $norm \leftarrow \|\mathbf{u}[0,q]\|_2$ ▷ VectorScalarProduct
  $\mathbf{c}[q] \leftarrow (1/norm)\,\mathbf{c}[q]$, ▷ In-place VectorScale
  $\mathbf{u}[0,q] \leftarrow (1/norm)\,\mathbf{u}[0,q]$, $\mathbf{u}[1,q] \leftarrow (1/norm)\,\mathbf{u}[1,q]$

Iteration

while $\|\mathbf{r}[0]\| > \epsilon$ **do**
  Perform IDR step from Algorithm 5.11
  Perform GMRES(ℓ) step from Algorithm 5.12
return $\mathbf{x}$

Algorithm 5.11 IDR(s)STAB(ℓ) IDR step.

for $j \leftarrow 1$ **to** ℓ **do** // For every repetition step
  for $k \leftarrow 0$ **to** $s-1$ **do**
    for $q \leftarrow 0$ **to** $s-1$ **do** $\sigma[q,k] \leftarrow \mathbf{u}[j,q] \cdot \tilde{\mathbf{r}}_0[k]$ ▷ VectorScalarProduct
    $m[k] \leftarrow \mathbf{r}[j-1] \cdot \tilde{\mathbf{r}}_0[k]$ ▷ VectorScalarProduct
  Solve $\vec{\alpha}\sigma = \vec{m}$ for $\vec{\alpha}$
  for $q \leftarrow 0$ **to** $s-1$ **do**
    $\mathbf{x} \leftarrow \mathbf{x} + \alpha[q]\,\mathbf{u}[0,q]$ ▷ In-place VectorAdd
    for $k \leftarrow 0$ **to** $j-1$ **do** $\mathbf{r}[k] \leftarrow \mathbf{r}[k] + (-\alpha[q])\,\mathbf{u}[k+1][q]$
  $\mathbf{r}[j-1] \leftarrow \mathbf{r}A$ ▷ VectorMatrixMultiplyFromLeft
  for $q \leftarrow 0$ **to** $s-1$ **do** // Build a new basis for the shadow space
    if $q = 0$ **then**
      $\mathbf{d}[0] \leftarrow (-1)\,\mathbf{x}$ ▷ VectorScale
      for $k \leftarrow 0$ **to** $j-1$ **do** $\mathbf{v}[k,0] \leftarrow \mathbf{r}[k]$ ▷ VectorSet
    else
      $\mathbf{d}[q] \leftarrow \mathbf{v}[0,q-1]$ ▷ VectorSet
      for $k \leftarrow 0$ **to** $j-1$ **do** $\mathbf{v}[k,q] \leftarrow \mathbf{v}[k+1,q-1]$ ▷ VectorSet
    for $k \leftarrow 0$ **to** $s-1$ **do** $m[k] \leftarrow \mathbf{v}[j,q] \cdot \tilde{\mathbf{r}}_0[k]$ ▷ VectorScalarProduct
    Solve $\vec{\beta}\sigma = \vec{m}$ for $\vec{\beta}$
    for $i \leftarrow 0$ **to** $s-1$ **do**
      $\mathbf{d}[q] \leftarrow \mathbf{d}[q] + (-\beta[i])\,\mathbf{c}[i]$ ▷ In-place VectorAdd
      for $k \leftarrow 0$ **to** j **do** $\mathbf{v}[k,q] \leftarrow \mathbf{v}[k,q] + (-\beta[i])\,\mathbf{u}[k,i]$
    $\mathbf{v}[j+1,q] \leftarrow \mathbf{v}[j,q]A$ ▷ VectorMatrixMultiplyFromLeft
    for $i \leftarrow 0$ **to** $q-1$ **do** // Attempt orthonormalization
      $proj \leftarrow \mathbf{v}[j,q] \cdot \mathbf{v}[j,i]$ ▷ VectorScalarProduct
      $\mathbf{d}[q] \leftarrow \mathbf{d}[q] + (-proj)\,\mathbf{d}[i]$ ▷ In-place VectorAdd
      for $k \leftarrow 0$ **to** $j+1$ **do** $\mathbf{v}[k,q] \leftarrow \mathbf{v}[k,q] + (-proj)\,\mathbf{v}[k,i]$
    $norm \leftarrow \|\mathbf{v}[j,q]\|_2$ ▷ VectorScalarProduct
    if $norm < \epsilon$ **then** // Gram–Schmidt breakdown
      message "early exit with $\mathbf{v}[j,q] \approx \mathbf{0}$"
      $sum \leftarrow \sum_{k=0}^{n-1} d[q][k]$, $\mathbf{x} \leftarrow (1/sum)\,d[q][k]$ ▷ In-place VectorScale
      return x
    $\mathbf{d}[q] \leftarrow (1/norm)\,\mathbf{d}[q]$ ▷ In-place VectorScale
    for $k \leftarrow 0$ **to** $j+1$ **do** $\mathbf{v}[k,q] \leftarrow (1/norm)\,\mathbf{v}[k,q]$ ▷ In-place VectorScale
  Swap the references to **C** and **D**
  Swap the references to **U** and **V**

Algorithm 5.12 IDR(s)STAB(ℓ) GMRES(ℓ) step.

Find $\arg\min_{\vec{\gamma}\in\mathbb{R}^\ell} \|\mathbf{r}[0] - R[1:\ell]\vec{\gamma}^{\mathrm{T}}\|_2$ **by solving the normal equation**

1 **for** $j \leftarrow 1$ **to** s **do**
2 **for** $i \leftarrow 1$ **to** s **do** $g[i,j] \leftarrow \mathbf{r}[i] \cdot \mathbf{r}[j]$ ▷ VectorScalarProduct
3 $\rho[j] \leftarrow \mathbf{r}[0] \cdot \mathbf{r}[j]$ ▷ VectorScalarProduct
4 Solve $\vec{\gamma}G = \vec{\rho}$ for $\vec{\gamma}$

Calculate the minimal residual

5 **for** $j \leftarrow 0$ **to** $j-1$ **do**
6 $\mathbf{x} \leftarrow \mathbf{x} + \gamma[j]\mathbf{r}[j]$ ▷ In-place VectorAdd
7 $\mathbf{r}[0] \leftarrow \mathbf{r}[0] + (-\gamma[j])\mathbf{r}[j+1]$ ▷ In-place VectorAdd
8 **for** $q \leftarrow 0$ **to** $s-1$ **do**
9 $\mathbf{c}[q] \leftarrow \mathbf{c}[q] + (-\gamma[j])\mathbf{u}[j,q]$ ▷ In-place VectorAdd
10 $\mathbf{u}[j,q] \leftarrow \mathbf{u}[j,q] + (-\gamma[j])\mathbf{u}[j+1,q]$ ▷ In-place VectorAdd
11 $\mathbf{u}[j+1,q] \leftarrow \mathbf{u}[j+1,q] + (-\gamma[j])\mathbf{u}[j+2,q]$ ▷ In-place VectorAdd

Implementation The pseudocode of our implementation of IDR(s)STAB(ℓ), which is based on the pseudocode of Sleijpen and Van Gijzen [78], is show in Algorithms 5.10 to 5.12. Modification to the algorithm to obtain better convergence properties with CTMC steady-state analysis are highlighted with shaded line numbers.

For convenient representation of memory requirements, we employ two different typographical styles for vectors. Vectors in bold, e.g. $\mathbf{x} \in \mathbb{R}^n$ are "long" vectors, while vectors with arrows, e.g. $\vec{\gamma}$ are "short" vectors of length s or $\ell \ll n$. Storage space of long vectors dominated memory requirements and their manipulations including vector–matrix products dominate computation time.

The algorithm works with three arrays of vectors, $\mathbf{R} \in \mathbb{R}^{(\ell+1)\times n}$, $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{((\ell+2)\times s)\times n}$, i.e. $\mathbf{r}[j'], \mathbf{u}[j,q], \mathbf{v}[j,q] \in \mathbb{R}^n$ for all $j' = 0,1,\ldots,\ell$; $j = 0,1,\ldots,\ell+1$; $q = 0,1,\ldots,s-1$.

The IDR part performs the projections, called a "repetition step", shown in Figure 5.1, for $j = 1,2,\ldots,\ell$ every iteration.

After a repetition step is complete, $\mathbf{U}$ and $\mathbf{V}$ are swapped and the process starts again with increased j or the GMRES(ℓ) part commences. Symbols with a subscript "—" sign refer to vectors from the previous repetition step.

