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

Towards tensor-based extra-functional analysis of complex distributed systems

Scientific Students' Association Report

Author:

Dániel Szekeres

Advisor:

Kristóf Marussy

2020

Contents

Kivonat

Napjainkban egyre több, egyre komplexebb elosztott kiberfizikai rendszer jelenik meg körülöttünk: egymással kommunikáló járművek, szenzorhálózatok, okos otthonok stb. Ezen rendszerek tervezésekor fontos figyelembe venni a funkcionális követelményeken felül különböző extrafunkcionális követelményeket is, ilyenek például a teljesítmény, az energiafogyasztás és a rendelkezésre állás. Biztonságkritikus felhasználási területeken fontos a megbízhatósági követelmények teljesítése is. Ezen követelmények alapvetően kvantitatívak, azaz különböző mérőszámokra határoznak meg elérendő célértékeket. Teljesülésük biztosításával már a rendszertervezés fázisában is foglalkozni kell, amikor még nem áll rendelkezésre a működő rendszer, csak annak modellje.

A különböző extrafunkcionális metrikák meghatározásához a rendszer viselkedésére jellemző véletlenszerűséget tartalmazó, sztochasztikus modelleket használunk. A munkámban egy elterjedt sztochasztikus modellezési formalizmust, az általánosított sztochasztikus Petri-hálókat vizsgálom. Ez a formalizmus alkalmas az aszinkron elosztott rendszerekre jellemző sztochasztikus működés leírására.

A modellből a szükséges extrafunkcionális mutatók számításához egy alacsonyabb szintű, matematikailag kezelhető analízis modellt kell származtatni. Az analízis modell elkészítésekor és elemzésekor felvetődő probléma az állapottérrobbanás: bár a magas szintű mérnöki modell még kezelhető méretű lehet, a hozzá tartozó analízis modell mérete ennek exponenciális függvénye. Így az elterjedt explicit elemzési módszerek csak korlátozottan skálázhatóak.

A probléma egy lehetséges megoldása, hogy az elemzés során megoldandó lineáris egyenletrendszert tenzorreprezentációs módszerek segítségével, tömör közelítő formában tároljuk, és a megoldást is ebben a formában keressük. Munkám során a Tensor Train (TT) formátumú reprezentáció alkalmazhatóságát vizsgáltam az általánosított sztochasztikus Petri-hálók elemzésére, melyet a szimbolikus modellellenőrzésben elterjedt döntési diagram állapottér reprezentáció segítségével állítok elő.

A számítások elvégzéséhez egy a szakirodalomból vett TT-alapú lineáris egyenletrendszer megoldó algoritmust adaptálok ezen formalizmusra, mely más területeken, mint például nagy méretű fizikai szimulációk, már jól teljesített. Sztochasztikus Petri-háló analízis területén felmerülnek olyan kihívások, amik a szakirodalomban ismert TT-alapú megoldóknál még nem lettek megvizsgálva, így az algoritmusok közvetlenül nem használhatóak fel. A javasolt algoritmust benchmark modellek segítségével értékelem ki.

Abstract

Increasingly complex distributed cyber-physical systems are becoming more and more widespread these days: vehicles communicating with each other, sensor networks, smart homes, etc. Throughout the design of such systems, various extra-functional requirements must be taken into account, such as performance, energy consumption, and availability. In the case of safety critical application areas, satisfying reliability requirements is also of utmost importance. These kinds of requirements are mostly quantitative, meaning that they give target values for some metrics of the system that must be achieved. Achieving these values must be assured already in the design phase, when no usable instance of the system under development is available for measurement, only a model describing its behavior.

The extra-functional metrics are derived using a stochastic model explicitly describing the randomness inherent in the behavior of the system. In my work, I focus on a widely used stochastic modeling formalism called generalized stochastic Petri-nets. This formalism is well suited to describe the stochastic behavior of asynchronous distributed system.

To calculate the necessary metrics from the model, a lower level analysis model must be derived from it, that can be handled mathematically. When creating and analyzing this low-level model, the problem of state-space explosion arises: even though the high-level engineering model is of tractable size, the size of the corresponding analysis model is exponential in the original one's size. Because of this, the scalability of widespread explicit analysis methods is limited.

A possible solution to this problem is storing the linear equation system that needs to be solved during the analysis in a concise approximate form using tensor representation methods, and seeking the solution in the same format. We examine the applicability of Tensor Train (TT) methods for generalized stochastic Petri-nets in this work. The proposed method uses decision diagram-based state space representation for the derivation of the compressed form, which is widely used in symbolic model checking.

