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

Efficient Stochastic Analysis of Asynchronous Systems

Master's thesis

Written by
Attila Klenik

Supervisors
András Vörös
dr. Miklós Telek
Vince Molnár

2016

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Klenik Attila*, szigorló hallgató kijelentem, hogy ezt a diplomatervet 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 autentikált 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.

Budapest, 2016. június 7.

Klenik Attila
hallgató

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

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

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

A markovi analízishez szükséges a modell lehetséges viselkedéseinek, azaz állapotterének felderítése, illetve tárolása, mely szimbolikus módszerekkel hatékonyan végezhető el. Ezzel szemben a sztochasztikus algoritmusokban használt vektor- és indexműveletek szimbolikus megvalósítása nehézkes. Munkám célja egy olyan hatékony sztochasztikus analízis keretrendszer és algoritmusok fejlesztése, amelyek lehetővé teszik a komplex sztochasztikus rendszerek kezelését a szimbolikus módszerek és hatékony mátrix-reprezentációk előnyeinek ötvözésével, továbbá lehetőséget adnak kisebb rendszerek extra-funkcionális jellemzőinek tetszőleges pontosságú, zárt függvény alakban történő meghatározására.

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

A megvalósított algoritmusok lehetővé teszik a különböző mátrix és állapottér reprezentációk kombinált használatát. Az implementált numerikus algoritmusokkal tetszőleges pontosságú állandósult állapotbeli költség- és érzékenység analízis és első hiba várható bekövetkezési idő analízis végezhető el sztochasztikus Petri-háló (*SPN*) alapú markovi költségmodelleken. A számított metrikák meghatározására a numerikus módszereken kívül szimbolikus kiértékelést is támogat az elkészített keretrendszer, amely egy zárt függvényként állítja elő a kívánt metrikákat. A keretrendszert integráltuk a PetriDotNet modellező szoftverrel. Az új módszer gyakorlati alkalmazhatóságát szintetikus és ipari modelleken végzett mérésekkel igazoljuk.

Kulcsszavak aszinkron rendszerek, teljesítményvizsgálat, sztochasztikus modell, szimbolikus módszerek, szimbolikus kiértékelés, érzékenységvizsgálat

Abstract Ensuring the correctness of critical systems – such as safety-critical, distributed and cloud applications – requires the rigorous, mathematically precise analysis of the functional and extra-functional properties of the system. 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 prevent the success of stochastic analysis due to the high sensitivity to the number of possible behaviors. In addition, temporal characteristics of the components may also 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 is influenced by various characteristics of the models, therefore no single best approach is known.

The prerequisite of Markovian analysis is the exploration of the state space, i.e., the possible behaviors of the system. Symbolic approaches provide an efficient state space exploration and storage technique, however their application to support the vector operations and index manipulations extensively used by stochastic algorithms is cumbersome. The goal of our work is to present an efficient framework with algorithms facilitating the analysis of complex, stochastic systems by combining the advantages of symbolic algorithms and compact matrix representations, and providing means to compute extra-functional properties of smaller systems with arbitrary precision given in a closed function form.

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

The implemented algorithms provide means to combine the different matrix and state space representations. Various algorithms are implemented for arbitrary precision and/or symbolically evaluated steady-state reward and sensitivity analysis, transient reward analysis and mean-time-to-first-failure analysis of stochastic Petri net (*SPN*) based Markovian reward models. The framework 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, symbolic methods, symbolic evaluation, sensitivity analysis

Chapter 1

Introduction

The growing need for ensuring the correctness of critical systems – such as safety-critical, distributed and cloud applications – requires the rigorous and mathematically precise analysis of the functional and extra-functional properties. A large class of typical quantitative questions regarding dependability and performability are usually addressed with the help of 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 stochastic 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 report our goal is to propose a framework that facilitates solving various problems occurring in stochastic analysis of complex systems.

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

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

The quantitative analysis of systems can be prone to numerical errors due to the limitations of the floating point hardware arithmetic. These numerical errors usually originate from the presence of special system events in the model (e.g., failure events) whose rates are generally an order of magnitude lower (or higher) than the rest of the event rates in the model. To alleviate this limitation we integrated arbitrary precision software arithmetic into the stochastic analysis framework. Software arithmetic improves the precision of calculations by allocating more CPU time and memory to the manipulation and storage of numbers during the analysis. The overhead caused by this method is an acceptable trade-off when it comes to the calculation of critical measures, e.g., measures of fault models of critical systems, which require the best precision available.

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

- Various state space exploration techniques,
- With decomposition algorithms and representation techniques for the stochastic behaviour of the systems,
- Various arbitrary precision 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 first failure values,
- Symbolic evaluation of the aforementioned measures for smaller systems.

Several problems were solved during our work: an approach is introduced to transform the different state space representations of models into stochastic behavior descriptor matrices of various format. Algorithms are implemented for steady-state reward and sensitivity analysis, transient reward analysis and mean-time-to-first-failure analysis of stochastic models in the *Stochastic Petri Net* (*SPN*) Markov reward model formalism. These algorithms provide arbitrary precision numerical computation for said analysis types and even symbolic evaluation for most analysis types. Benchmarks and industrial case studies are used to evaluate the applicability of our approach.

This report extends the research conducted in our previous report [46] and publications [52, 57, 77] by the addition of arbitrary precision and symbolic arithmetic to the

analysis types available in the framework. The parts of the framework that are not in the scope of this work are covered in detail in [51].

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

The remainder of this work is structured as follows: Chapter 2 reviews the necessary background for the stochastic analysis of stochastic Petri nets. Chapter 3 presents the configurable stochastic analysis pipeline. Chapter 4 introduces the different state space exploration techniques and the framework's integration with the PetriDotNet tool and the available features. Chapter 5 describes the decompositions of stochastic behaviors, including the hierarchical decomposition algorithm for symbolic state spaces in Section 5.2.3. Chapter 6 presents numerical steady-state and transient analysis algorithms and their implementations in our framework. Chapter 7 overviews the different methods for post-processing the numerical result that represents the state of the system at a given time. Chapter 8 details the integration of symbolic evaluation into our framework and the challenges it poses. After describing the testing and validation methodologies applied to our framework in Chapter 9 as well as the benchmark results, we conclude our paper in Chapter 10.

Chapter 2

Background

In this section we overview the basic formalisms and scope of our work. At first, Petri net based formalisms are introduced, which is the modelling language supported by our framework. Then the basic stochastic modelling background is discussed together with Kronecker algebra to give a base for the later sections. This chapter heavily uses the definitions and notations from our previous report [46] and other related works [58, 74].

2.1 Petri net

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

Definition 2.1 A *Petri net* is a 5-tuple $PN = (P, T, F, W, M_0)$, where:

- $P = \{p_0, p_1, \ldots, p_{n-1}\}$ is a finite, nonempty set of places;
- $T = \{t_0, t_1, \ldots, t_{m-1}\}$ is a finite, nonempty set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, also called the flow relation;
- $W : F \to \mathbb{N}^+$ is an arc weight function;
- $M_0 : P \to \mathbb{N}$ is the initial marking;
- $P \cap T = \emptyset$ [58].

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

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

Petri nets are graphically represented as directed bipartite graphs with arc weights.

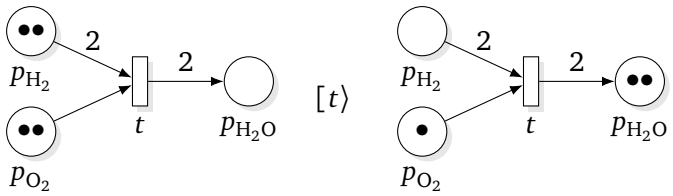


Figure 2.1 A Petri net model of the reaction of hydrogen and oxygen.

Places are drawn as circles, while transitions are drawn as bars or rectangles. Arc weights of 1 are usually omitted from presentation for sake of clarity. Dots (or a number) inside places correspond to tokens in the current marking.

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

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

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

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

$$M = M_1\,[t_{i_1}\rangle\,M_2\,[t_{i_2}\rangle\,M_3\,[t_{i_3}\rangle \cdots [t_{i_{k-2}}\rangle\,M_{k-1}\,[t_{i_{k-1}}\rangle\,M_k = M'.$$

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

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

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

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

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

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

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

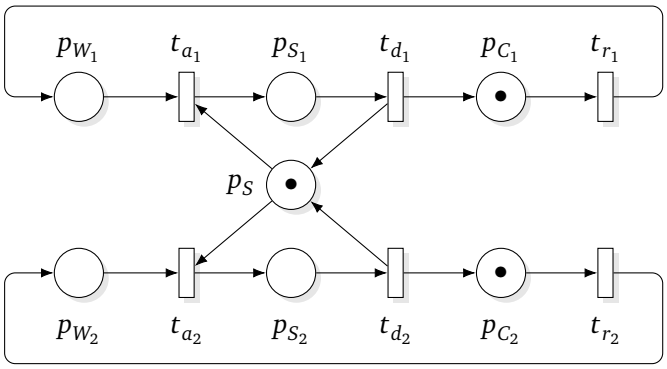


Figure 2.2 The *SharedResource* Petri net model.

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

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

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

2.1.1 Petri nets extended with inhibitor arcs

Inhibitor arcs are widely used extensions of Petri nets that can disable transitions even when the firing rule defined in Section 2.1 is satisfied. This modification gives Petri nets expressive power equivalent to Turing machines [18].

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

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

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

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

The firing rule (2.1) remains unchanged in Petri nets with inhibitor arcs.

2.2 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 2.4 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:

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

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

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

i.e., it is invariant to time shifting.

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

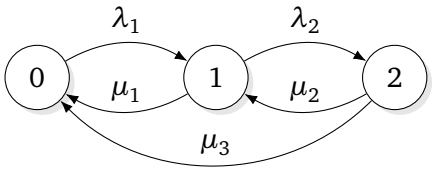
$$\pi(t)[x] = \mathbb{P}(X(t) = x)$$

and this vector satisfies the differential equation

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

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

- The diagonal elements $q[x, x] \leq 0$ 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*.



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

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

- The off-diagonal elements $q[x, y] \geq 0$ describe the state transitions. In state x the CTMC will jump to state y at the next state transition with probability $-q[x, y]/q[x, x]$. Equivalently, there is an 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 destination 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 [74].

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 $x, y \in S$, 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

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

Example 2.3 Figure 2.3 shows a CTMC with 3 states. The transitions from state 0 to 1 and from state 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, respectively. The transition from state 2 to 0 is also possible with rate μ_3.

The rows (corresponding to source states) and columns (destination states) of the infinitesimal generator matrix Q are labeled with the state numbers. The diagonal element $q[1, 1]$ is $-\lambda_2 - \mu_1$, hence the holding time in state 1 is exponentially distributed with rate $\lambda_2 + \mu_1$. The transition 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 from state 1 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 exists a unique steady-state distribution $\boldsymbol{\pi}$ independent from the initial distribution $\boldsymbol{\pi}_0$.

2.2.1 Markov reward models

Continuous-time Markov chains may be employed in the estimation of performance measures of models by defining *rewards* that associate *reward rates* with the states of a CTMC. The 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 *accumulated reward* random variables may correspond to performance measures associated with intervals of time, such as total downtime. Accumulated reward random variables are not in the scope of this work. For details we refer to [51] and [70].

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

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

The computation of the distribution function of $R(t)$ is a computationally intensive task (a summary is available at [68, Table 1]), while its mean value, $\mathbb{E}R(t)$, can be computed efficiently as discussed below.