The projections Π_i ($i = 0,1,\ldots,j$) are defined as

$$\Pi_i = I - A^{j-i}\widetilde{R}_0^{\mathrm{T}}\sigma^{-1}U[i,\cdot], \quad \sigma = \widetilde{R}_0(U[j,\cdot])^{\mathrm{T}},$$

where the matrix $U[k,\cdot]$ is the $s \times n$ matrix that has the vector $\mathbf{u}[k,q]$ as its qth row.

$$
\begin{array}{cccccccc}
\mathbf{x}_{-} & \rightarrow & \mathbf{x} & & \mathbf{v}[0,0] & & \mathbf{v}[0,1] & \cdots & \mathbf{v}[0,s-1], \\
 & \Pi_1 & \downarrow & \nearrow \Pi_0 & \downarrow A & \nearrow \Pi_0 & \downarrow A & & \downarrow A \\
\mathbf{r}_{-}[0] & \rightarrow & \mathbf{r}[0] & & \mathbf{v}[1,0] & & \mathbf{v}[1,1] & \cdots & \mathbf{v}[1,s-1], \\
 & \Pi_2 & \downarrow A & \nearrow \Pi_1 & \downarrow A & \nearrow \Pi_1 & \downarrow A & & \downarrow A \\
\mathbf{r}_{-}[1] & \rightarrow & \mathbf{r}[1] & & \mathbf{v}[2,0] & & \mathbf{v}[2,1] & \cdots & \mathbf{v}[2,s-1], \\
\vdots & \Pi_{j-1} & \vdots & & \vdots & & \vdots & & \vdots \\
\mathbf{r}_{-}[j-2] & \rightarrow & \mathbf{r}[j-2] & & \mathbf{v}[j-1,0] & & \mathbf{v}[j-1,1] & \cdots & \mathbf{v}[j-1,s-1], \\
 & \Pi_j & \downarrow A & \nearrow \Pi_{j-1} & \downarrow A & \nearrow \Pi_{j-1} & \downarrow A & & \downarrow A \\
\mathbf{r}_{-}[j-1] & \rightarrow & \mathbf{r}[j-1] & & \mathbf{v}[j,0] & & \mathbf{v}[j,1] & \cdots & \mathbf{v}[j,s-1], \\
 & & \downarrow A & \nearrow \Pi_j & \downarrow A & \nearrow \Pi_j & \downarrow A & & \downarrow A \\
 & & \boxed{\mathbf{r}[j]} & & \boxed{\mathbf{v}[j+1,0]} & & \boxed{\mathbf{v}[j+1,1]} & \cdots & \boxed{\mathbf{v}[j+1,s-1]}
\end{array}
\tag{5.9}
$$

Figure 5.1 Repetition step of the IDR(s) part of IDR(s)STAB(ℓ).

They ensure that the rows of $U[j,\cdot]$ form a basis of the Krylov subspace $\mathcal{K}_s(A\Pi_j, \mathbf{r}[j]\Pi_j)$ after the jth repetition step.

The relationships $\mathbf{r}[i+1] = \mathbf{r}iA$, $\mathbf{u}[i+1,q] = \mathbf{u}[i,q]A$, $\mathbf{v}[i+1,q] = \mathbf{v}[i,q]$ are maintained throughout the algorithm via the projections. This is signified by the gray $\downarrow A$ arrows in eq. (5.9). Notice that this means $\mathbf{r}[0]A^i = \mathbf{r}[i]$ and $U[0,\cdot]A^i = U[i,\cdot]$.

In the case of $\mathbf{r}[j]$ and $\mathbf{v}[j+1,q]$, a matrix multiplication is performed as shown by the $\downarrow A$ arrows. Vectors generated by matrix multiplication are shown in borders.

For improving numerical properties, Sleijpen and Van Gijzen [78] recommend performing Gram–Schmidt orthonormalization on $U[j,\cdot]$. The same subtractions and normalization operations must be performed on the rows of $U[i,\cdot]$, $i \neq j$ that are performed in $U[j,\cdot]$ in order to maintain their relationships. We realized the orthonormalization by the modified Gram–Schmidt process in lines 23–33 of Algorithm 5.11.

The storage of $\mathbf{R}$, $\mathbf{U}$ and $\mathbf{V}$ requires $(\ell+1)+2\cdot(\ell+1)\cdot s$ vectors of length n, while the initial shadow residual matrix $\widetilde{R}_0$ requires space equal to s vectors of length n. Thus, $1+\ell+3s+2\ell s$ intermediate vectors are needed in addition to the initial guess $\mathbf{x}_0$ and the right vectors $\mathbf{b}$. Although the memory requirements of IDR(s)STAB(ℓ) are quite high, s and ℓ can be selected to ensure that the solution fits in the available memory.

A different formulation of the IDR(s)STAB(ℓ) principles is GBi-CGSTAB(s,ℓ) [84], which avoids the allocation of $\mathbf{V}$ by updating $\mathbf{U}$ in place, albeit with lesser numerical properties due to the lack of orthonormalization steps. Another variant by Aihara et al. [1] replaces some vector updates with matrix multiplications to improve accuracy.

Numerical problems and breakdown Unfortunately, our initial experiments with IDR(s)STAB(ℓ) in Markovian steady-state analysis did not lead to success when one of the parameters s or ℓ were set to values strictly larger than 1. The only case that

managed to decrease the norm of the residual reliably, $s = \ell = 1$ is equivalent to BiCGSTAB, but due to properties of the IDR formulation,

In steady-state analysis, equations of the form

$$\mathbf{x}Q = \mathbf{0}, \quad \mathbf{x}\mathbf{1}^{\mathrm{T}} = 1 \tag{5.10}$$

are solved, where the infinitesimal generator matrix $Q \in \mathbb{R}^{n\times n}$ is matrix of rank $n-1$ (nullity 1) which satisfies $Q\mathbf{1} = \mathbf{0}^{\mathrm{T}}$. To obtain the solution, the matrix $A = Q$ is passed to IDR(s)STAB(ℓ) along with the right vector $\mathbf{b} = 0$ and the initial guess $\mathbf{x}$. The initial guess is chosen to satisfy $\mathbf{x}\mathbf{1}^{\mathrm{T}} = 1$.

We have identified the following three numerical problems in the original version of the algorithm algorithm that lead to eventual breakdown:

1. The quantity $\mathbf{x}\mathbf{1}^{\mathrm{T}}$ may increase indefinitely instead of staying constant. Thus, the normalization condition in eq. (5.10) is violated. Unconstrained increase results in the loss of precision and overflow of the elements of $\mathbf{x}$ such that the normalization cannot be restored by the division $\hat{\mathbf{x}} = \mathbf{x}/\mathbf{x}\mathbf{1}^{\mathrm{T}}$.

2. The vectors $\mathbf{V}[j,\cdot]$ that shall form the basis of the Krylov subspace $\mathcal{K}_s(A\Pi_j, \mathbf{r}[j]\Pi_j)$ may become linearly dependent, such that the Gram–Schmidt process in lines 23–33 of Algorithm 5.11 fail due to some $\mathbf{v}[j,q]$ becoming the zero vector.

3. The matrix σ may become singular such that the linear equation $\vec{\alpha}\sigma = \vec{m}$ has no solution in line 5 of Algorithm 5.11. Since $\sigma = U[j,\cdot]\widetilde{R}_0^{\mathrm{T}}$, this corresponds the projection of the vectors $\mathbf{U}[j,\cdot]$ into span $\widetilde{\mathbf{R}}_0$ becoming linearly dependent.

   In addition, a nearly singular σ results in the accumulation of errors that cause the norm of the residual to increase exponentially instead of converging to zero.

Handling numerical problems Is this section, we present the modification made to IDR(s)STAB(ℓ) in Algorithms 5.10 to 5.12 to improve its behavior with CTMC generator matrices. We restrict our attention to the solution of equations of the form (5.10), that is, steady-state normalized solution of Markovian models with zero right vector. Thus, we will write Q instead of A for the linear equation matrix and assume a right hand side $\mathbf{b} = \mathbf{0}$.

Observation 5.2 IDR(s)STAB(ℓ) only performs updates of the approximate solution of the form $\mathbf{x} \leftarrow \mathbf{x} + \lambda\mathbf{t}$, where

1. $\mathbf{t}$ is either an initial shadow residual $\tilde{\mathbf{r}}_0[j]$ for some j
2. or there exists a vector $\mathbf{w}$ such that $\mathbf{t} = \mathbf{w}Q$.

In the second case of Observation 5.2, one may notice that

$$\mathbf{t}\mathbf{1}^{\mathrm{T}} = \mathbf{w}Q\mathbf{1}^{\mathrm{T}} = \mathbf{w}\mathbf{0}^{\mathrm{T}} = 0,$$

therefore $\mathbf{x}\mathbf{1}^{\mathrm{T}}$ can be forced to stay constant by selecting initial shadow residual vectors $\widetilde{R}_0\mathbf{1}^{\mathrm{T}} = \mathbf{0}^{\mathrm{T}}$. Line 6 of Algorithm 5.10 contains this modification. The initialization of the shadow residuals is otherwise identical to the recommendation of Sonneveld [80], which selects a random orthonormalized set of s vectors.

The handling of the second problem of the Gram–Schmidt process failure is more complicated.

Observation 5.3 If $Q \in \mathbb{R}^n$ is a rank $n-1$ matrix such that $Q\mathbf{1}^{\mathrm{T}} = \mathbf{0}^{\mathrm{T}}$ and $\mathbf{b} = \mathbf{0}$, then $\mathbf{v}[i,q]\,\mathbf{1}^{\mathrm{T}} = 0$ for all i, q.