We adapt a TT-based linear equation system solver, which has been successfully used in other areas, such as large-scale physics simulations, to compute extra-functional metrics of generalized stochastic Petri nets. There are challenges that arise when using the TT format for the analysis of stochastic Petri-nets that are yet unexplored in the literature for TT-based solvers, hindering the direct application of these algorithms without modification. In this work, our aim is to overcome these challenges, and to verify the proposed algorithm using benchmark models.

Chapter 1

Introduction

Increasingly complex distributed cyber-physical systems are becoming more and more widespread these days: vehicles communicating with each other, sensor networks, smart homes, etc. Throughout the design of such systems, various extra-functional requirements must be taken into account, such as performance, energy consumption, and availability. In the case of safety critical application areas, satisfying reliability requirements is also of utmost importance. These kinds of requirements are mostly quantitative, meaning that they give target values for some metrics of the system that must be achieved. Achieving these values must be assured already in the design phase, when no usable instance of the system under development is available for measurement, only a model describing its behavior.

The extra-functional metrics are derived using a stochastic model explicitly describing the randomness inherent in the behavior of the system. In my work, I focus on a widely used stochastic modeling formalism called generalized stochastic Petri-nets (GSPNs) [1]. This formalism is well suited to describe the stochastic behavior of asynchronous distributed system.

To calculate the necessary metrics from the model, a lower level analysis model must be derived from it, that can be handled mathematically. For GSPNs without proper non-determinism, this analysis model is continuous-time Markov chain. When creating and analyzing this low-level model, the problem of state-space explosion arises: even though the high-level engineering model is of tractable size, the size of the corresponding analysis model is exponential in the original one's size. Because of this, the scalability of widespread explicit analysis methods is limited.

A possible solution to this problem is storing the linear equation system that needs to be solved during the analysis in a concise approximate form using tensor representation methods, and seeking the solution in the same format. We examine the applicability of Tensor Train (TT) [22] methods for generalized stochastic Petri-nets in this work. The proposed method uses decision diagram-based state space representation for the derivation of the compressed form, which is widely used in symbolic model checking.

We adapt a TT-based linear equation system solver, which has been successfully used in other areas, such as large-scale physics simulations, to compute extra-functional metrics of generalized stochastic Petri nets. There are challenges that arise when using the TT format for the analysis of stochastic Petri-nets that are yet unexplored in the literature for TT-based solvers, hindering the direct application of these algorithms without modification. In this work, our aim is to overcome these challenges, and to verify the proposed algorithm using benchmark models.

Our aim is to provide a method which scales well with the number of state variables, which is the number of places in the case of GSPNs. The motivation for scaling on this dimension is that to take advantage of the model's structure, it must be decomposable into components with small state spaces. In the extreme case when there is only a single state variable, the Tensor Train representation is equivalent to the dense explicit representation.

Our previous results The work presented in this report is based on our previous work presented in [27], which proposed a Tensor-train based method for the analysis of static fault trees. GSPNs are a much more expressive formalism, thus the current work aims to generalize the method proposed previously.

[27] focused on the computation of the mean time until first failure metric. The theorems used for that method are general enough to be also usable in the case of mean time until absorption (MTTA) computations in any structured CTMC whose rate matrix can be efficiently represented in the TT format.

In contrast to static fault trees, the computation of the steady-state distribution is extremely complicated for large GSPNs, so we also consider its TT-based computation here.

Generalizing the previous method needs answers to the following questions:

- How can the TT representation of a GSPN's rate matrix be efficiently computed from the model?
- Which TT-based linear systems solvers can handle the TT ranks of the resulting representation, if any? What modifications can help, if no appropriate solver exists in the literature?

The answer to the first question is presented in Chapter 4. Regarding the second question, however, this is a work-in-progress report. We present some modification possibilities in Section 5.3 which we implemented, but they were not enough to solve the problem yet.

Contributions The main contribution of this report is an answer to the first and a partial answer to the second question above:

- A method is given for the computation of GSPN metrics using Tensor Trains.
- Some potential modifications are proposed for the state-of-the-art TT-based linear equation system solver AMEn in order to make it able to solve the linear system arising in the previously mentioned method. This work is still in progress.

The source code for our prototype implementation can be found on github [1].

Structure of the report Chapter 2 reviews the necessary background in stochastics and extra-functional modeling, and introduces the Tensor Train (TT) format. Chapter 3 gives a short survey of the related literature. Chapters 4 and 5 describe the main contributions of the paper: a method for using Tensor Trains for computation of GSPN metrics. Chapter 6 describes a scalable benchmark model we used in our measurements and the results of our numerical experiments. Chapter 7 concludes the work and provides a summary of some potential areas of future research in the topic.

[1] `https://github.com/szdan97/tensortrain`

Chapter 2

Background

This chapter reviews the necessary mathematical and modeling background.