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

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

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

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

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

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

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

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

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

2.2.2 Sensitivity

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

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

$$\frac{\partial\, \mathbb{E}R(t)}{\partial\, \theta_i}.$$

The model reacts more prominently to the changes of parameters with high absolute sensitivity, 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. (2.4) is taken, yielding

$$\frac{\partial\, \mathbb{E}R(t)}{\partial\, \theta_i} = \frac{\partial \boldsymbol{\pi}(t)}{\partial\, \theta_i}\mathbf{r}^{\mathrm{T}} + \boldsymbol{\pi}(t)\left(\frac{\partial \mathbf{r}}{\partial\, \theta_i}\right)^{\mathrm{T}} = \mathbf{s}_i(t)\,\mathbf{r}^{\mathrm{T}} + \boldsymbol{\pi}(t)\left(\frac{\partial \mathbf{r}}{\partial\, \theta_i}\right)^{\mathrm{T}}, \tag{2.6}$$

where $\mathbf{s}_i$ is the sensitivity of $\boldsymbol{\pi}$ to the parameter θ_i.

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

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

where $V_i = \partial Q(\boldsymbol{\theta})/\partial\, \theta_i$ is the partial derivative of the generator matrix [69].

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

$$\mathbf{s}_i Q = -\boldsymbol{\pi} V_i, \quad \mathbf{s}_i \mathbf{1}^T = 0 \tag{2.7}$$

is solved [7].

Another type of sensitivity analysis considers *unstructured* small perturbations of the infinitesimal generator matrix Q instead of dependencies on parameters [35, 43]. This latter, unstructured analysis may be used to study the numerical stability and conditioning of the solutions of the Markov chain.

2.2.3 Time to first failure

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

Let $D \subsetneq S$ be a set of *failure states* of the CTMC $X(t)$ and $U = S \setminus D$ be a set of operational states. We will assume without loss of generality that $U = \{0, 1, \ldots, n_U - 1\}$ and $D = \{n_U, n_U + 1, \ldots, n - 1\}$. Using this state ordering we can write the infinitesimal generator Q in the following form:

$$Q = \begin{pmatrix} Q_{UU} & Q_{UD} \\ Q_{DU} & Q_{DD} \end{pmatrix},$$

where submatrices Q_{UU}, Q_{UD}, Q_{DU} and Q_{DD} represent the transitions between operational states, from operational to failure states, from failure states to operational states and between failure states, respectively.

The matrix

$$Q_{Ud} = \begin{pmatrix} Q_{UU} & \mathbf{q}_{Ud} \\ \mathbf{0} & 0 \end{pmatrix}$$

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 *column* vector $\mathbf{q}_{Ud} \in \mathbb{R}^{n_U}$ is defined as

$$\mathbf{q}_{Ud} = Q_{UD} \mathbf{1}^{\mathrm{T}}.$$

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 $(\boldsymbol{\pi}_U, Q_{UU})$ [60], 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:

$$MTFF = \mathbb{E}[TFF] = -\boldsymbol{\pi}_U Q_{UU}^{-1} \mathbf{1}^{\mathrm{T}}. \tag{2.8}$$

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

$$\mathbb{P}(X(TFF_{+0}) \in D') = -\boldsymbol{\pi}_U Q_{UU}^{-1} \mathbf{q}_{UD'}^{\mathrm{T}}, \tag{2.9}$$

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

2.3 Stochastic Petri nets

While reward processes based on continuous-time Markov chains allow the study of dependability or reliability, the explicit specification of stochastic processes and rewards is often cumbersome. More expressive formalisms include queueing networks, stochastic process algebras such as PEPA [29, 37], Stochastic Automata Networks [32] and Stochastic Petri Nets (SPN).

Stochastic Petri Nets extend Petri nets by assigning exponentially distributed random delays to transitions [48]. After the delay associated with an enabled transition is elapsed the transition fires *atomically* and transition delays are reset.

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

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

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

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

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

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

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

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

   The initial distribution vector will be set to

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

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

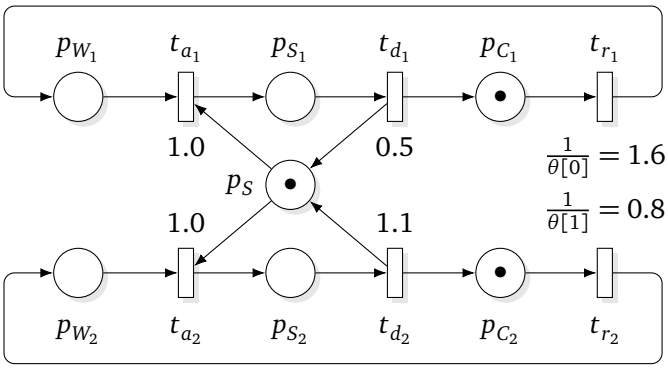


Figure 2.4 Example stochastic Petri net for the *SharedResource* model.

$$RS = \left\{ \begin{array}{|l|cccccccl|} \hline P\text{:} & S & C_1 & W_1 & S_1 & C_2 & W_2 & S_2 & \\ \hline M_0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & \text{initial} \\ M_1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \text{client 1 waiting} \\ M_2 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & \text{client 2 waiting} \\ M_3 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & \text{1 waiting, 2 waiting} \\ M_4 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & \text{client 1 shared working} \\ M_5 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \text{1 shared working, 2 waiting} \\ M_6 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \text{client 2 shared working} \\ M_7 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \text{1 waiting, 2 shared working} \\ \hline \end{array} \right\}$$

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

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

$$\begin{aligned} q_O[x,y] &= \sum_{\substack{t \in T \\ M_x[t\rangle M_y}} \Lambda(t) \quad \text{if } x \neq y, \\ q_O[x,x] &= 0, \\ Q &= Q_O + Q_D\}, \end{aligned} \tag{2.10}$$

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

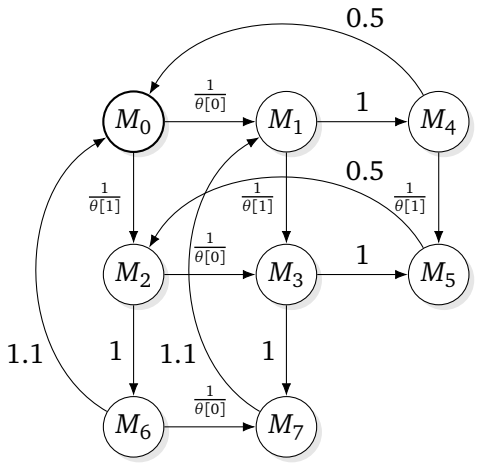


Figure 2.5 The CTMC associated with the *SharedResource* SPN model.

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

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

The reachable state space (Table 2.1) contains 8 markings which are mapped to the integers $S = \{0, 1, \ldots, 7\}$. The state space graph along with the transition rates of the CTMC is shown in Figure 2.5. The generator matrix is (also depicting state indices):

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

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

Extensions of stochastic Petri nets include transitions with general or phase-type delay distributions [47, 49], Generalized Stochastic Petri Nets (GSPN) with immediate transitions [50, 75] and Deterministic Stochastic Petri Nets (DSPN) with deterministic firing delays [72]. Among these, only phase-type distributed delays and GSPNs can be handled with purely Markovian analysis. Stochastic Well-formed Nets (SWN) are a class of colored Petri nets especially amenable to stochastic analysis [17]. Stochastic Activity Networks (SAN) also allow colored places, moreover, they introduce input and output gates for more flexible modeling [42].

2.3.1 Stochastic reward nets

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

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

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

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

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

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

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

where the summation is taken over all enabled transitions [23].

Running example 2.6 The SRN model

$$rr_1(M) = M(p_{S_1}) + M(p_{S_2}), \quad ir_1(t, M) \equiv 0 \tag{2.11}$$

describes the utilization of the shared resouce in the *SharedResouce* SPN (Figure 2.4 on page 14). $R_1(t) = 1$ if the resource is allocated, hence $\mathbb{E}R_1(t)$ is the probability that the resource is in use at time t.

Another reward structure

$$rr_2(M) \equiv 0, \quad ir_2(t,M) = \begin{cases} 1 & \text{if } t \in \{t_{r_1}, t_{r_2}\}, \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

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

The reward vectors associated with these SRNs are

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

2.3.2 Superposed stochastic Petri nets

In this section we define the base formalism of the decomposition algorithm introduced in Chapter 5.

Definition 2.9 A *Superposed Stochastic Petri Net* (SSPN) is a pair $SSPN = (SPN, \mathscr{P})$, where $\mathscr{P} = \{P^{(0)}, P^{(1)}, \ldots, P^{(J-1)}\}$ is the partitioning of the set of places and $P = P^{(0)} \cup P^{(1)} \cup \cdots \cup P^{(J-1)}$ [28]. Superposed stochastic Petri nets with inhibitor arcs $SSPN_I = (SPN_I, \mathscr{P})$ are defined analogously.

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

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

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

  $$T_L^{(j)} = \{t \in T : {}^\bullet t \cup t^\bullet \subseteq P^{(j)}\}. \tag{2.13}$$

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

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

  $$T_S^{(j)} = \{t \in T : ({}^\bullet t \cup t^\bullet) \cap P^{(j)} \neq \emptyset\} \setminus T_L^{(j)}. \tag{2.14}$$

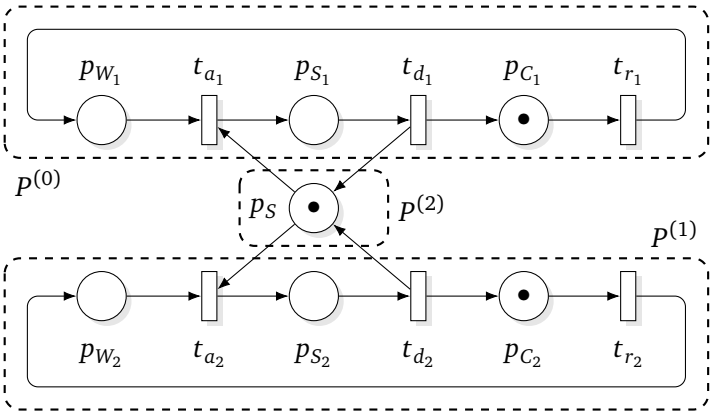


Figure 2.6 A partitioning of the *SharedResource* Petri net.

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

If there are inhibitor arcs in $SSPN_I$, inhibitor arcs must be considered when a local net $LN_I^{(j)}$ is constructed. The set ${}^\bullet t \cup t^\bullet$ is replaced with ${}^\bullet t \cup t^\bullet \cup {}^\circ t$ in eqs. (2.13) and (2.14) so that the enablement of local transitions only depends on the marking of places in $P^{(j)}$ and only places in $P^{(j)}$ may be affected upon firing. In addition, the inhibitor arc relation and weight function are restricted as $I^{(j)} = I \cap (P^{(j)} \cap T^{(j)})$, $W_I^{(j)} = W_I|_{I^{(j)}}$.

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

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

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

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

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

This is a *subset* of the reachable state space of $LN^{(j)}$, in particular, $RS^{(j)}$ is always finite if RS is finite, even if $LN^{(j)}$ is not bounded. Analysis techniques for generating local

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

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

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

state spaces include *partial P-invariants* [13] and explicit projection of global reachable markings [10].

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

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

which is a (generally proper) superset of the global reachable state space RS.

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

$$M = \mathbf{x} = (x^{(0)}, x^{(1)}, \ldots, x^{(J-1)}) \tag{2.15}$$

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

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

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

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

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

The potential state space PS contains $3 \cdot 3 \cdot 2 = 18$ potential markings, although only 8 are reachable (Table 2.1 on page 14). For example, the marking $(2, 2, 0)$ is not reachable, and it would violate mutual exclusion.

2.4 Kronecker algebra

Kronecker algebra defines the building blocks of the decomposition algorithm being introduced in Chapter 5. With its help we can represent a large matrix as the function of smaller matrices, without explicitly generating the large matrix (and storing it in memory for example).

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

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

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

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

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

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

Example 2.9 Consider the matrices

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

Their Kronecker product is

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

while their Kronecker sum is

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

Chapter 3

Overview of the approach

In this chapter we provide a brief overview about the general workflow of stochastic analysis, the arising challenges during the workflow and our approach to address these challenges.