Proof. Because $\mathbf{v}[i,q] = \mathbf{v}[i-1,q]Q$ for all $i > 0$, it suffices to show that $\mathbf{v}[0,q]\,\mathbf{1}^{\mathrm{T}} = 0$. The chain of projections in eq. (5.9) on page 60 shows that

$$\mathbf{v}[0,q] = \mathbf{w}\Pi_0 = \mathbf{w}(I - Q^j\widetilde{R}_0^{\mathrm{T}}\sigma^{-1}U[0,\cdot]),$$

where $\mathbf{w}$ is either $\mathbf{r}[0]$ or $\mathbf{v}[1,q-1]$.

Notice that $\mathbf{v}[1,q-1]\,\mathbf{1}^{\mathrm{T}} = 0$ as we have shown before and $\mathbf{r}[0]\,\mathbf{1}^{\mathrm{T}} = -\mathbf{x}Q\mathbf{1}^{\mathrm{T}} = 0$. Hence if $\mathbf{u}[0,k]\,\mathbf{1}^{\mathrm{T}} = \mathbf{0}$ for all k,

$$\begin{aligned}\mathbf{v}[0,q] = \mathbf{w}\Pi_0\mathbf{1}^{\mathrm{T}} &= \mathbf{w}\mathbf{1}^{\mathrm{T}} - \mathbf{w}Q^j\widetilde{R}_0^{\mathrm{T}}\sigma^{-1}U[0,\cdot]\,\mathbf{1}^{\mathrm{T}} \\ &= \mathbf{w}\mathbf{1}^{\mathrm{T}} - \mathbf{w}Q^j\widetilde{R}_0^{\mathrm{T}}\sigma^{-1}\mathbf{0}^{\mathrm{T}} = 0.\end{aligned}$$

Now it all remains to be shown that $\mathbf{u}[0,k]\,\mathbf{1}^{\mathrm{T}} = 0$. We will use induction over the number of repetitions performed. If $\mathbf{v}[0,k]\,\mathbf{1}^{\mathrm{T}}$ holds in the current repetition step, $\mathbf{u}[0,k]\,\mathbf{1}^{\mathrm{T}} = 0$ holds in the next, because the references to $\mathbf{U}$ and $\mathbf{V}$ are swapped every repetition step. This property is not disturbed by the performed Gram–Schmidt orthonormalizations and the GMRES(ℓ) steps.

We also notice that Algorithm 5.10 on page 57 initializes $\mathbf{u}[0,0] = \mathbf{r}[0] = -\mathbf{x}Q$ and $\mathbf{u}[0,q] = \mathbf{u}[0,q-1]Q$ for $q > 0$. This completes the induction. □

Observation 5.4 If the conditions from Observation 5.3 and $\mathbf{v}[j,q] = \mathbf{0}$ hold, then $\mathbf{v}[i,q] = \mathbf{0}$ for all $i < j$.

Proof. Suppose that there is some $\mathbf{v}[i,q] \neq \mathbf{0}$ such that $i < j$. Without loss of generality, we may assume that i is the largest index with this property, i.e. $\mathbf{v}[i',q] = \mathbf{0}$ for all $i' > i$. Then $\mathbf{0} = \mathbf{v}[i+1,q] = \mathbf{v}[i,q]Q$, therefore $\mathbf{v}[i,q]$ is the solution of the linear equation $\mathbf{x}Q = \mathbf{0}$.

However, we know that the nullspace $\ker Q = \operatorname{span}\{\boldsymbol{\pi}\}$, where $\boldsymbol{\pi}$ is the stationary distribution of the CTMC of which Q is the infinitesimal generator, i.e $\boldsymbol{\pi}Q = \mathbf{0}$, $\boldsymbol{\pi}\mathbf{1}^{\mathrm{T}} = 1$. Therefore, $\mathbf{v}[i,q] \in \operatorname{span}\{\boldsymbol{\pi}\}$ such that $\mathbf{v}[i,q]\,\mathbf{1}^{\mathrm{T}} = 0$ (Observation 5.3). This means $\mathbf{v}[i,q] = \frac{0}{1}\cdot\boldsymbol{\pi} = \mathbf{0}$. □

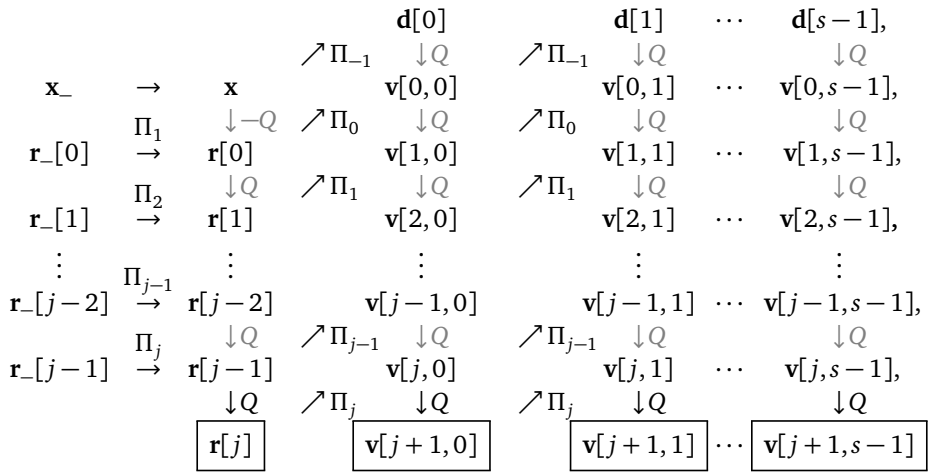


Figure 5.2 Extended repetition step of the IDR(s) part of IDR(s)STAB(ℓ).

Observation 5.4 means that if the Gram–Schmidt process breaks down with $\mathbf{v}[j,q] = \mathbf{0}$, we are unable to recover a solution vector by looking at some $\mathbf{v}[i,q]$, $i < j$, because those vector are also zero.

The projection scheme in eq. (5.9) on page 60 is extended with two additional arrays $\mathbf{C}, \mathbf{D} \in \mathbb{R}^{s\times n}$, which serve as the "(-1)th" rows of $\mathbf{U}$ and $\mathbf{V}$. This results in the extended projections shown in Figure 5.2, where $\Pi_{-1} = I - A^{j-i}\widetilde{R}_0^{\mathrm{T}}\sigma^{-1}D$. In order to avoid $\mathbf{d}[q] = \mathbf{0}$, $\mathbf{c}[0]$ is initialized to $-\mathbf{x}$ so that $\mathbf{v}[0,0] = \mathbf{r} = \mathbf{c}[0]Q$.

After a Gram–Schmidt breakdown, the solution vector $\mathbf{x}$ is obtained as

$$\mathbf{x} = \frac{1}{\mathbf{d}[q]\,\mathbf{1}^{\mathrm{T}}}\mathbf{d}[q].$$

Despite our attempts, we did not handle the third divergence problem of singular or nearly singular σ. A possible remedy is the more careful choice of the stabilizing polynomial Ω_k. Choosing $\vec{\gamma}$ through means other than the minimization of the residual norm $\|\mathbf{r}[0]\|_2$ may result in better converge behavior [71, 77].

Empirical results on the convergence of IDR(s)STAB(ℓ) in the steady-state analysis of Markovian models of various size are presented in Section 6.3 on page 79.

5.2 Transient analysis

5.2.1 Uniformization

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

$$\frac{\mathrm{d}\boldsymbol{\pi}(t)}{\mathrm{d}t} = \boldsymbol{\pi}(t)Q, \quad \boldsymbol{\pi}(t) = \boldsymbol{\pi}0 \tag{5.11}$$

by computing

$$\boldsymbol{\pi}(t) = \sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k e^{-\alpha t} \frac{(\alpha t)^k}{k!}, \tag{5.12}$$

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 probability function with rate αt at k.

Integrating both sides of eq. (5.12) to compute $\mathbf{L}(t)$ yields [70]

$$\begin{aligned}\int_0^t \boldsymbol{\pi}(u)\,\mathrm{d}u = \mathbf{L}(t) &= \sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k \int_0^t e^{-\alpha u}\frac{(\alpha u)^k}{k!}\,\mathrm{d}u \\ &= \sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k \frac{1}{\alpha} \sum_{l=k+1}^{\infty} e^{-\alpha t}\frac{(\alpha t)^l}{l!} \\ &= \frac{1}{\alpha}\sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k \left(1 - \sum_{l=0}^{k} e^{-\alpha t}\frac{(\alpha t)^l}{l!}\right).\end{aligned} \tag{5.13}$$

Both eqs. (5.12) and (5.13) can be realized as

$$\mathbf{x} = \frac{1}{W}\left(\sum_{k=0}^{k_{\text{left}}-1} w_{\text{left}}\boldsymbol{\pi}_0 P^k + \sum_{k=k_{\text{left}}}^{k_{\text{right}}} w[k-k_{\text{left}}]\boldsymbol{\pi}_0 P^k\right), \tag{5.14}$$

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. (5.14) is implemented by Algorithm 5.13. The algorithm performs *steady-state* detection in line 9 to avoid unnecessary 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 [35]. However, their algorithm is extremely complicated due to the limitations of single-precision floating-point arithmetic [47]. We implemented Burak's significantly simpler algorithm [20] in double precision instead (Algorithm 5.14 on page 66), 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

Algorithm 5.13 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: instantaneous or accumulated probability vector $\mathbf{x} \in \mathbb{R}^n$