2.1 Continuous-Time Markov Chains

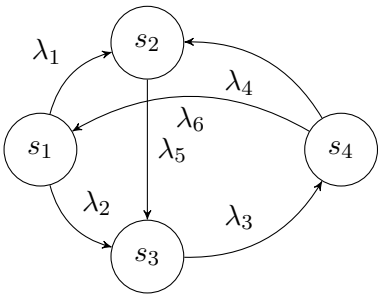
Markov Chains (MCs) form an important mathematical formalism used in stochastic modeling. Intuitively, a Markov Chain models a stateful system, where state transitions happen randomly, but with the Markov property: the future depends on the past only through the present. Markov Chains can have either a continuous or a discrete state space (although some authors refer only to those with discrete state space as Markov chains, and call continuous ones simply Markovian continuous stochastic processes), and can evolve in continuous or discrete time, leading to different types of MCs.

In reliability and performance modeling, MCs usually have a discrete state space, which can be either finite or countably infinite. For example, when dealing with queuing systems, the size of a queue can be modelled as a Markov chain with a countably infinite number of states, if the queue has infinite capacity. Most models in reliability analysis, like fault trees or Petri-nets (with a finite maximum number of tokens) yield MCs with a finite state space. The infinite case needs a very different approach to analyze than the finite case, so we focused only on MCs with a finite state space in this work.

On the time dimension, both discrete and continuous-time modeling can be a reasonable choice when dealing with extra-functional requirement verification, depending on the problem. For now, we considered only a continuous-time treatment of GSPNs for the computation of relevant metrics (see section 2.3). In a Continuous-Time Markov Chain, state transitions can happen at any point in time, but the probability of taking a given transition in a give time period depends only on the current state of the system, independent of the previous states and the time spent in the current state before the interval. The formal definition of a CTMC is as follows (from [26]):

Definition 1. A stochastic process $\{X(t), t \geq 0\}$ is a *Continuous-Time Markov Chain (CTMC)* if for all integers n and for any sequence $t_0, t_1, \ldots, t_n, t_{n+1}$ such that $t_0 < t_1 < \cdots < t_{n-1} < t_n$, the following equation holds:

$$\begin{aligned} Prob\{X(t_{n+1}) = x_{n+1} | X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \ldots, X(t_0) = x_0\} = \\ = Prob\{X(t_{n+1}) = x_{n+1} | X(t_n) = x_n\}. \end{aligned}$$



$$\begin{bmatrix} -(\lambda_1+\lambda_2) & \lambda_1 & \lambda_2 & 0 \\ 0 & -\lambda_5 & \lambda_5 & 0 \\ 0 & 0 & -\lambda_3 & \lambda_3 \\ \lambda_6 & \lambda_4 & 0 & -(\lambda_6+\lambda_4) \end{bmatrix}$$

Figure 2.1: CTMC example represented by its state graph and the corresponding infinitesimal generator matrix

This definition uses the Markov property in continuous time to define a Markov chain, but inspecting CTMCs only through this property is hardly useful. A more useful characterization of CTMCs comes from 2.1, which is described in the following.

For the sake of correctly identifying the scope of the work and the theorems used, a small aside is needed about time-homogeneity.

Definition 2. [26] A CTMC is called *time-homogeneous* if $Prob\{X(t_{n+1}) = x_{n+1}|X(t_n) = x_n\}$ depends only on x_{n+1}, x_n and the difference $t_{n+1} - t_n$. It is called *time-inhomogenous* otherwise. ▪

Analyzing time-inhomogenous CTMCs is a challenging task, they are rarely used. The CTMCs derived from GSPNs are always time-homogeneous unless the transition rates are time dependent, which is a generalization that is out of scope for this work, so we considered only time-homogeneous models. From now on, all CTMCs are implicitly assumed to be time-homogeneous.

We can assemble the *state probability vector* $\boldsymbol{\pi}(t)$, the ith element of which is the probability of being in the ith state at time t. As this is a vector made of all elements of a discrete probability distribution, $|\boldsymbol{\pi}(t)|_1 = 1$, and its elements are non-negative.

With this notation in place, it can be shown (see any textbook on the topic, like [26]), that the time of taking a transition follows an exponential distribution, and the evolution of the probabilities of being in each state can be described by the following linear differential equation, called the *Kolmogorov forward equation*:

$$\frac{d\boldsymbol{\pi}(t)}{dt} = \boldsymbol{\pi}(t)\boldsymbol{Q} \tag{2.1}$$

where $\boldsymbol{\pi}(t)$ is the state probability vector at time t, and $\boldsymbol{Q}$ is called the CTMC's *infinitesimal generator matrix*. It can be computed from the *rate matrix* of the CTMC, in which the diagonal is zero, and the off-diagonal element (i, j) is the rate (parameter) of the exponential distribution describing the amount of time after which the system transitions from state i to state j (0, if transitioning from state i to state j is not allowed). From this matrix, $\boldsymbol{Q} = \boldsymbol{R} - \text{diag}\{\boldsymbol{R} \cdot \boldsymbol{1}\}$. Each row of the infinitesimal generator sums to zero, so it is always singular. An example of a CTMC with its state graph and infinitesimal generator matrix can be seen on figure 2.1.