3.1 General workflow

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

1. *State space exploration.* The reachable state space *RS* of the higher level model, for example stochastic automata network or stochastic Petri net, is explored to enumerate the possible behaviors of the model. 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 local states.

   If the set of reachable states is infinite, only special algorithms, e.g., matrix geometric methods [41], may be employed later in the workflow. In this work we focus our attention to finite state spaces.

2. *Descriptor generation.* The infinitesimal generator matrix Q of the Markov chain $X(t)$ defined over *RS* is built. If the analyzed formalism is a Markov chain, Q is readily given. Otherwise, this matrix contains the transition rates between

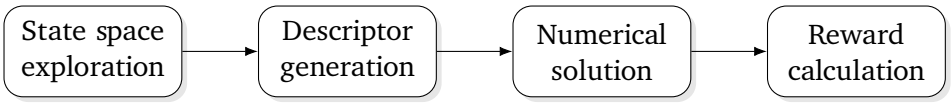


Figure 3.1 The general stochastic analysis workflow.

reachable states, which are obtained by evaluating rate expressions given in the model.

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

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 sensitivity of the rewards. Additional algebraic manipulations (for example, the calculation of the ratio of two instantaneous rewards) may be provided to the modeler for convenience.

In stochastic model checking, where the desired system behaviors are expressed with the help of stochastic temporal logics [1, 6], these analytic steps are called as subrouties to evaluate propositions. In the synthesis and optimization of stochastic models [16], the workflow is executed as part of the fitness function computation.

3.1.1 Challenges

The implementation of the stochastic analysis workflow poses several challenges.

State space size. It is difficult to handle large models due to the phenomenon of "state space explosion". As the size of the model grows (for example with the addition of more components) the number of reachable states can grow exponentially. Methods such as the *saturation* algorithm [21] 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.

Convergence properties. 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 that is most suitable for the properties of the model, as well as the decomposition method and hardware environment (like architectures with multiple physical cores).

Numerical precision. Sometimes it is preferable (or even necessary) to employ higher precision arithmetic calculations during the analysis. This could result in improved convergence properties of algorithms and reduced numerical errors that would otherwise

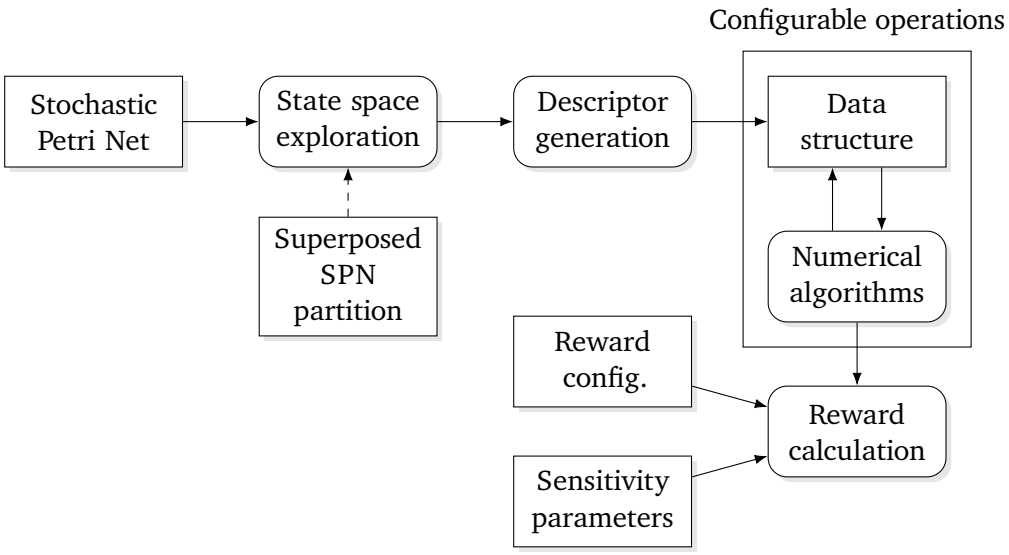


Figure 3.2 Configurable stochastic analysis workflow.

arise due to the limitations of the floating point hardware arithmetic. The execution time and memory usage of the analysis workflow may increase greatly depending on the used arithmetic precision. This further constrains the set of applicable algorithms during the analysis and has to be taken into account when configuring the workflow.

Scalability. 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

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

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

- The model state space may be explored either by an explicit state space traversal, or by symbolic saturation [21]. As symbolic methods are usually much faster and use

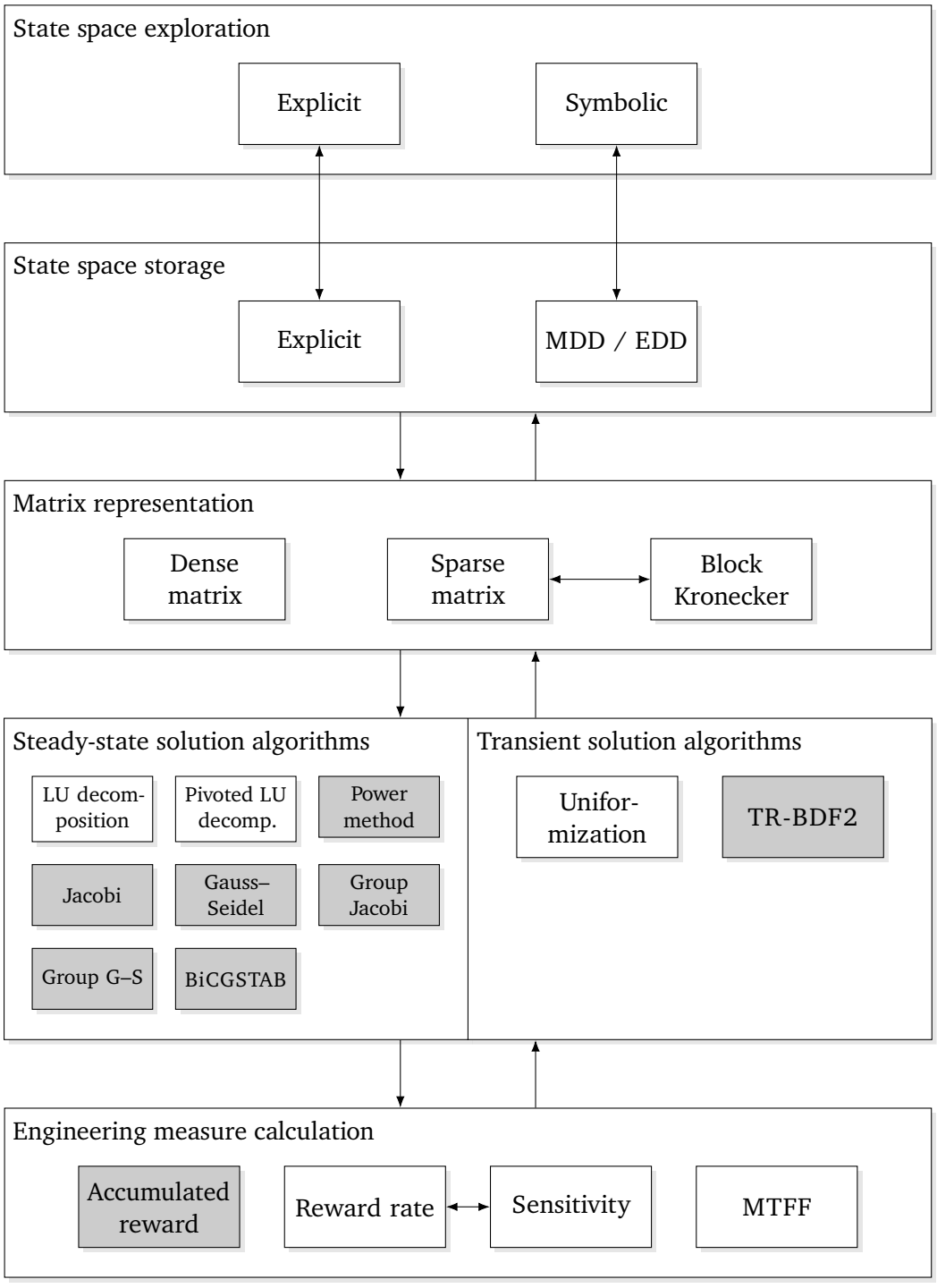


Figure 3.3 Architecture of the configurable stochastic analysis framework.

significantly less memory than explicit enumeration, they are the recommended approach for stochastic analysis. However, the explicit algorithms are less sensitive to the structure of the model, so they provide a robust solution as long as the state space fits into memory. In addition, they are implemented for benchmarking and software redundancy reasons, too.

The algorithms operating on a superposed SPN receive the model and a partitioning as an input. Partitions needed for the decomposition may be provided by the user as part of the model or generated on the fly. The given partitioning can greatly affect the performance of some algorithms like symbolic state space exploration or the properties of the resulting hierarchical stochastic descriptor matrix.

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

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

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

  State space exploration and generator matrix decomposition methods are presented in Chapters 4 and 5, respectively, including our theoretical and algorithmic contribution for block Kronecker decomposition.

- Several numerical algorithms are provided for steady-state and transient analysis of Markov chains. The user can select the algorithm most suitable for the model under study. The algorithm library supports the combination of the algorithms and data structures at different levels of computations. This allows us to fine-tune the numerical solution and solve every component with the most suitable algorithm. The algorithms with grey background in Figure 3.3 are not in the scope of this work. For details we refer to [51].

  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 selected solver algorithm in order to avoid excessive memory consumption.

Numerical algorithms that are in the scope of this work are discussed in Chapter 6.

3.2.1 Formalisms

Our stochastic analysis framework supports models in the Stochastic Petri Net with inhibitor arcs formalism (see Definition 2.7 on page 13). Structured models are handled as Superposed Stochastic Petri Nets (see Definition 2.9 on page 17). However, any modeling formalism can be processed by integrating the appropriate state space exploration algorithm with the workflow.

Transition rates in the SPNs can be arbitrary algebraic expressions (detailed in Chapter 8) containing references to *sensitivity variables*. These variables correspond to the parameter vector $\boldsymbol{\theta}$ of the Markov chain sensitivity analysis. However, currently the rate expression may not depend on the marking of the net.

Reward structures are defined as Stochastic Reward Nets (see Definition 2.8 on page 16). An SRN reward structure may be specified by composing any *reward expressions* in the forms detailed in Chapter 7.

3.2.2 Analysis

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

- expected steady-state reward rates $\mathbb{E}R$ for any reward structure defined by reward expressions,
- expected transient reward rates $\mathbb{E}R(t)$ and accumulated rewards (detailed in [51]),
- *composite rewards*, which are algebraic expressions composed of previously calculated reward rates (e.g., $1 + \mathbb{E}R_1(t)/\mathbb{E}R_2(t)$),
- sensitivity of mean steady-state reward rates and composite rewards involving steady-state rates,
- mean-time to failure *MTFF* and associated failure mode probabilities,
- calculating the aforementioned measures with arbitrary precision, and
- calculating the aforementioned measures in closed function form for smaller systems, except for transient reward rates and accumulated rewards.

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

Chapter 4

State space exploration

In order to perform qualitative or quantitative analyses on Petri net models, first we need to explore the possible behaviors, i.e. , the state space of the modelled system. In this chapter we provide a brief overview of state space exploration methods, which is the first step in our stochastic analysis workflow. Besides state space exploration we also cover the changes we made to the PetriDotNet tool in order to prepare it for performing stochastic analysis of SPN models.

4.1 Explicit state space exploration

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

The pseudocode of the implemented algorithm for explicit state space exploration is shown in Algorithm 4.1. Executing it on the *SharedResource* Petri net example model yields the reachable state space in Table 4.1, previously presented in Section 2.1.

Advantages Using explicit state space exploration and storage techniques has many advantages. The simplicity of the algorithm makes it easier to understand and maintain (and in case of errors, debug) the implementation. Due to the explicit (i.e complete) storage of markings the firing function at line 5 can be implemented to handle Petri net transitions even with complex properties like priority and/or guard condition.