1 **allocate** $\mathbf{x}, \mathbf{p}, \mathbf{q} \in \mathbb{R}^n$
2 $\alpha^{-1} \leftarrow 1/\max_i |a[i,i]|$
3 $\mathbf{p} \leftarrow \boldsymbol{\pi}_0$
4 **if** $w_{\text{left}} = 0$ **then** $\mathbf{x} \leftarrow \mathbf{0}$ **else** $\mathbf{x} \leftarrow w_{\text{left}} \cdot \mathbf{p}$ ▷ VectorScale
5 **for** $k \leftarrow 1$ **to** k_{right} **do**
6 $\quad \mathbf{q} \leftarrow \mathbf{p}Q$ ▷ VectorMatrixMultiplyFromLeft
7 $\quad \mathbf{q} \leftarrow \alpha^{-1} \cdot \mathbf{q}$ ▷ In-place VectorScale
8 $\quad \mathbf{q} \leftarrow \mathbf{q} + \mathbf{p}$ ▷ In-place VectorAdd
9 $\quad$ **if** $\|\mathbf{q} - \mathbf{p}\| \leq \tau$ **then**
10 $\quad\quad \mathbf{x} \leftarrow \mathbf{x} + \left(\sum_{l=k}^{k_{\text{right}}} w[l - k_{\text{left}}]\right) \cdot \mathbf{q}$ ▷ In-place VectorAdd
11 $\quad\quad$ **break**
12 $\quad$ **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 VectorAdd
13 $\quad$ **else if** $k \geq k_{\text{left}}$ **then** $\mathbf{x} \leftarrow \mathbf{x} + w[k - k_{\text{left}}] \cdot \mathbf{q}$ ▷ In-place VectorAdd
14 $\quad$ Swap the references to $\mathbf{p}$ and $\mathbf{q}$
15 $\mathbf{x} \leftarrow W^{-1} \cdot \mathbf{x}$ ▷ In-place VectorScale
16 **return** $\mathbf{x}$

eigenvalues of the infinitesimal generator matrix Q of minimum and maximum absolute value [69]. 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 values of α in line 2 of Algorithm 5.13, 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 [54], which decouples the slow and fast behaviors of the system, and adaptive uniformization [59], which varies the uniformization rate α. Alternatively, an L-stable differential equation solver may be used to solve eq. (5.11), such as TR-BDF2 [4, 69].

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

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

Algorithm 5.14 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 $W \in \mathbb{R}$

Calculate weights with high precision

$M_w \leftarrow 30, M_a \leftarrow 44, M_s \leftarrow 21$ // Constants determine cutoff estimation accuracy
$m \leftarrow \lfloor \lambda \rfloor, tSize \leftarrow \lfloor M_w \sqrt{\lambda} + M_a \rfloor, tStart \leftarrow \max\{m + M_s - \lfloor tSize/2 \rfloor, 0\}$
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**
    $tWeights[j+1] = \lambda\, tWeights[j]/(j + tStart)$

Determine normalization constant and cutoff points

$W \leftarrow 0$
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**
    $cdf \leftarrow cdf + tWeights[i]$
    $i \leftarrow i - 1$
$k_{\text{right}} \leftarrow tStart + i$

Copy weights between cutoff points

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 $k_{\text{left}}, k_{\text{right}}, \mathbf{w}, W$

and second order backward difference steps

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

which advance the time together by a step of size h_k. The constant $0<\gamma<1$ sets the break point between the two steps. We set it to $\gamma=2-\sqrt{2}\approx 0.59$ following the recommendation of Bank et al. [4].

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

$$LTE_{k+1}=\left\|2\frac{-3\gamma^4+4\gamma-2}{24-12\gamma}h_k\left[-\frac{1}{\gamma}\boldsymbol{\pi}_k+\frac{1}{\gamma(1-\gamma)}\boldsymbol{\pi}_{k+\gamma}-\frac{1}{1-\gamma}\boldsymbol{\pi}_{k+1}\right]\right\| \tag{5.15}$$

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 [69]. Moreover, the accumulated error at the end of integration may be larger than the prescribed tolerance τ, since eq. (5.15) is only an approximation of the true error.

An implementation of TR-BDF2 based on the pseudocode of A. L. Reibman and Trivedi [69] is shown in Algorithm 5.15.

In lines 10 and 13 any linear equation solver from Section 5.1 on page 44 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 [69].

5.3 Mean time to first failure

In MTFF calculation (Section 2.1.3 on page 7), quantities of the forms

$$MTFF=-\underbrace{\boldsymbol{\pi}_UQ_{UU}^{-1}}_{\gamma}\mathbf{1}^{\mathrm{T}},\quad \mathbb{P}(X(TFF_{+0})=y)=-\underbrace{\boldsymbol{\pi}_UQ_{UU}^{-1}}_{\gamma}\mathbf{q}_{UD'}^{\mathrm{T}} \tag{2.4, 2.5 revisited}$$

Algorithm 5.15 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 $\boldsymbol{\pi}_k, \boldsymbol{\pi}_{k+\gamma}, \boldsymbol{\pi}_{k+1}, \mathbf{d}_k, \mathbf{d}_{k+1}, \mathbf{y} \in \mathbb{R}^n$
$maxIncrease \leftarrow 10, leastDecrease \leftarrow 0.9$
$timeLeft \leftarrow t, h \leftarrow 1/\max_i |q[i,i]|, \gamma \leftarrow 2-\sqrt{2}, C \leftarrow \left|\frac{-3\gamma^4+4\gamma-2}{24-12\gamma}\right|, errorSum \leftarrow 0$
$\boldsymbol{\pi}_k \leftarrow \boldsymbol{\pi}_0$, $\mathbf{d}_k \leftarrow \boldsymbol{\pi}_k Q$ ▷ VectorMatrixMultiplyFromLeft
while $timeLeft > 0$ **do**
  $stepFailed \leftarrow$ false, $h \leftarrow \min\{h, timeLeft\}$
  while true **do**
    TR step
    $\mathbf{y} \leftarrow 2 \cdot \boldsymbol{\pi}_k$ ▷ VectorScale
    $\mathbf{y} \leftarrow \mathbf{y} + \gamma h \cdot \mathbf{d}_k$ ▷ In-place VectorAdd
    Solve $\boldsymbol{\pi}_{k+\gamma}(2I + -\gamma h Q) = \mathbf{y}$ for $\boldsymbol{\pi}_{k+\gamma}$ with initial guess $\boldsymbol{\pi}_k$
    BDF2 step
    $\mathbf{y} \leftarrow -\frac{(1-\gamma)^2}{\gamma} \cdot \boldsymbol{\pi}_k$ ▷ VectorScale
    $\mathbf{y} \leftarrow \frac{1}{\gamma} \cdot \boldsymbol{\pi}_{k+\gamma}$ ▷ In-place VectorScale
    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
    $\mathbf{y} \leftarrow -\frac{1}{\gamma}\mathbf{d}_k$ ▷ VectorScale
    $\mathbf{y} \leftarrow \mathbf{y} + \frac{1}{\gamma(1-\gamma)}\boldsymbol{\pi}_{k+\gamma} Q$ ▷ VectorMatrixAccumulateMultiplyFromLeft
    $\mathbf{d}_{k+1} \leftarrow \boldsymbol{\pi}_{k+1} Q$ ▷ VectorMatrixMultiplyFromLeft
    $\mathbf{y} \leftarrow \mathbf{y} + \left(-\frac{1}{1-\gamma}\right)\mathbf{d}_{k+1}$ ▷ In-place VectorAdd
    $LTE \leftarrow 2Ch\|\mathbf{y}\|, localTol \leftarrow (\tau - errorSum)/timeLeft \cdot h$
    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 $\boldsymbol{\pi}_k$

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

$$\boldsymbol{\gamma} Q_{UU} = \boldsymbol{\pi}_U \tag{5.16}$$

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 $\varphi_D = \neg \mathbf{AX}\,\text{true} \vee \varphi_1 \vee \varphi_2 \vee \cdots$ and $\varphi_U = \neg\varphi_D$, where the deadlock condition $\neg \mathbf{AX}\,\text{true}$ is added to make (5.16) 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 (5.16) can be executed as in steady-state analysis.

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.

In this section, we summarize work presented in [49] that was performed to verify the correctness of our implementation of the data structure, operations framework and the stochastic analysis algorithms.

6.1.1 Combinatorial testing

As described in Chapter 4 algorithms use the common vector and matrix data structure 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 concerning the data structure and operations is mathematical correctness regardless of the storage technique and manner of execution (e.g. parallel or sequential) used. Considering the number of implementations for a given interface and the previous requirement we used a simple unit testing design pattern (also known as interface testing pattern) as the core building block for the data structure testing [61].

The basic idea behind this pattern is to write unit tests for interface operations without any knowledge about the concrete implementation. Hiding implementation details can be achieved in a number of ways. Some unit testing frameworks, such as NUnit [65], 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 an instance if a given interface is needed, the instantiation is delegated to an abstract factory method in the test class.

Abstract test cases were created to describe desired behaviors of the operations. *Concrete test cases* are derived from abstract tests and contain calls to the data structure factory methods. Thus, the behavior of any operation may be tested for all possible data structure classes.