A CTMC has three kinds of special distributions for the state probabilities, defined as follows [26]:

Definition 3. The elements of a state probability vector $\boldsymbol{\pi}_{st}$ define a *stationary distribution* of the CTMC, if $\frac{d\boldsymbol{\pi}_{st}}{dt} = \boldsymbol{0}$. ▪

Definition 4. Given an initial state distribution with state probability vector $\boldsymbol{\pi}(0)$, the distribution defined by the elements of $\boldsymbol{\pi}_{limit} = \lim_{t\to\infty} \boldsymbol{\pi}(t)$ is called a *limiting distribution* of the CTMC, if the limit exists and defines a proper probability distribution. Limiting distributions are always stationary distributions. ▪

Definition 5. If the limiting distribution of a CTMC is independent of the initial distribution, than this distribution is called the *steady state distribution* of the Markov Chain.▪

Intuitively, the steady state distribution tells us about the probability of finding the Markov Chain in a given state after leaving it on its own to evolve for an infinite amount of time, independent of the state we left it in. This can often be a close enough approximation to the behavior of a system after an initial, not so long transient period. Because of this, computing the steady state distribution is an important task in stochastic modeling. As the steady state distribution is stationary, we know from equation 2.1, that

$$\boldsymbol{\pi}_{steady}\boldsymbol{Q} = 0, \tag{2.2}$$

meaning that the state probability vector of the steady state distribution lies in the left null space of $\boldsymbol{Q}$. This leads to two methods for finding the steady state distribution, if it exists:

a. calculate a non-zero vector in left the null space of $\boldsymbol{Q}$, and normalize it so that the elements sum to 1.

b. replace one of the columns of $\boldsymbol{Q}$ and the corresponding element on of the right hand side with the normalizing equation $\boldsymbol{\pi}_{steady}\mathbf{1} = 1$ (where $\mathbf{1}$ is a vector of ones with the appropriate size).

In the first case, we need to solve a homogeneous linear equation system, and a non-homogeneous one in the second case. Both can be solved using either direct or iterative solvers, but for structured matrix and vector representations like the Tensor Train format (see Section 2.6), only the first one is usable with most solvers.

Markov Chains are widely used in reliability and performance analysis, as several metrics and characteristics of the system can be derived by analysing them. Some examples are:

- Transient analysis, which describes the time evolution of the system, answering questions like what is the probability that the system fails throughout its lifetime, or how long the system can be operated without maintenance if a given maximum failure probability is prescribed

- Mean time to absorption, which will be described in more detail in Section 2.2, is used when calculating the mean time to failure, mean time to repair, or mean time between failures metrics for reliability analysis, or mean service time for performance analysis.

- Steady-state analysis tells us how the system approximately behaves after it is left on its own for a long amount of time

- Reward-based analysis can be used to compute metrics like expected energy consumption of a system

2.2 Phase-type Distributions

Phase-type distributions is a family of continuous probability distributions related to Markov chains. They are useful when modeling a random variable representing the amount of time passed until some event happens.

Definition 6. A state of a CTMC is called *absorbing*, if it has no outgoing transitions. This is equivalent for the all the elements of corresponding row in the infinitesimal generator to be zero. ▪

Definition 7. A *phase-type distribution* is the distribution of a random variable representing the time until absorption in a Markov chain with a single absorbing state. ▪

A phase-type distribution can be represented by the infinitesimal generator matrix and initial probability vector of the associated Markov chain. If the states of the CTMC are ordered such that the absorbing state is the last one, the infinitesimal generator and the initial probability vector can be written in the following block form:

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{S} & \boldsymbol{s}_{absorb} \\ \boldsymbol{0}^T & 0 \end{bmatrix}, \ \boldsymbol{\pi}(0) = [\boldsymbol{\sigma} \mid \sigma_{absorb}]$$

With this notation, any moment of the phase-type distributed random variable X can be computed as:

$$\mathbb{E}[X^j] = (-1)^j j! \boldsymbol{\sigma} \boldsymbol{S}^{-j} \boldsymbol{1} \tag{2.3}$$

Setting j=1 in this formula, we get the formula for the expectation of the random variable:

$$\mathbb{E}[X] = -\boldsymbol{\sigma} \boldsymbol{S}^{-1} \boldsymbol{1}$$

Computing this formula directly involves taking the inverse of $\boldsymbol{S}$, which is often computationally infeasible if the associated Markov chain has a large number of states. Alternatively, we can decompose this formula into solving the linear system $\boldsymbol{S}\boldsymbol{x} = \boldsymbol{1}$, and then computing $-\boldsymbol{\sigma}\boldsymbol{x}$, or doing the same in the other direction. Thus, the task of finding the expected value of a phase-type distributed random variable can be reduced to solving a linear system.