Disadvantages In spite of the simplicity of the algorithm some parts of it can quickly became a bottleneck while trying to explore the state spaces of larger models. The

Algorithm 4.1 Explicit state space exploration.

Input: transitions T, initial marking M_0
Output: explicit state space RS

```
1  allocate set RS = {M0}, FIFO queue Q = {M0}
2  while Q ≠ ∅ do
3      M = dequeue(Q)
4      foreach t ∈ T, where M[t⟩ do
5          M* = fire(M, t)
6          if M* ∈ RS then
7              continue
8          RS ← RS ∪ {M*}
9          enqueue(Q, M*)
10 return RS
```

$$RS = \left\{ \begin{array}{|c|ccccccc|l|} \hline P: & S & C_1 & W_1 & S_1 & C_2 & W_2 & S_2 & \\ \hline M_0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & \text{initial} \\ M_1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \text{client 1 waiting} \\ M_2 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & \text{client 2 waiting} \\ M_3 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & \text{1 waiting, 2 waiting} \\ M_4 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & \text{client 1 shared working} \\ M_5 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \text{1 shared working, 2 waiting} \\ M_6 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & \text{client 2 shared working} \\ M_7 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & \text{1 waiting, 2 shared working} \\ \hline \end{array} \right\}$$

Table 4.1 Reachable state space of the *SharedResource* model.

firing function at line 5 creates the marking M^* that was reached by firing transition t in marking M. Storing the marking of every place in every state of a complex system leads to high memory usage. The other bottleneck manifests itself at line 6 where we check whether the acquired marking M^* was already explored (part of RS) or not. Even by using hash-like storage schemes for RS we still need to calculate, store and look up hash values among a large number of markings which can severely impact the performance of the algorithm.

4.2 Symbolic state space exploration

To alleviate the memory and performance limitations of explicit state space exploration, more efficient, albeit more complex methods were introduced. In this section we provide a brief background for symbolic methods that use decision diagrams to efficiently enumerate and encode the reachable state space of a system.

4.2.1 Multivalued decision diagrams

Multivalued decision diagrams (MDDs) [21] provide a compact, graph-based representation for functions of the form $\mathbb{N}^J \to \{0,1\}$.

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

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

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

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

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

4.2.2 Symbolic state spaces

Symbolic state space exploration has great advantages compared to explicit techniques but also raises some issues that didn't arise with explicit state space exploration.

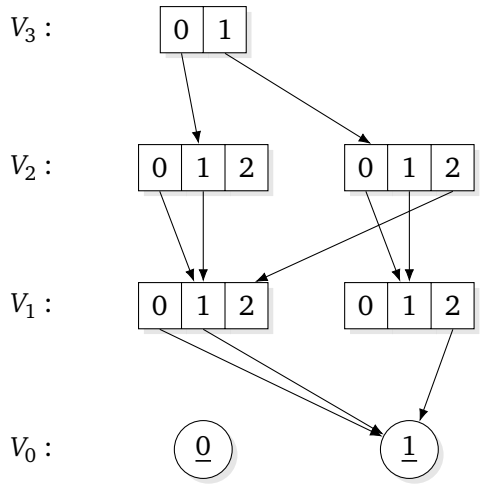


Figure 4.1 MDD state space encoding for the *SharedResource* SSPN.

Advantages Symbolic techniques involving MDDs can efficiently store large reachable state spaces of superposed Petri nets. Reachable states $x \in RS$ are associated with state codings $\mathbf{x} = (x^{(0)}, x^{(1)}, \ldots, x^{(J-1)})$. The function $f : PS \to \{0,1\}$ can be stored as an MDD where $f(\mathbf{x}) = 1$ if and only if $x \in RS$. The domains of the MDD are the local state spaces $D^{(j)} = RS^{(j)}$.

> **Running example 4.1** Figure 4.1 shows the state space of the *SharedResource* model encoded as an MDD. The paths from the root node to the terminal one node $\underline{1}$ along the arcs of the MDD represent the reachable local state combinations of the system. Arcs to the terminal zero node $\underline{0}$ were omitted for the sake of simplicity.

Examples of iteration strategies for MDD state space exploration include *breath-first search* and *saturation* [21]. We use the implementation of saturation from the PetriDotNet framework [26, 31].

Disadvantages Efficiency of symbolic techniques is achieved by using complex storage schemes and iteration strategies. This results in a more complex implementation which impacts maintainability and makes debugging harder. Furthermore it isn't trivial anymore to incorporate complex transition firing rules into the exploration algorithm because of the decomposed information about the state space. Handling immediate transitions (i.e., priority) for example is necessary to support the GSPN formalism, however, it also raises new problems, like how to eliminate vanishing markings from

the reachable state space [24].

4.3 PetriDotNet integration

The introduced state space exploration algorithms are defined for regular Petri nets and don't use the additional transition rate information provided by stochastic Petri nets. In order to map *RS* to a CTMC the PetriDotNet framework had to be extended with additional capabilities to support and facilitate the analysis of SPNs.

In this section we describe the changes made to the PetriDotNet framework in order to successfully integrate our stochastic analysis framework with it. The exact feature set of the PetriDotNet framework is detailed in its official manual.[1]

Net parameters A general requirement for modeling tools is to support the usage of model parameters (where it is meaningful). This feature can facilitate the quick modification of the model's behavior by providing a centralized access for some properties of the model. This also increases the maintainability of the model and lowers the probability of modeling errors when introducing changes to the model properties. In case of Petri net models we can introduce parameters for multiple purposes, such as initial token numbers on places, arc weights and transition properties.

As part of this thesis we extended PetriDotNet in order to support net parameters that can be used in the definition of transition rates. Unlike token number and arc weight parameters, transition rate parameters have other purposes beside facilitating modeling – they are the basis for sensitivity analysis of computed performance metrics as described in Section 2.2.2. To define a parameter we need to provide a name and a default (floating point) value for it with an optional description. The purpose of default values is described in the next paragraphs. The exact syntax of parameter arithmetic expressions is detailed in Section 8.1

Net parameter configurations Even though net parameters provide a convenient way to modify the model's behavior in a centralized way, sometimes we want to change more than one parameter to express a different operational mode of the model (e.g., increased load of requests and/or increased failure rate of components due to degradation). In the extended PetriDotNet framework we also provide support for defining parameter configurations that can assign a (usually different) value to the defined parameters.

With the help of this feature we can define different parameter configurations for the different operational modes of the model without modifying the parameter definitions themselves. This way the modeler can easily switch between the configurations to achieve the desired behavior of the model. The stochastic analysis framework will

[1] `https://inf.mit.bme.hu/en/research/tools/petridotnet`

always consider the currently set configuration if there is one. Using the parameter configuration feature is optional, so if no configuration is defined then the parameters will be considered with their default values.

Transition rate expressions The PETRIDOTNET framework already supported different timing properties of transitions for the purpose of simulation, however, this wasn't sufficient for sensitivity analysis which depends on model (more precisely, transition rate) parameters. Thus we extended the transition data structure with the capability of storing an arithmetic expression tree that denotes the transition rate, possibly with the help of net parameters. The value of the already existing transition rate property is synchronized with this expression tree based on the current parameter configuration so the simulation capability of the tool is not affected by this change.

Additional Petri net data The newly introduced extensions, like net parameters, parameter configurations, transition rate trees and measure definitions (detailed in Chapter 7) are integral part of the model so we need to serialize them with the rest of the Petri net. The main serialization format of PETRIDOTNET is the Petri Net Markup Language standard (PNML) [78], which provides means to store additional information (not closely related to Petri nets) by defining a tool specific information part in its XML schema. We used this method to store the additional data so other tools can still process the resulting PNML file without considering these data.

Chapter 5

Efficient generation and storage of continuous-time Markov chains

5.1 Explicit methods

5.1.1 Explicit matrix construction

Given the finite state space of size $n = |RS|$ in an explicit form (for example acquired by executing Algorithm 4.1 on page 30) along with a bijection between the markings and the natural numbers $\{0, 1, \ldots, n-1\}$, the generator matrix Q can be directly created by Algorithm 5.1. The algorithm stores the transition rate $\Lambda(t)$ in Q for all pairs of reachable markings $M_x \, [t\rangle \, M_y$ and transitions $t \in T$.

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

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

5.1.2 Block Kronecker generator matrices

Kronecker generator matrices

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

Algorithm 5.1 Generator matrix construction from explicit state space.

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

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

associated CTMC in the form

$$Q = Q_O + Q_D, \quad Q_O = \bigoplus_{j=0}^{J-1} Q_L^{(j)} + \sum_{t \in T_S} \Lambda(t) \bigotimes_{j=0}^{J-1} Q_t^{(j)}, \quad Q_D = -\operatorname{diag}\{Q_O \mathbf{1}^{\mathrm{T}}\}, \tag{5.1}$$

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

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

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

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

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

It can be seen that

$$\begin{aligned} q_O[\mathbf{x}, \mathbf{y}] &= \sum_{j=0}^{J-1} \sum_{t \in T_L^{(j)}} \Lambda(t)\, q_t^{(j)}[x^{(j)}, y^{(j)}] + \sum_{t \in T_S} \Lambda(t) \prod_{j=0}^{J-1} q_t^{(j)}[x^{(j)}, y^{(j)}] \\ &= \sum_{j=0}^{J-1} \sum_{\substack{t \in T_L^{(j)} \\ x^{(j)}[t\rangle y^{(j)}}} \Lambda(t) + \sum_{t \in T_S, \mathbf{x}[t\rangle \mathbf{y}} \Lambda(t) = \sum_{t \in T, \mathbf{x}[t\rangle \mathbf{y}} \Lambda(t), \end{aligned} \tag{5.2}$$

which is the same as eq. (2.10) on page 14. Indeed, eq. (5.1) is a representation of the infinitesimal generator matrix.

The matrices $Q_L^{(j)}$ and $Q_t^{(j)}$ and the vector $-Q_O \mathbf{1}^{\mathrm{T}}$ together are usually much smaller than the full generator matrix Q even when stored in a sparse matrix form. Hence

Kronecker decomposition may save a significant amount of storage at the expense of some computation time.

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

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

In contrast, *actual Kronecker methods* [5, 11, 45] 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 [3, 10, 13].

Macro state construction

The hierarchical structure of the reachable state space expresses RS as

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

where $\widetilde{RS} = \{\tilde{0}, \tilde{1}, \ldots, \widetilde{\tilde{n}-1}\}$ is the set of *global macro states*, $\widetilde{RS}^{(j)} = \{\tilde{0}^{(j)}, \tilde{1}^{(j)}, \ldots, \widetilde{\tilde{n}_j - 1}^{(j)}\}$ is the set of *local macro states* of $LN^{(j)}$, and $RS^{(j)}_{\tilde{x}} = \{0^{(j)}_{\tilde{x}}, 1^{(j)}_{\tilde{x}}, \ldots, (n_{j,\tilde{x}}-1)^{(j)}_{\tilde{x}}\}$ are the *local micro states* in the local macro state $\tilde{x}^{(j)}$. The product symbol denotes the composition of local markings, as in eq. (2.15) on page 19.

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

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

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

$$x^{(j)} \sim^{(j)} y^{(j)} \iff \{\hat{\mathbf{z}}^{(j)} : \mathbf{x} \in RS, z^{(j)} = x^{(j)}\} = \{\hat{\mathbf{z}}^{(j)} : \mathbf{y} \in RS, z^{(j)} = y^{(j)}\}, \tag{5.3}$$

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

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

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

Algorithm 5.2 Hiearchical decomposition of the reachable state space into macro states by Buchholz [10].