Abstract tests

Writing unit tests for valid parameter values is straightforward since it is possible to cover multiple valid parameter ranges with a single unit test. However testing for invalid parameter values requires some care. There must only one invalid parameter per unit test lest one error can obscure the others. This significantly increases the number of unit tests. Therefore we aimed to gather every possible invalid parameter range automatically.

We used Microsoft IntelliTest[1] [57], 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 many invalid parameter values which were used in abstract unit tests.

Concrete tests

Derived classes of abstract tests are created for every possible combinations of data structure classes by implementing the abstract factory method. Since the number of possible combinations is too large to implement manually derived classes were generated with a Microsoft Text Template Transformation Toolkit (T4) [58] template.

Pairwise testing was used to decrease the number of generated tests compared to full combinatorial testing of implementation combinations. To generate the combinations for pairwise testing we used the ACTS tool [12].

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. performance 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 successfully for both sequential and parallel operation implementations.

[1] formerly known as Pex [85]

Concrete tests were executed with both configurations provided by the operation framework as defaults, i.e. parallel and sequential, to ensure that computations are logically equivalent.

6.1.2 Software redundancy based testing

In addition to testing the data structure and operation implementations, it is vital to test the correctness of higher level algorithms used in the analysis workflow, e.g. the linear equation solver and transient analysis algorithms.

Testing every implemented algorithm with unit tests would be tremendous work that cannot be easily automated or maintained. 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 two different analysis methods can only differ from each other due to the numerical properties of the 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.

In this testing phase our analysis workflow is ran with a given configuration and the calculated reward and sensitivity values are saved. 588 mathematically consistent configurations were generated and executed on multiple benchmark models and case studies. The maximum absolute difference of the calculated results was examined as an error indicator.

6.2 Measurements

In this section we introduce the models used throughout the testing and benchmarking phase and present results about the performance of solver algorithms using the sparse

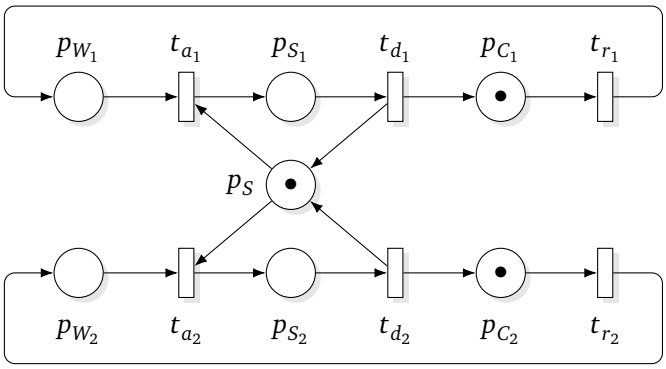


Figure 6.1 Stochastic Petri net for the *SharedResource* model.

matrix and block Kronecker decomposition matrix forms.

Every model used for testing and measurement is publicly available at `https://github.com/kris7t/stochastic-analysis` in a variety of stochastic modeling formats.

6.2.1 Models

Shared resource

Benchmark models were generated based on the Stochastic Petri Net (SPN) *SharedResource* model shown in Figure 6.1.

The model contains a number of clients competing for a central resource (p_S). Each client may run a number of processes, which are represented by token on p_{C_i}.

The model can be scaled by increasing the number of available shared resources, the number of clients and the number of processes per client. In Section 6.2.2, 5 clients were used and the number of processes and shared resources were set to equal values. In Section 6.3, all parameters were swept independently.

Symmetric (Sym), slightly asymmetric (Asym) and significantly asymmetric (Degen) versions of the model were created by assigning transition rates. In the first case, all transition rates are equal to 1, while in the third case there are orders of magnitude of difference between the transitions rates.

KanBan

The SPN model of *KanBan* (KB) manufacturing process [23] was used as another benchmark model. The model was scaled by modifying the available resources at each stage of the model resulting in an increase in the size of the state space.

Table 6.1 Operation configurations and algorithms used in the benchmarks.

Analysis task	Operation config.	Solver	Inner solver
Steady-state	Parallel	Gauss–Seidel	—
		BiCGSTAB	—
	Sequential	Group Gauss–Seidel	BiCGSTAB
		Group Gauss–Seidel	Jacobi
Transient	Parallel	Uniformization	—

Cloud performability

The represents a *Cloud* architecture [37] with physical and virtual machines serving incoming jobs using warm and cold spare resources in case of increasing load. Some aspects of the model in [37] were modified because our library currently does not support the Generalized Stochastic Petri Net (GPSN) formalism.

6.2.2 Results

Benchmarks on large models were performed with the analysis configurations shown in Table 6.1. Each analysis configuration was run with both sparse and block Kronecker generator matrices. 19 models were created by scaling the models described in Section 6.2.1.

Execution time was limited to 1 hour and the maximum memory allocation was limited to 32 GiB. As runs for scaled versions of models were omitted if the analysis was terminated due to limit crossing for a smaller model, only 152 runs were performed in total. 81 runs exited after successful calculation of a result. Selected results are shown in Table 6.2.

Figures 6.2 and 6.3 illustrate the state space size dependence of the behavior of the BiCGSTAB linear equation solver. Figures 6.4 and 6.5 illustrate the state space size dependence of the uniformization transient analysis algorithm.

As expected the storage requirements of block Kronecker generator matrices are almost an order of magnitude smaller than those of sparse matrices.

For models with less than a few millions states the sparse form of the generator matrix outperforms the block Kronecker form. However, for models with considerably bigger state space sizes the block Kronecker form reduces both the execution time and memory requirements of the analysis compared to the sparse form, which is also apparent in Figures 6.2 to 6.5. A possible reason for this phenomenon is inefficient CPU memory cache usage of the sparse storage.

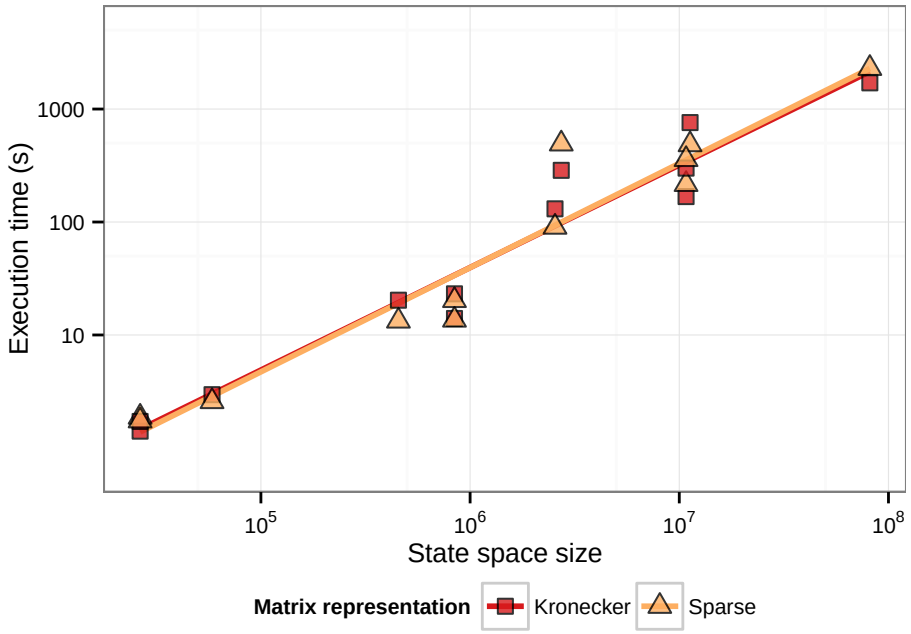


Figure 6.2 Execution times of steady-state analysis with the BiCGSTAB solver.

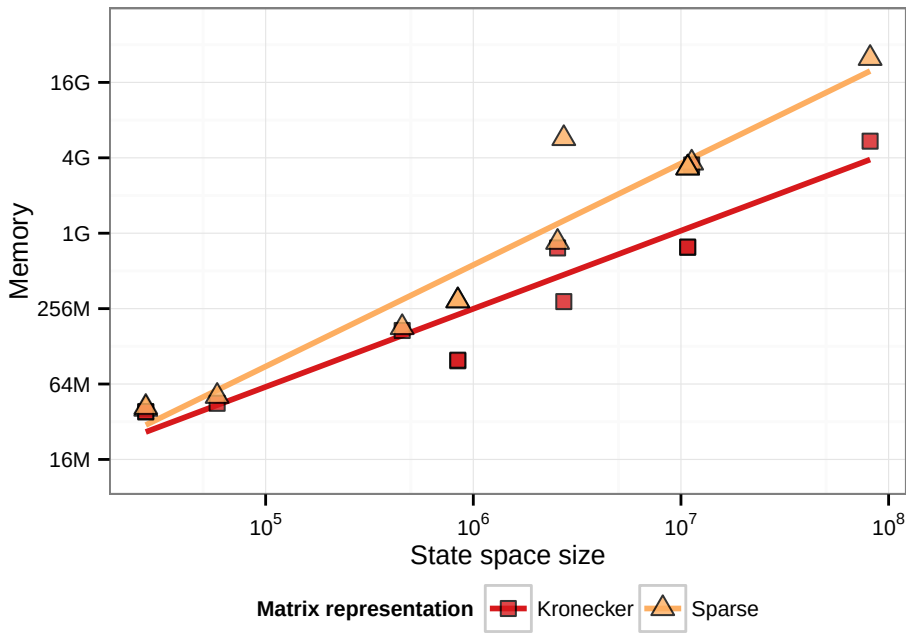


Figure 6.3 Memory consumption of steady-state analysis with the BiCGSTAB solver.