Although the original definition of a phase-type distribution involves a Markov chain with a single absorbing state, it can be easily shown that the time until absorption in a CTMC with more than one absorbing state is also a Phase-type distribution.

To see this, we only need to substitute all the absorbing states with a single, abstract absorbing state, and redirect all the transitions originally leading to one of the absorbing states to this state. As there are (by definition) no outgoing transitions from any of the absorbing states, merging them into a single state does not change the behaviour of the Markov chain. The time until absorption in the Markov chain with the merged absorbing state is thus the same as in the original one.

When dealing with a Markov chain with an explicitly represented state space, the absorbing states can easily be abstracted into a single state. In the case of structured representation, such as those we deal with in this work, however, this is not possible, so the extension to multiple failure states is needed. Calculation methods can also be easily extended: instead of eliminating the rows and columns of the single absorbing state, all the rows and columns corresponding to an absorbing state must be eliminated.

2.3 Stochastic Petri-Nets

Definition 8 (Stochastic Petri Net). A Stochastic Petri Net (SPN) $N = \langle \mathcal{P}, \mathcal{T}, A, R, M_0 \rangle$ is a tuple where:

- $\mathcal{P}$ is the set of *places*
- $\mathcal{T}$ is the set of *transitions*
- $\mathcal{A} = \langle A^-, A^+, A^\circ \rangle$ is a triplet of arc weight functions
  - $A^- : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ are the *input arc weights*
  - $A^+ : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ are the *output arc weights*
  - $A^\circ : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ are the *inhibitor arc weights*
- $R : \mathcal{T} \times \mathcal{M} \to \mathbb{R}^+$ is the marking-dependent *rate function*, where $\mathcal{M} = \{F \mid F : \mathcal{P} \to \mathbb{N}\}$ is the set of possible markings (a marking assigns a current number of *tokens* to each place)
- $M_0 : \mathcal{P} \to \mathbb{N}$ is the *initial marking* of the Petri net ▪

Definition 9 (SPN Semantics). The evolution of the SPN can be given in continuous time by specifying its current marking $M(t)$ at each time instant $t \in \mathbb{R}^+$. The net starts in the initial marking, so $M(0) = M_0$.

In each marking, a transition $r \in \mathcal{T}$ is *enabled*, if $\forall p \in \mathcal{P} : M(t)(p) \geq A^-(p, r) \wedge M(t)(p) < A^\circ(p, r)$. The set of enabled transitions in marking m will be denoted by $\mathcal{T}_{En}(m)$. At $t = 0$, and each time the marking of the net changes, a delay $\delta_k \in \mathbb{R}$ is sampled for each enabled transition k from an exponential distribution with parameter $R(k, M(t))$.

In the marking-independent case (when $\forall m \in \mathcal{M} : R(k, m) = R(k)$, the rate depends only on the transition), resampling the firing time on each marking-change is statistically equivalent to sampling it only when the transition becomes enabled because of the memoryless property of the exponential distribution. The same is true in the marking-dependent case for those transitions whose rate is not changed by the marking change. For those whose rate does change, however, the resampling semantics is needed to account for the changed firing time distribution.

The enabled transitions "race" against each other:

$$k_{min} = \underset{k \in \mathcal{T}_{En}(M(t))}{\arg\min} \{\delta_k\}$$

will be the transition that actually fires, as its firing changes the marking, which may enable new transitions and disable others in the net, and even for those that stay enabled the firing time is resampled. At time $t_{fire} = t + \delta_{k_{min}}$, the marking of the net changes such that $\forall p \in \mathcal{P} : M(t_{fire})(p) = M(t)(p) - A^-(k_{min}, p) + A^+(k_{min}, p)$. Between firings, the marking stays the same. ▪

Definition 10 (Prioritized Stochastic Petri Net). A *Prioritized Stochastic Petri Net (PSPN)* is a stochastic Petri net with an additional *priority function* $\Pi : \mathcal{T} \to \mathbb{N}$. The semantics are similar to SPNs, except that an enabled transition is considered for firing if and only if no transition with higher priority is enabled. ▪

Definition 11 (Generalized Stochastic Petri Net). A *Generalized Stochastic Petri Net (GSPN)* has the same components as a PSPN, but the transition set $\mathcal{T} = \mathcal{T}_t \sqcup \mathcal{T}_i$ is partitioned into two sets, the set of *timed transitions* $\mathcal{T}_t$ and the set of *immediate transitions* $\mathcal{T}_i$. The priority function must obey the constraints $\forall k \in \mathcal{T}_t : \Pi(k) = 0$ and $\forall k \in \mathcal{T}_i : \Pi(k) \geq 1$.