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

```
allocate bit vector b ∈ {0,1}^{n_0 n_1 ⋯ n_{J-1}} initialized with zeroes
foreach x ∈ RS do
    // Fill b with ones corresponding to reachable states
    b[n_{J-1} n_{J-2} ⋯ n_1 x^(0) + n_{J-1} n_{J-2} ⋯ n_2 x^(1) + ⋯ + n_{J-1} x^(J-2) + x^(J-1)] ← 1
for j ← 0 to J − 1 do
    Reshape b into matrix B with n_j columns
    Partition the columns of B by componentwise equality
    foreach element S of the equality partition of the columns of B do
        Create a new local macro state ỹ^(j) in RS̃^(j)
        Assign all local micro states z ∈ S to ỹ^(j)
        Drop all columns of B corresponding to S but a single representant of ỹ^(j)
foreach x̃ ∈ RS^(0) × RS^(1) × ⋯ × RS^(J-1) do
    if b[x̃] = 1 then Add x̃ to RS̃ as a global macro state
return RS̃, {RS̃^(j)}_{j=0}^{J-1}
```

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

3. The set of potential macro states is

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

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

The pseudocode for this process is shown in Algorithm 5.2. The decomposition is extremely memory demanding due to the allocation of the bit vector $\mathbf{b}$ of length $|PS|$.

In [10], sorting the columns of B lexicographically was recommended to calculate the equality partition of the columns of B. In our implementation, we insert the columns

of B into a bitwise trie and detect duplicates instead, so that no mapping between the original order and sorted ordering of columns needs to be maintained.

Running example 5.1 The macro states of the *RunningExample* SSPN model (Figure 2.6 on page 18) are obtained from its component state space (Table 2.2 on page 19) as follows:

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

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

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

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

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

   in order to conclude that

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

3. After removing all local states of $LN^{(0)}$ except representants of $\widetilde{RS}^{(0)}$, the order of components is shifted by one and $\mathbf{b}$ is reshaped again

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

   to find that

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

4. Finally, after shifting the order of components again, we reshape **b**

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

and conclude

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

5. Unfolding the matrix B

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

shows that the reachable global macro states are

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

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

Block Kronecker matrix composition

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

The matrices $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}]$ and $Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{x}^{(j)}] \in \mathbb{R}^{n_{j,x} \times n_{n,y}}$ describe the effects of a single transition $t \in T$ and the aggregated effects of local transitions on $LN^{(j)}$ as its state changes from the local macro state $\tilde{x}^{(j)}$ to $\tilde{y}^{(j)}$, respectively. Formally,

$$q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}][a_x^{(j)}, b_y^{(j)}] = \begin{cases} 1 & \text{if } a_x^{(j)}\,[t\rangle\, b_y^{(j)}, \\ 0 & \text{otherwise,} \end{cases} \tag{5.4}$$

$$Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] = \sum_{t \in T_L^{(j)}} \Lambda(t)\, Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}]. \tag{5.5}$$

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

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

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

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

where $N_1 = \prod_{f=0}^{h-1} n_{h,x^{(h)}}$, $N_2 = \prod_{f=h+1}^{J-1} n_{h,x^{(h)}}$. If $\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$, the matrix block describes transitions which leave the global macro state unchanged, therefore any local transition may fire. If $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$ is slmst. at h, only local transitions on the component h may cause the global state transition, since no other local transition may affect $LN^{(h)}$. In every other case, only synchronizing transitions may occur.

This expansion of block matrices is equivalent to eq. (5.1) on page 36 except the considerations to the hierarchical structure of the state space.

The full Q matrix is written as

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

as usual.

Algorithm 5.3 shows the construction of the local transition matrices according to eqs. (5.4) and (5.5).

The construction of the block matrix Q is shown in Algorithm 5.4 on page 48. We optimized the formulation from eq. (5.2) in several ways:

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

Algorithm 5.3 Transition matrix construction for block Kronecker matrices

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

for $j \leftarrow 0$ **to** $J-1$ **do**
  foreach $(\tilde{x}^{(j)}, \tilde{y}^{(j)}) \in \widetilde{RS}^{(j)} \times \widetilde{RS}^{(j)}$ **do**
    if $j \in \operatorname{supp} t$ **then**
      allocate $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \in \mathbb{R}^{n_{j,x} \times n_{n,y}}$
      Fill in $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}]$ according to eq. (5.4) on page 40
    else if $\tilde{x}^{(j)} = \tilde{y}^{(j)}$ **then** $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \leftarrow I_{n_{j,x} \times n_{j,y}}$
    else $Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \leftarrow 0_{n_{j,x} \times n_{j,y}}$
  allocate $Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \in \mathbb{R}^{n_{j,x} \times n_{n,y}}$
  foreach $t \in T_L^{(j)}$ **do**
    $Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] \leftarrow Q_L^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}] + \Lambda(t) Q_t^{(j)}[\tilde{x}^{(j)}, \tilde{y}^{(j)}]$

5.2 Symbolic methods

5.2.1 Edge-valued multivalued decision diagrams

Edge-valued multivalued decision diagrams (EDDs) [71] provide a compact, graph-based representation for functions of the form $\mathbb{N}^J \to \mathbb{N}$.

Definition 5.1 A quasi-reduced ordered *edge-valued multivalued decision diagram* (EDD) encoding the function $g(x^{(0)}, x^{(1)}, \ldots, x^{(J-1)}) \in \mathbb{N}$ is a tuple $EDD = (V, \underline{r}, \underline{0}, \underline{1}, level, children, label)$, where

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

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

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

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

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

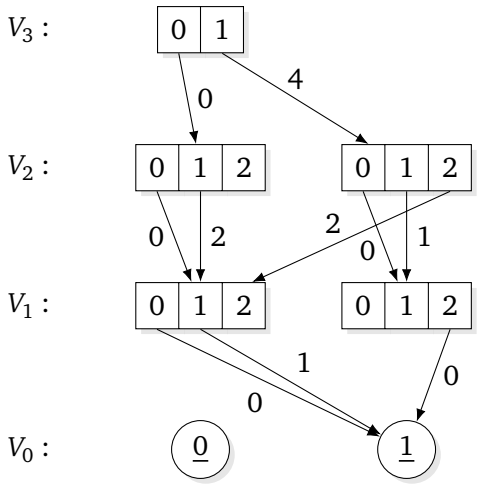


Figure 5.1 EDD state space mapping for the *SharedResource* SSPN.

5.2.2 Symbolic state spaces

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

Running example 5.2 Figure 5.1 shows the state space of the *SharedResource* model encoded as an EDD. The edge labels express the lexicographic mapping of symbolic state codes $\mathbf{x}$ to state indices $\mathbf{x}$. Edges to the terminal zero node $\underline{0}$ were omitted for the sake of clarity.

Algorithms 5.5 and 5.6 on page 49 illustrate the construction of a generator matrix based on the state space encoded as EDDs. The procedure FillIn descends the EDD following a path for the target and the source state simultaneously. The edge labels, representing state indices, are summed on both paths. If a transition $\mathbf{x}\,[t\rangle\,\mathbf{y}$ is found, the matrix element $q_O[x, y]$ corresponding to the summed indices is incremented by the transition rate $\Lambda(t)$. Algorithm 5.6 repeats FillIn for all transitions.

5.2.3 Symbolic hierarchical state space decomposition

The memory requirements and runtime of Algorithm 5.2 on page 38 may be significantly improved by the use of symbolic state space storage instead of a bit vector.

To symbolically partition the local states $RS^{(j)}$ into macro states $\widetilde{RS}^{(j)}$, we will use the following notations of *above* and *below* substates from Ciardo et al. [20]:

Definition 5.2 The set of *above* substates coded by the node $\underline{n}$ is

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

such that

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

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

Definition 5.3 The set of *below* substates coded by the node $\underline{n}$ is

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

such that

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

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

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

Observation 5.4 The set of states which contain some local state $x^{(j)}$ is

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

Proof. Any reachable state $\mathbf{z} \in RS$ that has $z^{(j)} = x^{(j)}$ is represented by a path in the MDD that passes through a pair of nodes $\underline{n} \in V_{j+1}$ and $children(\underline{n}, x^{(j)}) \neq \underline{0}$. Therefore, some path $\mathbf{a} \in \mathcal{A}(\underline{n})$ must be followed from $\underline{r}$ to reach $\underline{n}$, then some path $\mathbf{b} \in \mathcal{B}(children(\underline{n}, x^{(j)}))$ must be followed from $children(\underline{n}, x^{(j)})$ to $\underline{1}$.

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

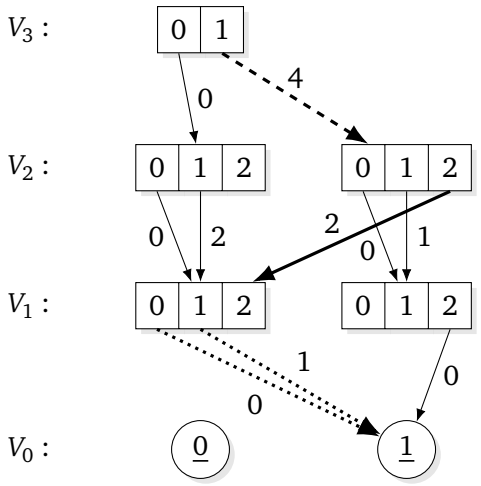


Figure 5.2 The set of all paths having $x^{(1)} = 2^{(1)}$ in the *SharedResource* EDD.

Running example 5.3 Figure 5.2 shows all path in the *SharedResource* MDD with $x^{(1)} = 2^{(1)}$.

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

Observation 5.5 If $\underline{n}$ and $\underline{m}$ are distinct nonterminal nodes of a quasi-reduced ordered MDD, $\mathcal{A}(\underline{n}) \cup \mathcal{A}(\underline{m}) = \emptyset$ and $\mathcal{B}(\underline{n}) \neq \mathcal{B}(\underline{m})$.

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

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

Observation 5.6 The relation $x^{(j)} \sim^{(j)} y^{(j)}$ can be expressed as

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

Proof. Let

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

According to eq. (5.3) on page 37, $x^{(j)} \sim^{(j)} y^{(j)}$ if and only if $X = Y$.

Define

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

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

According to Observation 5.4,

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

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

Observation 5.6 can be interpreted as the statement that $x^{(j)} \sim^{(j)} y^{(j)}$ if and only if the MDD edges corresponding to $x^{(j)}$ are always parallel, i.e., from the node $\underline{n}$ they all go to the same node $\underline{m}(\underline{n})$ for all $\underline{n} \in V_{j+1}$.

The macro states can be constructed from the parallel edges in the MDD by partition refinement. This process is performed by Algorithm 5.7 on page 50.

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

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

A block Kronecker matrix may be constructed from the decomposed state space by Algorithm 5.4.

5.3 Matrix storage

Existing linear algebra and matrix libraries, such as [8, 30, 40, 53, 73], usually have unsatisfactory support for operations required in stochastic analysis algorithms with decomposed matrices, for example, multiplications with Kronecker and block Kronecker matrices. Therefore, we have decided to develop a linear algebra framework in C#.NET specifically for stochastic algorithms as a basis of our stochastic analysis framework.

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

Figure 5.3 Compressed Column Storage of a matrix.

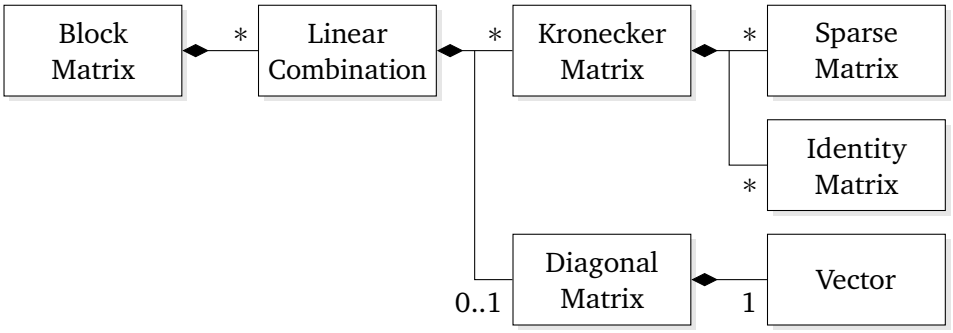


Figure 5.4 Data structure for block Kronecker matrices.