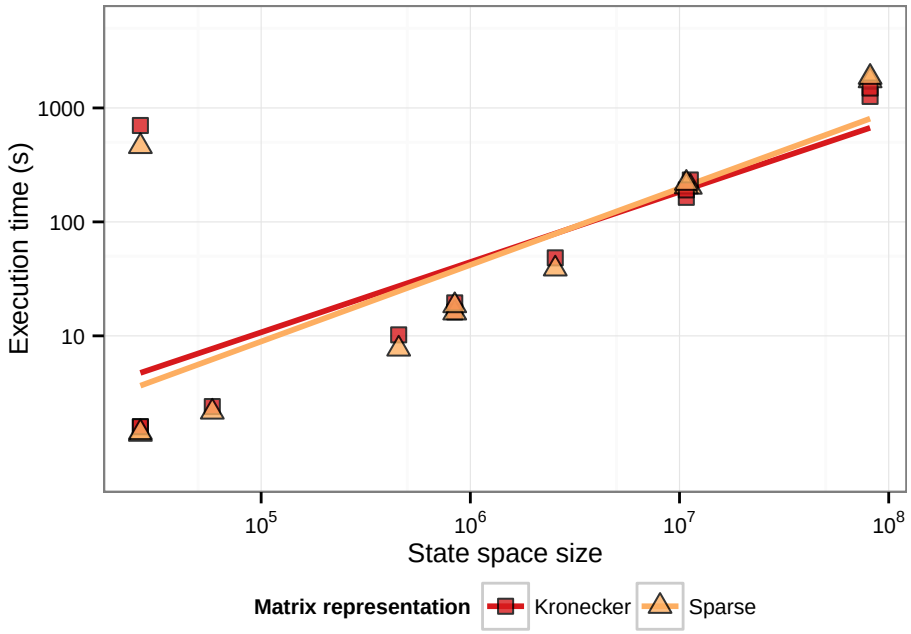


Figure 6.4 Execution times of transient analysis with uniformization.

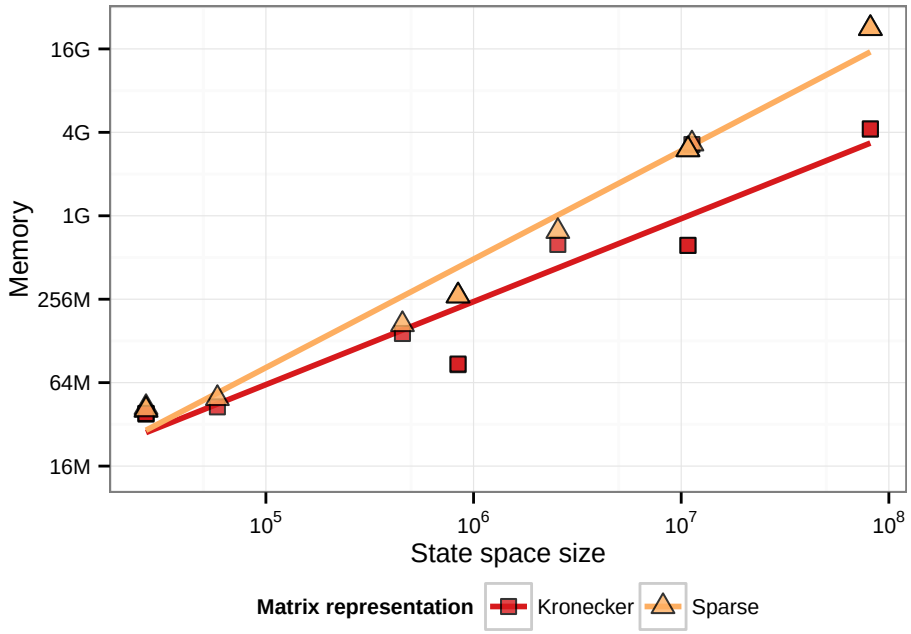


Figure 6.5 Memory consumption of transient analysis with uniformization.

Table 6.2 Selected benchmark results (adapted from [49]).

Model	States	Generator	Algorithm	Memory	Time
SR-Sym-7	10 775 710	Sparse	Uniformization	3 120 MiB	279 s
			BiCGSTAB	3 450 MiB	236 s
		BK	Uniformization	650 MiB	222 s
			BiCGSTAB	815 MiB	162 s
SR-Asym-7	10 775 710	Sparse	Uniformization	3 116 MiB	316 s
			BiCGSTAB	3 450 MiB	236 s
		BK	BiCGSTAB	812 MiB	373 s
SR-Degen-7	10 775 710	Sparse	BiCGSTAB	Breakdown	
		BK	Group GS / Jacobi	No convergence	
SR-Sym-9	81 466 099	Sparse	BiCGSTAB	25 564 MiB	2 542 s
SR-Asym-9	81 466 099	Sparse	BiCGSTAB	Oscillation	
		BK	Group GS / Jacobi	2 388 MiB	9 402 s
Cloud-3-2	20 047 500	Sparse	BiCGSTAB	Out of memory	
		BK	BiCGSTAB	Breakdown	
			Group GS / Jacobi	684 MiB	3 379 s
KanBan-5	2 546 432	Sparse	Uniformization	833 MiB	54 s
			BiCGSTAB	911 MiB	92 s
		BK	Uniformization	360 MiB	70 s
			BiCGSTAB	392 MiB	124 s
KanBan-7	41 644 800	Sparse	Uniformization	12 471 MiB	909 s
		BK	Uniformization	6 253 MiB	1 135 s

We can conclude that while block Kronecker generator matrices stored as expression trees are significantly more complicated than sparse storage, the configurable operations framework managed to perform the matrix multiplications efficiently for both small and large model state spaces.

For models with similar transition rates the BiCGSTAB algorithm was found the most effective steady-state solution method for both sparse and block Kronecker matrices.

Slower, but more memory efficient algorithms, such as Gauss-Seidel iteration and Jacobi iteration, often diverged with the sparse matrix form while converged using the block Kronecker form. This is due to different state orderings in the different matrix representations.

For irregular models with transition rates of orders of magnitude difference it is possible that BiCGSTAB will not converge while other algorithms yield a solution, albeit only after many iterations.

In transient analysis, orders of magnitude differences in transition rates lead to

highly *stiff* models. High stiffness increases the number of iterations performed by uniformization significantly. In Figure 6.4, the stiff *SR-Degen-3* model appears as an outlier with nearly 500 s execution time despite having only 26 464 states. Transient analysis of larger versions of the *SR-Degen* did not finish withing the time limit.

6.3 The convergence of IDRSTAB

To study the convergence properties of IDR(s)STAB(ℓ), we created several scaled versions of the *SharedResource-Sym* model by varying the the number of available shared resources, the number of clients and the number processes per client between 1 and 5. This allowed observing the behavior of the solver with a variety of matrix sizes. In all problems sparse matrix storage was selected due to the relatively small size of the state space.

IDR(s)STAB(ℓ) was executed on the steady-state linear equations arising from the generated stochastic models with parameter settings $s = 1, 2, 4, 8$ and $\ell = 1, 2, 4, 8$. Note that the case $s = \ell = 1$ is mathematically equivalent to BiCGSTAB, albeit with worse numerical properties in finite-precision arithmetic.

Due to the random initialization of the shadow Krylov subspace (see Algorithm 5.10 on page 57) each experiment was repeated with five different random seeds. Therefore 10 000 runs were performed in total.

The histogram of the observed behaviors of IDR(s)STAB(ℓ) is shown in Figure 6.6. Each histogram shows the observations for a particular (s, ℓ) parameter setting. As the cases $s = 4$ and $\ell = 4$ show performance very similar to $s = 8$ and $\ell = 8$, therefore they were omitted in the interest of reducing clutter.

- "Success" refers termination of the algorithm with a solution within 200 iterations.
- "Breakdown" refers to calculations that resulted in a singular σ matrix and therefore terminated.
- "Divergence" yielded residual norms increasing to the limit of double-precision floating point.
- "No result" cases failed to produce a solution in 200 iterations, but did not otherwise fail.

In addition, the first 50 iterations of trajectories from 5 625 short experiments are summarized in Figure 6.7. The trajectories are grouped by (s, ℓ) parameters and state space size.

It is apparent that only $s = 1$ cases managed to decrease residual norm and only $\ell = 1$ lead to reliable decrease of residual norm over iterations for larger state spaces.

For the other parameter settings, successful exit from the iteration happened only due to our modifications to IDR(s)STAB(ℓ) introduced in Algorithms 5.10 to 5.12 on pages 57–59. However, stability of convergence is still lacking, as our proposed modifications are unable to handle singular or nearly singular σ matrices that arise when IDR(s)STAB(ℓ) is applied to steady-state analysis of CTMC models.

Settings with $\ell > s$ tend to result in rapid divergence of the residual to infinity, while $\ell \leq s$ often breaks down or oscillates.