A Generalized Petri net marking M where no transition t with $\Pi(t) \geq 1$ is enabled (so only timed transitions are enabled) is called tangible, while markings with at least one enabled transition with $\Pi(t) \geq 1$ (so at least one immediate transition is enabled) are called vanishing. We write $M \in T$ if $M \in Sr$ is a reachable tangible marking and $M \in V$ if M is a reachable vanishing marking. In tangible markings, the timed semantics of Stochastic Petri nets apply to GSPNs. In contrast, immediate transitions are fired in vanishing markings while no time is elapsed. For immediate transitions, the function R gives their *weight* instead of rate: if more than one immediate transition can fire at the same time, one of them is chosen randomly according to a distribution where the probability of each transition is proportional to its weight.

Similarly to the transitions of PSPNs, an enabled transition in a GSPN is considered for firing exactly if no transition with higher priority is enabled.

Generally, GSPN definitions allow leaving weights for some immediate transitions unspecified. In this case, the transition is chosen non-deterministically, instead of probabilistically, from the firable ones [8]. This leads to proper non-deterministic behavior, making GSPNs a more expressive formalism than Markov chains. Generalizing the TT-based method for metric computation from Markov chains to formalisms with proper non-determinism (e.g. Markov Automata [7]) is out of this work's scope. Therefore, we consider only GSPNs with fully defined R here, so that no non-determinism can arise.

Graphical notation The standard graphical notation of GSPNs consists of the following elements:

- Circles denote places
- Empty rectangles denote timed transitions
- Filled thin rectangles (or lines) denote immediate transitions
- An arrow from a place's circle to a transition's rectangle denotes an input arc between them
- An arrow from a transition's rectangle to a place's circle denotes an output arc between them
- A line between a place's circle and a transition's rectangle with a small circle on the transition's end denotes an inhibitor arc between them
- An integer label on an arc's line denotes the arc's weight; an implicit weight 1 is assumed if not explicitly given
- The initial marking is notated by writing the number of tokens inside the circle of each place; 1-3 tokens might be notated using 1-3 filled circles inside the place instead of writing the number; leaving a place's circle empty means 0 tokens on the place

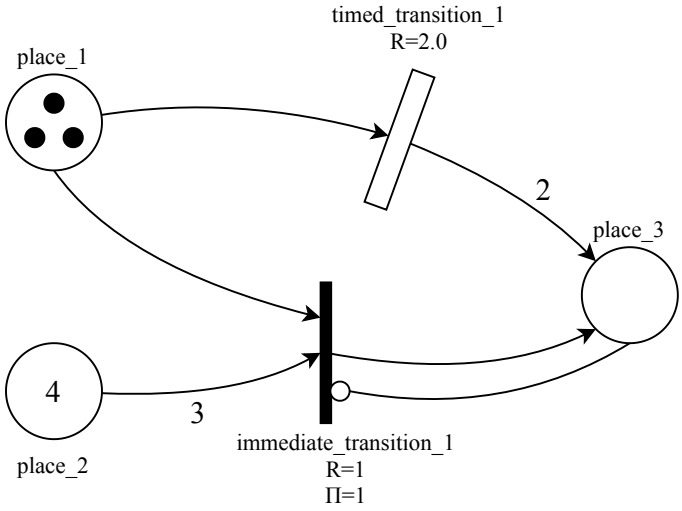


Figure 2.2: An example generalized stochastic Petri-net

- Rates, weights, and priorities of transitions, names of places and transitions can be stated by using labels next to the notational elements

A small example can be seen on Figure 2.2.

2.4 Decision Diagrams

Definition 12. A *Binary Decision Diagram* is a rooted DAG with one or two terminal nodes of out-degree zero labeled 0 or 1 and a set of non-terminal nodes of out-degree two. Non-terminal nodes are labeled with a corresponding *variable* (more than one node can be labeled with the same variable), and the edges of a non-terminal node correspond to choosing the value of the node's variable. A path from the root to a terminal node thus means choosing the values of the variables the way the edges on the path specify them.