Decomposed Kronecker and block Kronecker matrices are stored as algebraic expression trees as shown in Figure 5.4. Matrix multiplication and manipulation algorithms for expression trees are detailed in Section 6.4 on page 56. Benchmark results for the block Kronecker decomposition are discussed in Section 9.3 on page 73.

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.

Algorithm 5.4 Block Kronecker matrix construction.

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

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

Algorithm 5.5 FillIn procedure for matrix construction from EDD state space.

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

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

Algorithm 5.6 Sparse matrix construction from EDD state space.

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

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

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

Algorithm 5.7 Local macro state construction by partition refinement.

Input: Symbolic state space *MDD*

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

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

Chapter 6

Algorithms for stochastic analysis

Steady state, transient 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 large.

In steady-state and sensitivity analysis, linear equations of the form $\mathbf{x}A = \mathbf{b}$ are solved, such as eqs. (2.3) and (2.7). The steady-state probability vector is the solution of the linear system

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

where the infinitesimal generator Q is a rank-deficient matrix. Therefore, steady-state solution methods must handle various generator matrix decompositions and homogenous linear equation with rank deficient matrices. Convergence and 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 such as eq. (2.2) on page 8 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. We achieved genericity by defining an interface between the algorithms and the data structures with operations including

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

In this chapter, we describe some of the algorithms implemented in our stochastic analysis framework restricting our attention to direct solvers that can compute solutions

Algorithm 6.1 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 $j \leftarrow 0$ **to** $n-1$ **do**
  for $i \leftarrow 0$ **to** j **do** $a[i,j] \leftarrow a[i,j] - \sum_{k=0}^{i-1} a[i,k]a[k,j]$
  for $i \leftarrow j+1$ **to** $n-1$ **do** $a[i,j] \leftarrow \left(a[i,j] - \sum_{k=0}^{j-1} a[i,k]a[k,j]\right) \big/ a[i,i]$
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

of smaller CTMCs with high precision. For additional numerical algorithms supported by our framework we refer to [51].

6.1 Direct linear equation solvers

6.1.1 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 [74].

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

$$A = LU.$$

To solve the equation

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

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

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

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

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

We used Crout's LU decomposition [66, Section 2.3.1], presented in Algorithm 6.1, 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 during the decomposition even if it was initially sparse, therefore it should first be copied to a dense

Algorithm 6.2 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** $x[n-1] \leftarrow 1$ // 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 x**

array storage for efficiency reasons. This considerably limits the size of Markov chains that can be analysed by direct solution due to memory constraints. Our data structure allows access to upper and lower diagonal parts of matrices and linear combinations, therefore no additional storage is needed other than A itself.

The forward and back substitution process is shown in Algorithm 6.2. If multiple equations are solved with the same matrix, its LU decomposition may be cached.

Matrices of less than full rank

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

In practice, the algorithm can be used to solve homogenous equations in Markovian 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 $\pi = \mathbf{x}/(\mathbf{x}\mathbf{1}^{\mathrm{T}})$ to get a stationary probability distribution vector.

6.1.2 Improving LU decomposition with partial pivoting

Despite being a direct solver, LU decomposition is inherently numerically unstable, just like Gaussian elimination. This instability originates from line 3 of Algorithm 6.1 where we perform a division by $a[i,i]$. In order to stabilize the LU decomposition numerically

we can use partial pivoting, i.e., for every elimination step we reorder the rows of the matrix in a way that ensures the largest denominator $a[i,i]$ in that step. This way we can minimize the occurring numerical errors with relatively small performance overhead.

The partial pivoting method for Crout's decomposition is fairly simple [67, Section 2.3]. We delay the division during the iterations at line 3 and calculate the values $a[i,j]$ without it. Once the values are computed for a column, we find the element with the biggest absolute value in the column's subdiagonal part and perform the necessary row ordering. As the final step we perform the division on the reordered values.

Parts of the referenced algorithm can be simplified in case of Markovian analysis. Firstly, we only have to iterate through the diagonal to find the element of the matrix with the biggest absolute value, since the diagonal elements are the negative of the sum of the corresponding offdiagonal row elements. Secondly, instead of performing explicit row ordering we can use indirect indexing of elements, so we only need to perform the ordering on the index mapping.

6.2 Transient analysis

6.2.1 Uniformization

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

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

by computing

$$\boldsymbol{\pi}(t) = \sum_{k=0}^{\infty} \boldsymbol{\pi}_0 P^k e^{-\alpha t} \frac{(\alpha t)^k}{k!}, \tag{6.1}$$

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

eq. (6.1) can be realized as

$$\mathbf{x} = \frac{1}{W}\left(\sum_{k=0}^{k_{\text{left}}-1} w_{\text{left}} \boldsymbol{\pi}_0 P^k + \sum_{k=k_{\text{left}}}^{k_{\text{right}}} w[k-k_{\text{left}}] \boldsymbol{\pi}_0 P^k \right), \tag{6.2}$$

where $\mathbf{x}$ is $\boldsymbol{\pi}(t)$, 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. (6.2) is implemented by Algorithm 6.3. The algorithm performs *steady-state* detection in line 9 to avoid unnecessary work once the iteration vector $\mathbf{p}$ reaches the

Algorithm 6.3 Uniformization.

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

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

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

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 [34]. However, their algorithm is extremely complicated due to the limitations of single-precision floating-point arithmetic [44]. The stochastic framework provides an implementation of Burak's significantly simpler algorithm [14] (which is not in the scope of this work) in double precision instead, which avoids underflow by a scaling factor $W \gg 1$.

6.3 Mean time to first failure

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

$$MTFF = -\underbrace{\boldsymbol{\pi}_U Q_{UU}^{-1}}_{\gamma} \mathbf{1}^{\mathrm{T}}, \quad \mathbb{P}(X(TFF_{+0}) \in D') = -\underbrace{\boldsymbol{\pi}_U Q_{UU}^{-1}}_{\gamma} \mathbf{q}_{UD'}^{\mathrm{T}} \qquad \text{(2.8, 2.9 revisited)}$$

Algorithm 6.4 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
4 |   for i ← 0 to k − 1 do
5 |   |   c[j] ← c[j] + b[i]A[i, j]        // Scaled addition of vector-matrix product
```

Algorithm 6.5 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: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^m$