Surprisingly, breakdowns occur even if $s = \ell = 1$, i.e. if BiCGSTAB mode is used. Before breakdown, the residual norm is approximately $6 \cdot 10^{-7}$. This number is probably the limit to the accuracy of the IDR(s)STAB(ℓ) formulation of BiCGSTAB, as the original BiCGSTAB algorithm utilizes a more stable residual update strategy.

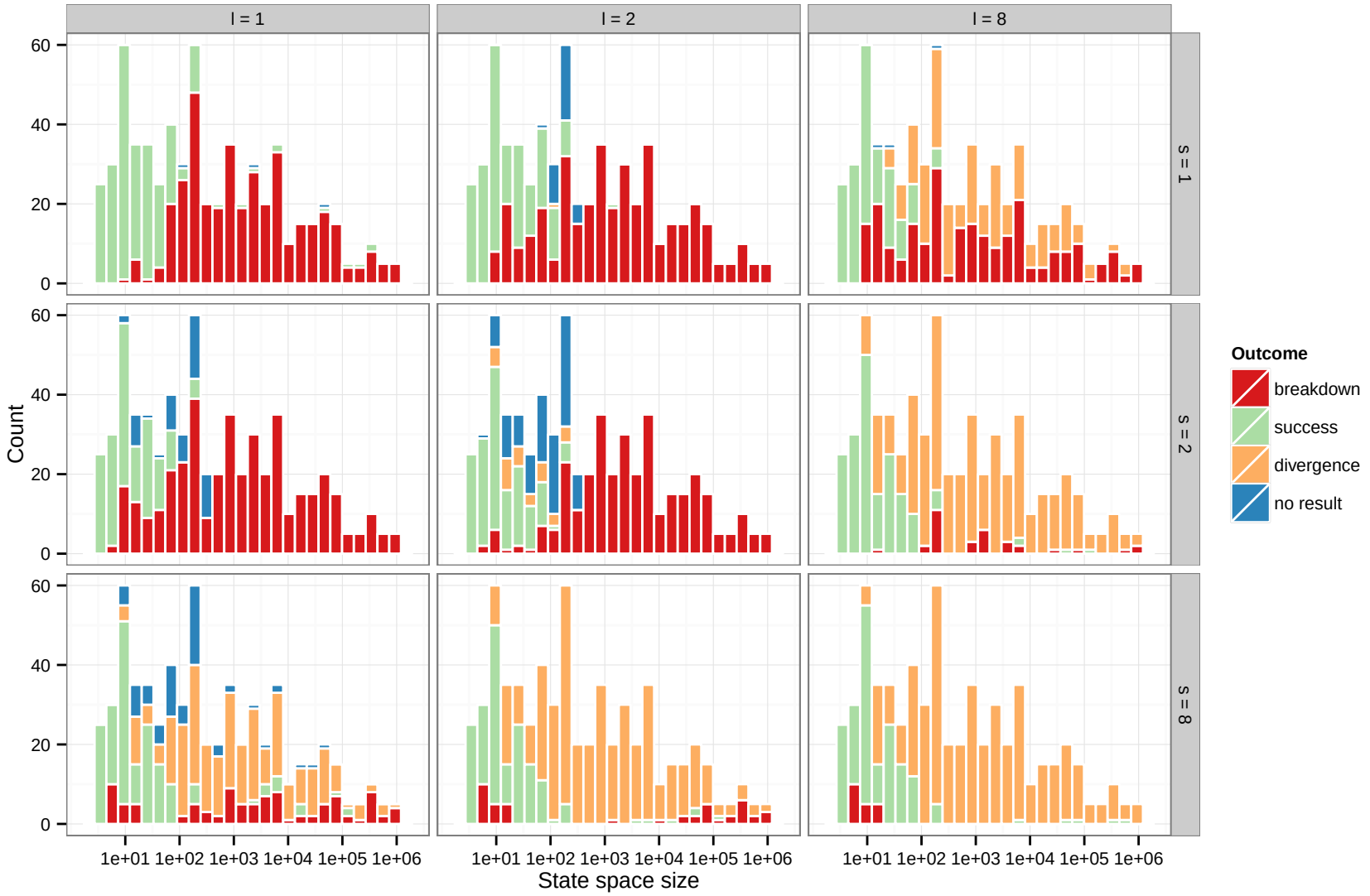


Figure 6.6 Histogram of observed behaviors of IDR(s)STAB(ℓ).

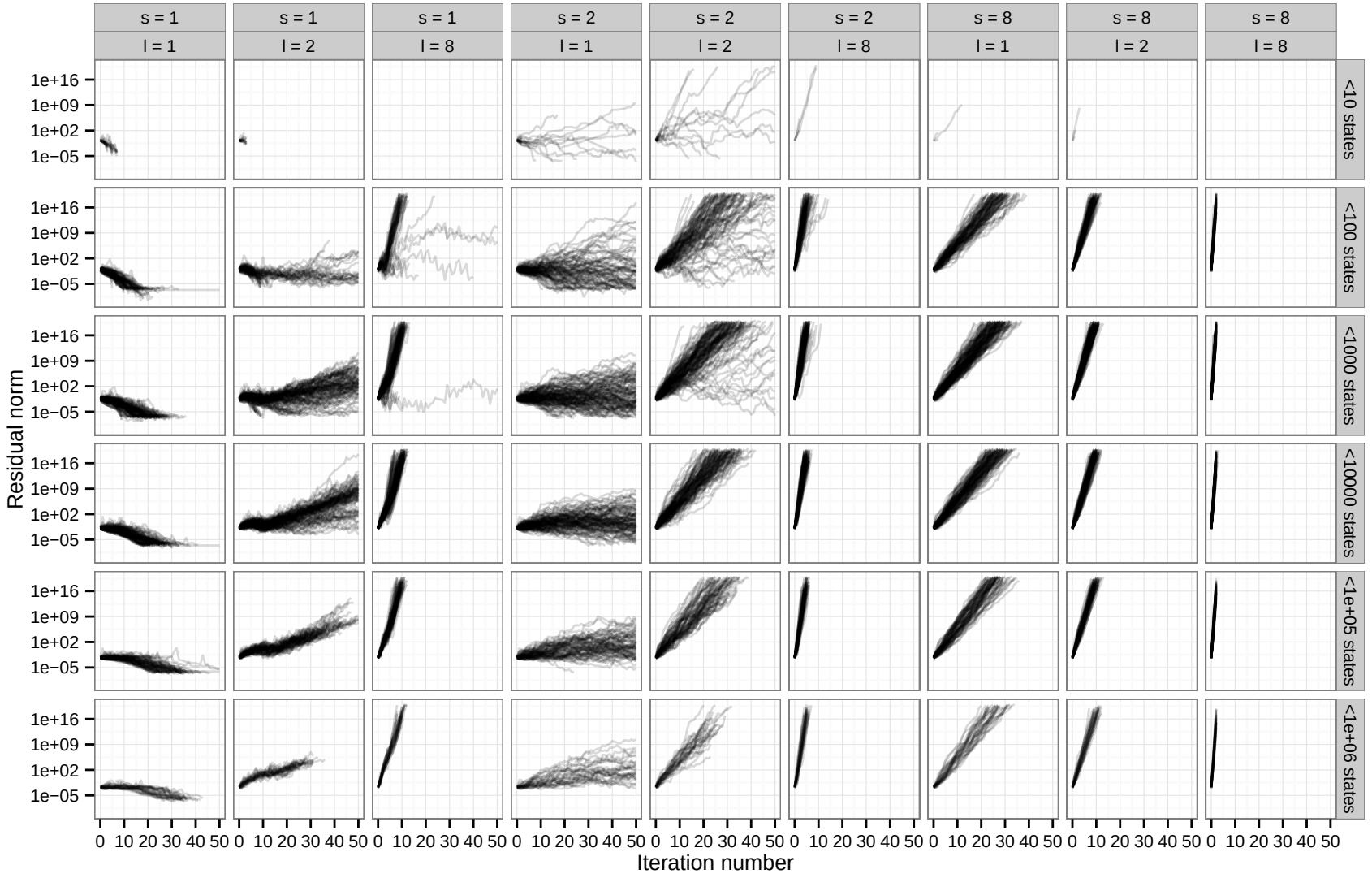


Figure 6.7 Convergence histories observed in various runs of IDR(s)STAB(ℓ).

Chapter 7

Conclusion and future work

We have developed and presented our numeric backend to the *configurable stochastic analysis framework* for the dependability, reliability and performability analysis of complex asynchronous systems. Our presented approach is able to combine the strength and advantages of the different algorithms and data structures into one framework. Various optimization techniques were used during the development and many of the algorithms are parallelized to exploit the advantages of modern mulitcore processor architectures.

From the theoretical side, we have obtain preliminary results in the adaptation of the state-of-the-art IDR(s)STAB(ℓ) algorithm to steady-state stochastic analysis tasks for integration into the framework.

In addition we have investigated the composability of the various data storage, numerical solution and infinitesimal generator matrix representation techniques and combined them together to provide configurable stochastic analysis in our framework.

Extensive investigation was executed in the field to be able to develop 3 generator matrix decomposition and representation techniques, 7 steady-state solvers, 2 transient analysis algorithms for the computation of engineering measures.