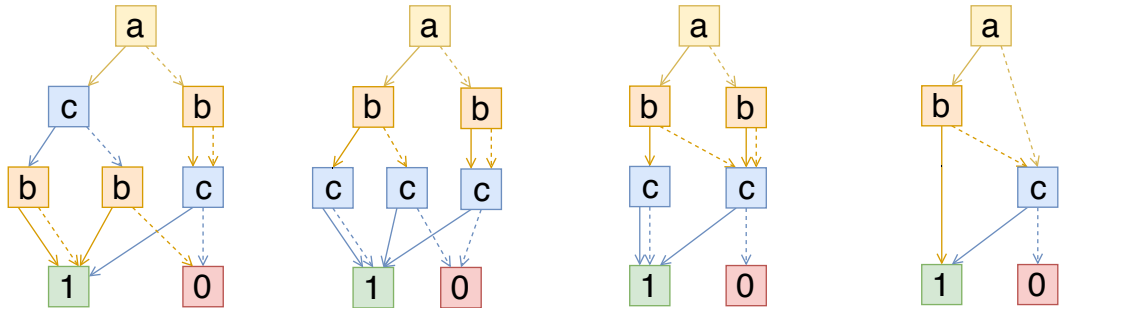
A BDD is *ordered* if on all paths through the graph, the variables respect a given linear order. In this case, the nodes of the graph are organised into levels, each of the levels corresponding to a *variable*, except for the last one, which consists of the terminal nodes.

An ordered BDD is *reduced*, if it satisfies the following properties (where $u[0]$ and $u[1]$ means the node at the end of the outgoing edge of u labeled 0 and 1, respectively):

- Uniqueness: $(\, u[0] = v[0] \wedge u[1] = v[1] \,) \implies u = v$
- Non-redundant tests: $u[0] \neq u[1]$ ▪

BDDs were originally created to represent Boolean functions in a compact format. The variables of the BDD are the inputs of the function, the edges are the values of the inputs, and the terminal node reached by choosing the appropriate edges for each node is the output of the function. A reduced ordered BDD is a canonical representation of a Boolean function this way.

Another canonical form can be given if the non-redundant tests constraint is dropped, and instead all paths leading from the root to the terminal level are required to go through exactly one node in each level. Such BDDs are called *quasi-reduced*. This practically



(a) BDD with no constraints **(b)** ordered BDD **(c)** quasi-reduced ordered BDD **(d)** reduced ordered BDD

Figure 2.3: BDDs representing the function $f(a, b, c) = (a \wedge b) \vee c$, with solid lines signifying 1 edges and dashed lines signifying 0 edges.

means that all the input variables are considered in each path, even if their value does not matter. Although this means storing redundant nodes, this form is needed for the structured representations used in this work. The computations can be implemented in such a way that while building the BDD, the more efficient fully-reduced format is used internally. Throughout the report, "BDD" refers to a quasi-reduced ordered binary decision diagram.

An example for different BDD types is given in Figure 2.3.

A subset of a state space can be represented similarly if the states can be decomposed into binary state variables: in this case, a Boolean function can be given which takes the state variables as inputs, and returns 1, if the state is included in the set, and 0 otherwise. Efficient algorithms exist for performing set operations, like union, intersection or difference, on sets represented by BDDs this way.

The set of failure states in a fault tree can be represented as a BDD using the state space decomposition described above, by using the state of the basic events as binary state variables (0 - operational, 1 - failed), and representing the Boolean function encoded by the tree's gates as a BDD.

2.4.1 Multi-valued decision diagrams

Multi-valued decision diagrams (MDDs) are an extension of BDDs, which allows specifying an arbitrary finite number of labels for each variable, and using this number of different labeled edges for the corresponding nodes instead of only the original two (zero and one) labels. This makes it possible to represent functions whose output is still Boolean, but the input variables can come from any finite domain.

MDDs can be used when the state space of the atomic components isn't binary. Generally, the places of a GSPN may contain more than a single token at a time, so MDDs are needed to represent state sets when the state variables are the number of tokens in each place.

2.4.1.1 Interval Decision Diagrams

Interval Decision Diagrams (IDDs) are a more compact form of multi-valued decision diagrams. Instead of single values, the edges are labeled with intervals such that the

intervals on the edges leaving a node are disjoint and their union equals the domain of the variable corresponding to the node. Other than being more compact than regular MDDs, they are also able to handle infinite domains using unbounded intervals.

IDDs are very suitable for the analysis of GSPNs, as enabledness of transitions depends on whether the number of tokens on a place is contained in the interval between the weight of the input arc (or 0 if it does not exist) and the inhibitor arc (or $+\infty$ if it does not exist) between the transition and the place.

Edge-valued Interval Decision Diagrams (EVIDDs) are used in [17] for computing the reachable state-space of a GSPN. Unlike regular decision diagrams which assign an output value to a valuation of the variables solely based on the terminal node at the end of the path corresponding to the valuation, edge-valued decision diagrams have values assigned to their edges, and the output value is the sum of the values corresponding to the edges on the path.