```
1 allocate c ∈ ℝ^m if no target buffer is provided
2 c ← 0
3 for h ← 0 to k − 1 do
4 |   c ← ν_h · bA_h          // In-place scaled addition of vector-matrix product
5 return c
```

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

$$\boldsymbol{\gamma} Q_{UU} = \boldsymbol{\pi}_U \tag{6.3}$$

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 $\varphi_D = \neg\mathsf{AX}\,\mathsf{true} \vee \varphi_1 \vee \varphi_2 \vee \cdots$ and $\varphi_U = \neg\varphi_D$, where the deadlock condition $\neg\mathsf{AX}\,\mathsf{true}$ is added to make (6.3) 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 (6.3) can be executed as in steady-state analysis.

6.4 Efficient vector-matrix products

Iterative linear equation and transient distribution solvers require several vector-matrix products per iteration. Therefore, efficient vector-matrix multiplication algorithms are

Algorithm 6.6 The SHUFFLE algorithm for vector-matrix multiplication.

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

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

required for the various matrix storage methods (i.e., dense, sparse and block Kronecker matrices) to support configurable stochastic analysis.

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

Implemented matrix multiplication for the data structure (see Figure 5.4 on page 47) routines are

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

  Operations with vectors and sparse matrices are implemented in an `unsafe`[1] context. The elements of the data structures are not under the influence of the Garbage Collector runtime, but stored in natively allocated memory. This allows the handling of large matrices without adversely impacting the performance of other parts of the program, albeit the cost of allocations is increased.

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

- Multiplication with block matrices by delegation to the constituent blocks of the matrix (Algorithm 6.4 on page 56). The input and output vectors are converted to block vectors before multiplication. If parallel execution is required, each block of the output vector can be computed in a different task, since it is independent from the others.

- Multiplication by a linear combination of matrices is delegated to the constituent matrices (Algorithm 6.5 on page 56). An in-place scaled addition of vector-matrix product to a vector operation is required for this delegation. To facilitate this, each vector-matrix multiplication algorithm is implemented also as an in-place addition and in-place scaled addition of vector-matrix product, and the appropriate implementation is selected based on the function call arguments.

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

- Multiplications by Kronecker products is performed by the SHUFFLE algorithm [4, 11] as shown in Algorithm 6.6 on page 57.

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

  SHUFFLE rewrites the Kronecker products as

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

  where $I_{a\times a}$ denotes an $a \times a$ identity matrix. Multiplications by terms of the form $I_{N\times N} \otimes A^{(h)} \otimes I_{M\times M}$ are carried out in the loop at line 8 of Algorithm 6.6.

  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 [33] and SPLIT [25] algorithms, which are more amenable to parallel execution than SHUFFLE. Their implementation is in the scope of our future work.

Chapter 7

Post-processing numerical results

The output of the numerical solution step in the stochastic analysis workflow is a (possibly accumulated) probability distribution vector that assigns a probability to every state of the modeled system. In case of steady state and transient analysis this vector is $\boldsymbol{\pi}$ and $\boldsymbol{\pi}(t)$, respectively.

Usually we need more information about the system than just the probability distribution of its states. We can assign reward rates to the individual states of the system to describe some desired metric (as detailed in Sections 2.2.1 and 2.3.1) to for example compute the expected value of that metric. For the computation of the expected value, the probability distribution vector is given as the output of the previous analysis step. However, we still need an efficient way to describe the reward rates associated with the states of the system.

In this chapter we detail the different ways of assigning reward rates to states in an easy and maintainable manner. Furthermore, we provide some insight into the efficient calculation of these measures and their sensitivity analysis to net parameters.

7.1 Reward configurations

In order to construct the reward vector we need to be able to somehow reference the states of the system (preferably not one-by-one). Once we can reference a set of states, we need to assign to the a reward rate that potentially depends on the actual state it's assigned to. The implemented stochastic analysis framework supports this reward vector construction with the following definitions:

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

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

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

The reward weight expression in the CTL-based definition is an algebraic expression that may contain various operations and references:

1. The common arithmetic operators: $+, -, \hat{}, *$, / and // for integer division.

2. Some predefined functions: $exp()$, $lg()$, $lb()$, $ln()$, $log(,)$, $sin()$, $cos()$.

3. The defined net parameters (accessed through the variable's name).

4. The rate of a transition (accessed through the $rate()$ function and the transition's name as the argument).

5. The number of tokens on a place in the currently evaluated state (accessed through the name of the place).

References to places are replaced by the number of tokens upon evaluation, while references to transition rates can be evaluated at the beginning of the analysis. Furthermore, note, that the reward expression (p, w) may be written as $\mathsf{true} \to p \cdot w$ or $p > 0 \to p \cdot w$ using CTL. Reward expressions with CTL are only allowed when symbolic state space representation is used, as CTL evaluation[1] is performed symbolically [26].

Defining the entire reward vector $\mathbf{r}$ using only one definition from above is cumbersome. To alleviate this problem the framework uses the concept of reward configuration to describe the vector $\mathbf{r}$. A reward configurations may contain one or more reward definitions and each definition corresponds to a reward vector $\mathbf{r}_i \in \mathbb{R}^{|RS|}$. The complete reward vector for the modeled system is expressed as the following sum:

$$\mathbf{r} = \sum_i \mathbf{r}_i \tag{7.1}$$

Running example 7.1 Consider the reward structures defined over the *Shared-Resource* Petri net from Running example 2.6 on page 16.

[1] The symbolic state space exploration and CTL evaluation component is currently provided by the PetriDotNet [31] tool.

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

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

which is equivalent to the SRN reward structure

$$rr_1(M) = M(p_{S_1}) + M(p_{S_2}), \quad ir_1(t, M) \equiv 0. \tag{2.11 revisited}$$

This can also be written as

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

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

Completed calculations are described by

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

which is equivalent to the reward structure

$$rr_2(M) \equiv 0, \quad ir_2(t, M) = \begin{cases} \Lambda(t) & \text{if } t \in \{t_{r_1}, t_{r_2}\}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.12 revisited}$$

7.2 Efficient reward calculation

To calculate the expected value of a measure we need to form the inner product $\mathbb{E}R(t) = \boldsymbol{\pi}(t)\mathbf{r}^{\mathrm{T}}$. If we were to generate $\mathbf{r}$ according to eq. (7.1) that would pose a severe memory overhead. Usually not every state has a reward rate value associated with it so the vector $\mathbf{r}$ in the inner product is a sparse vector. Thus eq. (2.4) on page 10 can be changed accordingly:

$$\mathbb{E}R(t) = \sum_{i, r[i] \neq 0} \pi(t)[i] r[i] \tag{7.2}$$

Combining eq. (7.2) with eq. (7.1) we can write the calculation of the expected value as:

$$\mathbb{E}R(t) = \sum_{j} \sum_{i, r_j[i] \neq 0} \pi(t)[i] r_j[i] \tag{7.3}$$

As we can see from eq. (7.3) every $r_j[i]$ value is used only once, when we multiply it with $\pi(t)[i]$. This means that even though the vectors $\mathbf{r}_j$ are sparse vectors, it is

meaningless to store their elements: as soon as we calculate the appropriate $r_j[i]$ value we can perform the multiplication with $\pi(t)[i]$ and accumulate the result.

The framework employs several optimization methods to ensure the efficient generation of $r_j[i]$ reward rate values.

Arithmetic expression evaluation As the reward rates are defined with the help of arithmetic expression trees, it is essential to evaluate them in an efficient way. If an arithmetic expression references only net parameters and transition rates then it can be reduced to a constant expression by substituting the appropriate values at the beginning of the workflow (before constructing the generator matrix). However, if the expression contains references to the marking of the currently evaluated state then that expression must be re-evaluated for every necessary state.

Evaluation of an expression tree means that we substitute every variable in the tree with their respective values. Generally this can be achieved by a recursive graph traversal method (for example by using the visitor pattern [62]) to find and replace the variables in the tree. However, in order to acquire the value represented by the expression we also need to evaluate (i.e., ”execute”) the operator nodes which could severely impact the performance. Fortunately, the tree describes an arithmetic expression and most of the programming languages provide built-in support for efficiently executing such expressions.

Our framework ”compiles” the expression tree by generating executable Intermediate Language (IL) code from it that contains parameters.[2] This parameterized code can be more efficient than ”manually” substituting variables. We only need to compile the expression once, then execute it for each relevant state with the appropriate markings (expression parameters).

Guided state space enumeration The other challenge of efficient reward evaluation is the enumeration of only those states that have a nonzero $r_j[i]$ reward rate value associated with them. In case of explicit state spaces we have to enumerate every state in order to evaluate the associated (place- and transition-based) reward definitions. However, decision diagram-based state spaces provide means to exclude a large number of states based on a local information in a component’s state space. For example, it is easy to find every reachable state that contains a specific marking pattern for some places or where a certain transition is enabled.

Based on this observation the framework makes it possible to ”guide” the symbolic state space enumeration based on some constraint that can be one or more of the following:

Unrestricted The guide doesn’t constrain the enumeration of states at all.

[2] `https://msdn.microsoft.com/en-us/library/mt654263.aspx`

MDD guide The allowed paths to take in the decision diagram are constrained by an other MDD. This other MDD is usually the result of a model checking algorithm that was used to gather the states affected by a CTL-based reward definition. Note that place-based reward definitions can be transformed to CTL-based reward definitions.

Target state guide This iterator guide only allows iteration through states in which a given transition is enabled and firing it results in a global state allowed by another iterator guide. This guide combined with an unrestricted guide can be used to evaluate the transition-based reward definitions. Even though these definitions could be transformed to CTL-based definitions using the structure of the net, it is more efficient to use the additional local state space information of components than performing model checking and using an MDD guide.

Composite guide Different guides can be combined together resulting in the *conjunction* of their respective constraints.

7.3 Sensitivity calculation

Transition and reward rates are stored as algebraic expression trees in the input SPN models. Symbolic operations, such as partial differentiation may be performed directly on the trees using algebraic laws, before evaluating the expressions (for example in order to construct the generator matrix).

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

Steady-state sensitivity calculation, shown in Figure 7.1, is the most complicated post-processing in the workflow. Partial derivatives of the transition rate expressions and reward weight expressions are taken to calculate $\partial \mathbb{E}R/\partial\theta_i$ using eqs. (2.6) and (2.7) on page 11 and on page 12.

7.4 Interval-based measure calculation

Usually we are interested in the performance metric values not only for one value of a parameter, but for a range of values of parameters. The framework provides means to define ranges and sample point densities for parameters and then it performs the selected analysis for every sample point in the parameter space.

The drawback of this method is that it performs the complete analysis workflow for every sample point, even though the result of state space exploration or the macro state-based decomposition could be reused. However, the individual analyses are completely

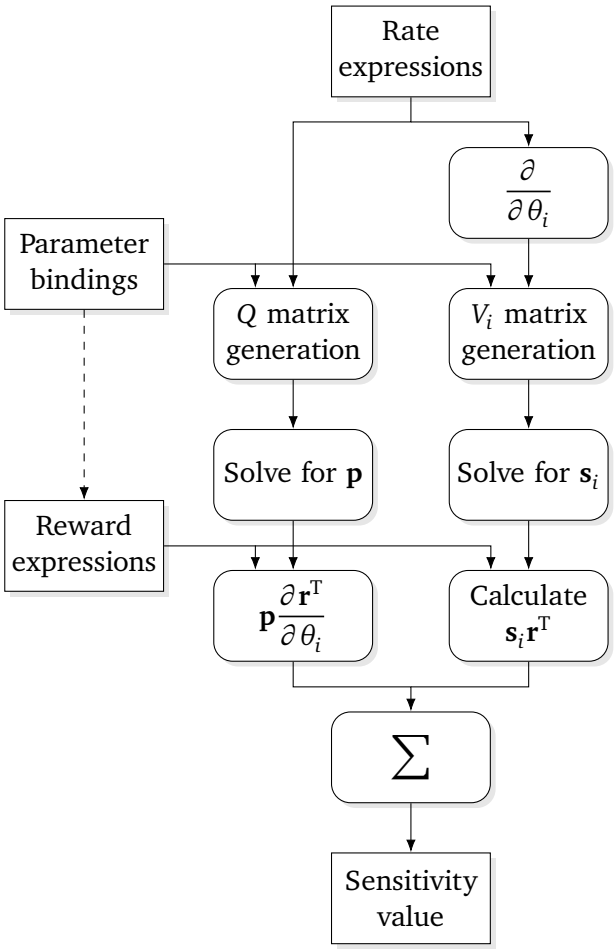


Figure 7.1 Reward and sensitivity calculation from expression tree inputs.

independent of each other so the parameter space can be divided into multiple parts and distributed across arbitrary number of computers. The next chapter in Section 8.2 will present an alternative method to evaluate measures across parameter intervals without performing the analysis multiple times. However, it lacks the scalability of this approach and is only applicable for smaller models.

Chapter 8

Symbolic evaluation

The presence of parameters in a model opens up new possibilities among the performable analysis types, e.g., sensitivity analysis. However, to be able to perform such analyses, a special infrastructure is needed that can handle the various tasks that the incorporation of parameters into the workflow presents.

In this chapter we provide an overview of the parameter-related parts of the framework, including the arithmetic grammar, the application of expression trees during analysis and the integration of higher precision software arithmetic.

8.1 Arithmetic grammar

So far in this thesis we always mentioned expression trees when talking about arithmetic expressions, like those used in transition rate expressions or reward weight expressions. These trees usually need to be read from user provided input and later persisted in a similar form. In order to do this we need to specify a grammar for the textual specification of arithmetic expressions.

For the construction of the arithmetic grammar we used the ANTLR4 tool [63]. The tool allows the specification of annotated grammar files from which it generates the necessary lexers and parsers at the selected programming language (C#.NET in our case). Another useful feature of the tool is the ability to resolve left recursive definitions which makes it easier to define the usual priority rules among the arithmetic operators.

Besides the operators, the basic grammar file defines the syntax for variable names and function invocations only. The variable name syntax conforms to the naming rules of the PNML standard and allows references to hierarchical variable names which is essential to support the element names of a hierarchical model in the PetriDotNet tool. The syntax for variable names is the following:

```
ID : [a–zA–Z] ([a–zA–Z.] | DIGIT | '_')* ;
```

The definition of function syntax also allows hierarchical naming, moreover it allows the function to have an arbitrary number of parameters. The syntax is the following:

```
ID LPAR expr (, expr)* RPAR
```

where *LPAR* and *RPAR* are the symbols for the left and right parenthesis, respectively.

After parsing an expression, ANTLR4 returns an expression tree with the defined primitives in it. We use the visitor pattern to transform that tree into our own expression tree data structure. This data structure can be extended with custom tree builder components that can handle the unknown parts of the expression tree, such as functions or variables. The basic functions listed in Section 7.1 on page 59 were added to the data structure using this extension point.

Besides the support for arbitrary function and variable definitions, there was an other reason why we created our own arithmetic library instead of using an existing one. The PetriDotNet framework already contained a library that defined a similar grammar for boolean and CTL expressions. In that library CTL expressions can reference boolean expressions as atomic propositions. We plan to merge the two expression libraries, making it possible for boolean expressions to reference arithmetic expressions in their relational propositions (e.g., in the two operands of the $<$ operator). This would provide support for defining more complex temporal and boolean expressions.

8.2 Symbolic measure computation

In the original workflow of the framework, we substituted every parameter reference in the transition rates for its value defined by the current parameter configuration in an early step. This resulted in a purely numerical problem that can be handled by a variety of solvers provided by the framework.

If we do not perform this substitution, then we can construct a generator matrix whose elements are expression trees instead of floating point values. The unknown values (parameters) of this matrix, however, impose some restrictions on the solvers that we can use for the computation of the probability distribution vector (that will also contain expression trees). Solvers that need a predefined precision or error margin value can not be used, since we can not perform the necessary convergence test to determine the termination criteria due to the unknown quantities in the matrix. For this reason employing iterative solvers and Krylov-subspace methods is not feasible in this context.

What we can use are direct solvers like LU decomposition that computes the elements of the required result vector in a closed form. Since all the necessary arithmetic operators have a corresponding representation in the expression tree, we can simulate the execution of these operators by constructing new expression trees from their operands. This results in a vector whose elements are closed functions of the net

parameters. Once we acquire the symbolic probability vector, we can perform the same post processing on it as discussed in Section 7.2 on page 61. Again, we ”execute” the arithmetic operators during post processing by constructing new expression trees. At the end of the symbolic workflow the required measure will be defined by a closed function of the net parameters.

The main advantage of the symbolic evaluation of measures is that we acquire a closed function of net parameters that is independent of the actual values of the parameters. Even if we change the parameter configuration we don't need to re-run the analysis workflow since the result will be the same. Instead, we can evaluate the function for arbitrary parameter values which can be further improved by generating executable IL code from the measure expression (as described in Section 7.2 on page 61). This means that we can perform the interval-based measure calculation (see Section 7.4 on page 63) more efficiently.

Another advantage of the symbolic result is that we have a wide arsenal of mathematical methods at our disposal to perform additional post processing on the function. For example, instead of the complicated sensitivity analysis depicted in Figure 7.1 on page 64, we can simply differentiate symbolically (one or more times) the measure expression to gain some information about its sensitivity to model parameters.

Despite the advantages of symbolic evaluation, the mandatory use of direct solvers (like LU decomposition) severely limits the scalability of the approach. Direct solvers usually operate on dense matrices even if the generator matrix is originally a sparse matrix. This results in a high memory and performance overhead when performing symbolic evaluation on larger models.

8.3 Arbitrary precision evaluation

Even thought symbolic evaluation does not use excessive numerical calculations, it doesn't solve the instability problem of the LU decomposition, it only delays it to the time when we evaluate the measure expression. In performability related problems it is preferable to use arbitrary precision arithmetic to avoid the unwanted impact of numerical errors at the price of a runtime overhead originating in using a software library instead of the hardware arithmetic. This is achieved by allocating more memory and CPU time to the operations. The additional precision of the results, which is an important requirement for the verification of safety-critical systems, should outweigh the overhead of the arbitrary precision calculation.

In order to alleviate the problems caused by numerical errors we provide support for arbitrary precision arithmetic in the stochastic framework. This option can be used during both numerical and symbolical evaluation of performance measures, so it is not confined only to the LU decomposition algorithm, other solvers can benefit from it as

well. For the prototype phase of this feature we integrated a C# library called *Numbers*[1] with our arithmetic expression library and created a high precision version of the data structures and related algorithms.

[1] https://github.com/peteroupc/Numbers

Chapter 9

Evaluation

9.1 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 framework.

9.1.1 Combinatorial testing

As described in Chapter 6, algorithms use the common vector and matrix interfaces to perform various operations. This makes the used storage techniques transparent which in turn makes the code base more concise, reusable and less prone to errors.

The most important requirement against the data structure operations is mathematical correctness regardless of the storage technique used. Considering the number of implementations for a given interface and the previous requirement we used a simple unit testing design pattern (also known as interface testing pattern) as the core building block for the data structure testing [59].

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 (like *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 we need an instance for a given interface we delegate the instantiation to an abstract factory method in the test class.

A drawback of this approach is that the test class itself becomes abstract so we can't run the tests inside it directly. However we can easily inherit from the base test class and implement the abstract factory methods in any way we'd like. But the most important advantage of this approach manifests itself when we apply the virtual modifier to one or more unit tests in the base class. This way we can completely override tests in the derived classes if needed based on the types of the interface implementations. So the first step in testing the data structure library was to implement these abstract unit tests that operate on an interface level.

Abstract tests

In order to make sure we cover the most possible usage scenarios of the data structure we followed some common testing techniques. As a first step we used equivalence partitioning to identify the valid and invalid ranges of the parameters of the operations. Next we implemented the parameter value checks in interface code contract classes using Microsoft's Code Contract library [54]. This enabled us to implement the parameter check logic in one place for an operation making the code more maintainable. Moreover every class implementing a data structure interface and it's operations will automatically contain these logics if code contracts are enabled. Code contracts can be disabled if needed resulting in a performance boost for the data structure library since the parameter checks are skipped.

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

For this purpose we used Microsoft's IntelliTest tool [55] (formerly known as Pex, [76]) which assists in automating white-box and unit-testing. IntelliTest automatically generates unit tests using constraint satisfaction problem solving based on the source code of the method under test. Using IntelliTest on our interface code contract classes provided us with many invalid parameter values which we could use in our abstract unit tests.

Concrete tests

Once the abstract unit tests were implemented the next step was to create the derived classes for every storage combination and implement the abstract factory methods. Since the number of possible combinations were too many to implement manually we used Microsoft's Text Template Transformation Toolkit (T4, [56]) to generate the derived

classes. The created template files provide ways to modify the behavior of abstract tests (through simple regular expression based configuration files) and to decrease the number of generated test by using pairwise testing instead of full combinatorial testing of implementation combinations. To generate the combinations for pairwise testing we used the ACTS tool [9].

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.

9.1.2 Software redundancy based testing

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

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

> **Observation 9.1** The result of a performance analysis (e.g., reward calculation) is mathematically independent of the used analysis workflow. It only depends on the possible behaviors of the system and the definition of the required performance measure. Two results calculated by using two different analysis methods can only differ from each other due to the numerical precision properties of the used algorithms.

Combining our fully configurable workflow with Observation 9.1 presents a new approach for testing the algorithm implementations in a maintainable and almost automatic manner. We can take advantage of the concept of software redundancy commonly used in safety critical applications. The main idea behind software redundancy is to perform a calculation multiple times with usually fundamentally different algorithms (often developed by independent teams) thus minimizing the possibility of common mode failures. After the calculations a voting component examines whether every algorithm calculated the same result. If that's not the case then one or more of the algorithms are incorrect.

The building block for this testing phase consists of running our analysis workflow for a given configuration and saving the calculated results (reward and sensitivity values). We generated 588 mathematically consistent configurations in total, executed them for our running example (Figure 2.4), multiple benchmark models and case studies. Finally we examined the maximum absolute difference of the calculated results as an error indicator for each performance measure in each model as presented in the next sections.

Beside verifying the correctness of the developed algorithms, our main goal with software redundancy based testing is to gather a knowledge base about the effectiveness of different analysis approaches for models with varying properties. The gathered observations are summarized in Section 9.3.

9.2 Measurements

In this section we introduce the models used throughout the testing and benchmarking phase then we present preliminary results about the performance of solver algorithms using the implemented block Kronecker decomposition matrix form.

9.2.1 Shared resource

One of the benchmark models was the modified version of stochastic *SharedResource* (SR) system (presented in Figure 2.4). We added three more nodes to the system and modified some of its parameters along two dimensions. On one hand we increased the number of reachable states by adding more resources and local processes to the model. On the other hand we changed the rates of transitions in the model resulting in changes in it's stochastic behaviour. We created symmetric, slightly asymmetric and significantly asymmetric versions of the model. In the third case there are orders of magnitude of difference between the transitions rates of the model.

9.2.2 Kanban

We used the *SPN* version of the *kanban* (KB) system [19] as the other 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.

9.2.3 Cloud performability

One of the models we used for analysis represents a cloud architecture [36] with physical and virtual machines serving incoming jobs using warm and cold spare resources in case of increasing load. We modified some aspects of the model in [36] since our library currently doesn't support the GSPN formalism.

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

Table 9.1 Preliminary benchmark results.

9.2.4 Industrial case study

As an industrial case study we performed stochastic analysis on a safety model of a subsystem that contains two redundant components with self-checking capabilities. The model consisted of a relatively small amounts of states so it was especially amenable to high precision symbolic evaluation of measures.

9.3 Results

Observation 9.2 Based on the preliminary measurements and combinatorial testing we can note some interesting observations:

1. As expected the storage requirement of the block Kronecker form is almost an order of magnitude lower than that of the sparse form.

2. For models with moderate state space sizes (approximately a few millions) the sparse form outperforms the block Kronecker form.

3. However for models with considerably bigger state space sizes (almost a hundred million) the block Kronecker form outperforms the sparse form not just in memory usage but in analysis time as well. This is probably because of the inefficient cache usage of the sparse structure.

4. Slower, but more memory efficient solvers (Gauss-Seidel iteration, Jacobi iteration) often diverged using sparse matrix form while converged using the block Kronecker form. This is due to the possibility of different state ordering in the block Kronecker form.

Chapter 10

Conclusion and future work

We have developed and presented our *configurable stochastic analysis framework* for the dependability, reliability and performability analysis of complex asynchronous systems. Our presented approach is able to combine the strengths and advantages of the different algorithms into one framework. We have not only implemented a stochastic analysis library, but we integrated the various state space traversal, generator matrix representation and numerical analysis algorithms together.

From the theoretical side, we have developed an algorithm which can efficiently compile the symbolic state space representation into the complex data structure representation of the stochastic process. We have formalised our algorithm and proved its correctness. This new algorithm helps us to exploit the efficient state space representation of symbolic algorithms in stochastic analysis.

In addition we have investigated the composability of the various data storage, numerical and symbolic solution and state space 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 more than two state space exploration algorithms, three state space representation algorithms, three generator matrix decomposition and representation algorithms, two steady-state solvers, one transient analysis algorithm and four different computation algorithms for engineering measures. Our long term goal is to provide these analysis techniques for a wider community, so we have integrated our library into the PetriDotNet framework. Our algorithms are also used in 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 comes from there. 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.

Some 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 in order to provide an adaptive framework;

- Distributed implementations of the existing algorithms [15];
- Support for fully symbolic storage and solution of Markov chains [22, 61, 79];
- The use of tensor decompositions instead of vectors to store state distributions and intermediate results to greatly reduce memory requirements of solution algorithms [2, 27, 38];
- Exploiting the advantages of the SPLIT [25] algorithm and providing efficient heuristics for splitting.

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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

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

[15] 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.

[16] 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.

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

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

[19] 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.

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

[21] 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.

[22] 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.

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

[24] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. “Transactions on Petri Nets and Other Models of Concurrency V”. In: ed. by Kurt Jensen, Susanna Donatelli, and Jetty Kleijn. Springer Berlin Heidelberg, 2012. Chap. Ten Years of Saturation: A Petri Net Perspective, pp. 51–95. DOI: 10.1007/978-3-642-29072-5_3.

[25] 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: 9781450300698. URL: http://dl.acm.org/citation.cfm?id=1878537.1878789.

[26] 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.

[27] 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.

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

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

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

[31] 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.

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

[33] 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.

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

[35] 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.

[36] Rahul Ghosh. "Scalable stochastic models for cloud services". PhD thesis. Duke University, 2012.

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

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

[41] 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.

[42] *International Workshop on Timed Petri Nets, Torino, Italy, July 1-3, 1985*. IEEE Computer Society, 1985. ISBN: 0818606746.

[43] 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.

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

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

[46] 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.

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

[48] 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.

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

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

[51] Kristóf Marussy. “Configurable Numerical Solutions for Stochastic Models”. Bachelor’s Thesis. Budapest University of Technology and Economics, 2015. URL: https://diplomaterv.vik.bme.hu/en/Theses/Konfiguralhato-numerikus-modszerek.

[52] Kristóf Marussy, Attila Klenik, Vince Molnár, András Vörös, Miklós Telek, and István Majzik. “Configurable Numerical Analysis for Stochastic Systems”. In: *2nd International Workshop on Symbolic and Numerical Methods for Reachability Analysis (SNR’16)*. Accepted, in press. IEEE, 2016.

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

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

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

[56] 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.

[57] Vince Molnár, Kristóf Marussy, Attila Klenik, András Vörös, István Majzik, and Miklós Telek. "Efficient decomposition algorithm for stationary analysis of complex stochastic Petri net models". In: *Application and Theory of Petri Nets and Concurrency*. Vol. 9698. Lecture Notes in Computer Science. Accepted. Springer, 2016. DOI: 10.1007/978-3-319-39086-4_17.

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

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

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

[61] *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*. IEEE Computer Society, 2012. ISBN: 9781467323468. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6354262.

[62] J. Palsberg and C. B. Jay. "The essence of the Visitor pattern". In: *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*. 1998, pp. 9–15. DOI: 10.1109/CMPSAC.1998.716629.

[63] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999, 9781934356999.

[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] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. Cambridge University Press, 2007. ISBN: 0521880688, 9780521880688.

[68] 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.

[69] 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.

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

[72] *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets, Oxford, UK, June 1986*. Vol. 266. Lecture Notes in Computer Science. Springer, 1987. ISBN: 3540180869.

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

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

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

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

[77] András Vörös, Dániel Darvas, Vince Molnár, Attila Klenik, Ákos Hajdu, Attila Jámbor, Tamás Bartha, and István Majzik. “PetriDotNet 1.5: Extensible Petri Net Editor and Analyser for Education and Research”. In: *Application and Theory of Petri Nets and Concurrency*. Vol. 9698. Lecture Notes in Computer Science. Accepted. Springer, 2016. DOI: 10.1007/978-3-319-39086-4_9.

[78] Michael Weber and Ekkart Kindler. “Petri Net Technology for Communication-Based Systems: Advances in Petri Nets”. In: ed. by Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber. Springer Berlin Heidelberg, 2003. Chap. The Petri Net Markup Language, pp. 124–144. DOI: 10.1007/978-3-540-40022-6_7.

[79] 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.