Our long term goal is to provide these analysis techniques also for a wider community, we have integrated our library into the PetriDotNet framework. Our algorithms are used also in the education for illustration purposes of the various stochastic analysis techniques. In addition, our tool was also used in an industrial project: one of our case-studies is based on that project. More than 70 000 generated test cases serve to ensure correctness as much as possible. In addition, software redundancy based testing was applied to further improve the quality of our library.

Despite our attempts to be as comprehensive as possible, many promising directions for future research and development are

- more extensive benchmarking of algorithms to extend the knowledge base about the effectiveness and behavior of stochastic analysis approaches toward and adaptive framework for stochastic analysis;

- completion of the work started on IDR(s)STAB(ℓ) by improving the stabilization part [77, 80] of the algorithm to attain convergence on a wide range of stochastic models and parameter settings;

- the implementation and development of further numerical algorithms, including those that can take advantage of the various decompositions of stochastic models [16, 17, 30];
- reduction of the size of Markov chains through the exploitation of model symmetries [14, 43];
- the development of preconditioners for the available iterative numerical solution methods [51];
- distributed implementations of the existing algorithms [21];
- support for fully symbolic storage and solution of Markov chains [26, 63, 87];
- the use of tensor decompositions instead of vectors to store state distributions and intermediate results to greatly reduce memory requirements of solution algorithms [3, 31, 38].

Acknowledgment We would like to thank Prof. Peter Buchholz for his helpful comments on Krylov subspace solution algorithms.

We would like to thank IncQueryLabs Ltd. for their support during the summer internship.

References

[1] Kensuke Aihara, Kuniyoshi Abe, and Emiko Ishiwata. “A variant of IDRstab with reliable update strategies for solving sparse linear systems”. In: *Journal of Computational and Applied Mathematics* 259 (2014), pp. 244–258.

[2] 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.

[3] Jonas Ballani and Lars Grasedyck. “A projection method to solve linear systems in tensor format”. In: *Numerical Linear Algebra with Applications* 20.1 (2013), pp. 27–43.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] Mehra N Borazjany, Linbin Yu, Yu Lei, Raghu Kacker, and Rick Kuhn. “Combinatorial testing of ACTS: A case study”. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE. 2012, pp. 591–600.

[13] Peter Buchholz. “A class of hierarchical queueing networks and their analysis”. In: *Queueing Syst.* 15.1-4 (1994), pp. 59–80. DOI: 10.1007/BF01189232.

[14] Peter Buchholz. “Exact and ordinary lumpability in finite Markov chains”. In: *Journal of applied probability* (1994), pp. 59–75.

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

[16] Peter Buchholz. “Multilevel solutions for structured Markov chains”. In: *SIAM Journal on Matrix Analysis and Applications* 22.2 (2000), pp. 342–357.

[17] Peter Buchholz. “Structured analysis approaches for large Markov chains”. In: *Applied Numerical Mathematics* 31.4 (1999), pp. 375–404.

[18] 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.

[19] 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.

[20] 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.

[21] Jaroslaw Bylina and Beata Bylina. “Merging Jacobi and Gauss-Seidel methods for solving Markov chains on computer clusters”. In: *Proceedings of the International Multiconference on Computer Science and Information Technology, IMCSIT 2008, Wisla, Poland, 20-22 October 2008*. IEEE, 2008, pp. 263–268. DOI: 10.1109/IMCSIT.2008.4747250.

[22] 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.

[23] Gianfranco Ciardo, Robert L Jones, Andrew S Miner, and Radu Siminiceanu. “Logical and stochastic modeling with SMART”. In: *Computer Performance Evaluation. Modelling Techniques and Tools*. Springer, 2003, pp. 78–97.

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

[25] 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.

[26] Gianfranco Ciardo and Andrew S. Miner. “Implicit data structures for logic and stochastic systems analysis”. In: *SIGMETRICS Performance Evaluation Review* 32.4 (2005), pp. 4–9. DOI: 10.1145/1059816.1059818.

[27] PJ Courtois and P Semal. “Block iterative algorithms for stochastic matrices”. In: *Linear algebra and its applications* 76 (1986), pp. 59–70.

[28] 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.

[29] 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.

[30] Tugrul Dayar. *Analyzing Markov chains using Kronecker products: theory and applications*. Springer Science & Business Media, 2012.

[31] Sergey V Dolgov. “TT-GMRES: solution to a linear system in the structured tensor format”. In: *Russian Journal of Numerical Analysis and Mathematical Modelling* 28.2 (2013), pp. 149–172.

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

[33] 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.

[34] 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.

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

[36] 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.

[37] Rahul Ghosh. “Scalable stochastic models for cloud services”. PhD thesis. Duke University, 2012.

[38] Lars Grasedyck, Daniel Kressner, and Christine Tobler. “A literature survey of low-rank tensor approximation techniques”. In: *arXiv preprint arXiv:1302.7121* (2013).

[39] 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.

[40] 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.

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

[42] Oleg Gusak, Tuğrul Dayar, and Jean-Michel Fourneau. “Lumpable continuous-time stochastic automata networks”. In: *European Journal of Operational Research* 148.2 (2003), pp. 436–451.

[43] Serge Haddad and Patrice Moreaux. “Evaluation of high level Petri nets by means of aggregation and decomposition”. In: *Petri Nets and Performance Models, 1995., Proceedings of the Sixth International Workshop on*. IEEE. 1995, pp. 11–20.

[44] Hans Hansson and Bengt Jonsson. “A Logic for Reasoning about Time and Reliability”. In: *Formal Asp. Comput.* 6.5 (1994), pp. 512–535. DOI: 10.1007/BF01211866.

[45] 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.

[46] 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.

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

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

[49] Attila Klenik and Kristóf Marussy. *Configurable Stochastic Analysis Framework for Asynchronous Systems*. 1st prize. 2015. URL: https://tdk.bme.hu/VIK/DownloadPaper/Aszinkron-rendszerek-konfigurarhato.

[50] 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.

[51] 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.

[52] 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.

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

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

[55] John F. Meyer, Ali Movaghar, and William H. Sanders. “Stochastic Activity Networks: Structure, Behavior, and Application”. In: *International Workshop on Timed Petri Nets, Torino, Italy, July 1-3, 1985*. IEEE Computer Society, 1985, pp. 106–115. ISBN: 0-8186-0674-6.

[56] Microsoft Research. *The Microsoft CodeContract webpage*. Accessed October 26, 2015. URL: http://research.microsoft.com/en-us/projects/contracts/.

[57] Microsoft Research. *The Microsoft IntelliTest webpage*. Accessed October 26, 2015. URL: http://research.microsoft.com/en-us/projects/pex/.

[58] Microsoft Research. *The Text Template Transformation Toolkit webpage*. Accessed October 26, 2015. URL: https://msdn.microsoft.com/en-us/library/bb126445(v=vs.120).aspx.

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

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

[61] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

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

[63] *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*. IEEE Computer Society, 2012. ISBN: 978-1-4673-2346-8. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6354262.

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

[65] Poole, Prouse, Busoli, Colvin, Popov. *The NUnit webpage*. Accessed October 26, 2015. URL: http://www.nunit.org/.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] Olaf Rendel and MZ Jens-Peter. "Tuning IDR to fit your applications". In: *Proceedings of a Workshop at Doshisha University*. 2011.

[72] 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.

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

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

[75] Valeria Simoncini and Daniel B Szyld. "Interpreting IDR as a Petrov–Galerkin method". In: *SIAM Journal on Scientific Computing* 32.4 (2010), pp. 1898–1912.

[76] Gerard LG Sleijpen and Diederik R Fokkema. “BiCGstab (l) for linear equations involving unsymmetric matrices with complex spectrum”. In: *Electronic Transactions on Numerical Analysis* 1.11 (1993), p. 2000.

[77] Gerard LG Sleijpen and Henk A Van der Vorst. “Maintaining convergence properties of BiCGstab methods in finite precision arithmetic”. In: *Numerical Algorithms* 10.2 (1995), pp. 203–223.

[78] Gerard LG Sleijpen and Martin B Van Gijzen. “Exploiting BiCGstab (ℓ) strategies to induce dimension reduction”. In: *SIAM journal on scientific computing* 32.5 (2010), pp. 2687–2709.

[79] 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.

[80] Peter Sonneveld. *On the convergence behaviour of IDR (s)*. Tech. rep. Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft Institute of Applied Mathematics, 2010.

[81] 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.

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

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

[84] Masaaki Tanio and Masaaki Sugihara. “GBi-CGSTAB (s, L): IDR (s) with higher-order stabilization polynomials”. In: *Journal of computational and applied mathematics* 235.3 (2010), pp. 765–784.

[85] Nikolai Tillmann and Jonathan De Halleux. “Pex–white box test generation for. net”. In: *Tests and Proofs*. Springer, 2008, pp. 134–153.

[86] 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.

[87] Yang Zhao and Gianfranco Ciardo. “A Two-Phase Gauss-Seidel Algorithm for the Stationary Solution of EVMDD-Encoded CTMCs”. In: *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*. IEEE Computer Society, 2012, pp. 74–83. DOI: 10.1109/QEST.2012.34.