2.5 Kronecker product and multi-index notation

The Kronecker product of matrices, denoted by $\boldsymbol{A} \otimes \boldsymbol{B}$ is defined as follows:

$$(\boldsymbol{A} \otimes \boldsymbol{B})[i_1 r_{\boldsymbol{B}} + i_2, j_1 c_{\boldsymbol{B}} + j_2)] = \boldsymbol{A}[i_1, j_1] \cdot \boldsymbol{B}[i_2, j_2]$$

where $r_{\boldsymbol{B}}$ and $c_{\boldsymbol{B}}$ denote the number of rows and columns of $\boldsymbol{B}$, respectively. The Kronecker product is essentially an every-element-by-every-element product. When selecting an element of the result, it must be specified which element of each input matrix is taken. Because of this, the result has $r_{\boldsymbol{A}} r_{\boldsymbol{B}}$ rows and $c_{\boldsymbol{A}} c_{\boldsymbol{B}}$ columns.

Example 1 (Kronecker product).

$$\boldsymbol{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 4 \\ 3 & 0 & 4 & 0 \\ 0 & 6 & 0 & 8 \end{bmatrix}$$

A more natural indexing convention for the result is the *multi-index notation*: instead of a single row index and a single column index, two indices are used for each, specifying the element indices in the input matrices separately. With this a convention, an element of a matrix resulting from a Kronecker product is denoted as follows:

$$\boldsymbol{C}[(i_1, i_2), (j_1, j_2)] = \boldsymbol{C}[i_1 r_2 + i_2, j_1 c_2 + j_2)]$$

This notation can be also be used with more than two indices when the indexed matrix is the Kronecker product of more than two matrices:

$$\boldsymbol{C} = \bigotimes_{k=1}^{N} \boldsymbol{A}_k$$

$$\boldsymbol{C}[(i_1, i_2, \ldots, i_N), (j_1, j_2, \ldots, j_N)] = \prod_{k=1}^{N} \boldsymbol{A}_k[i_k, j_k]$$

Example 2 (Multi-index notation). *With $\boldsymbol{M}$ from the previous example:*

$$\boldsymbol{M}[(2,2),(2,2)] = 4 \cdot 2 = 8$$
$$\boldsymbol{M}[(2,1),(1,1)] = 3 \cdot 1 = 3$$

Treating n long vectors as $n \times 1$ matrices extends the operations naturally to vectors. The multi-index notation can be used in this case as well with only one index group:

$$\boldsymbol{v} = \bigotimes_{k=1}^{N} \boldsymbol{u}_k$$
$$\boldsymbol{w}[(i_1, \ldots, i_N)] = \prod_{k=1}^{N} \boldsymbol{u}_k[i_k]$$

Multi-index notation is also used when a vector or a matrix is not the result of a Kronecker operation, but its elements can be indexed by an index group (two index groups, in case of matrices) more naturally instead of a single index. This is the case for most of the vectors and matrices in this work. For example, the state probability vector of the underlying Markov chain of a stochastic Petri-net can be indexed by specifying the current number of token on each place, instead of specifying the index of the state directly in the product state space.

Kronecker algebra is used in two ways in this work. First, some linear algebra operations for tensor trains are implemented using Kronecker products. Second, one of the ideas for the modification of TT-based linear equation solvers examined in this work is constraining the TT-cores to be Kronecker products of an unknown matrix and a matrix derived from the structure of the model's reachable state space.

2.5.1 Kronecker Product as a Linear Operator

Let $\mathrm{vec}(\boldsymbol{X})$ denote the row-major vectorization of the matrix $\boldsymbol{X}$, meaning that $\mathrm{vec}(\boldsymbol{X})$ consists of the elements of $\boldsymbol{X}$ in the order of reading the rows after each other. There exists a matrix $\boldsymbol{K}_{\boldsymbol{A}}^{r,c}$ for any matrix $\boldsymbol{A}$ and positive integers r and c, such that

$$\boldsymbol{K}_{\boldsymbol{A}}^{r,c} \cdot \mathrm{vec}(\boldsymbol{X}) = \mathrm{vec}(\boldsymbol{A} \otimes \boldsymbol{X})$$

for every $\boldsymbol{X} \in \mathbb{R}^{r \times c}$. This means that computing the Kronecker product with a fixed matrix can be represented as a linear operator for vectorized matrices. $\boldsymbol{K}_{\boldsymbol{A}}^{r,c}$ can be easily constructed from the elements of $\boldsymbol{A}$.

2.6 The Tensor Train Format

The *tensor train (TT)* format has been defined by Oseledets in [22]. The format itself had already been used in the quantum physics community under the name *Matrix Product State* before. The decomposition is formally defined originally for *tensors*, which in this area are essentially multi-way arrays. It can also be used to represent vectors and matrices, when indexing the vector or matrix can be done using index groups, like the ones in Section 2.5 used to index the result of Kronecker operations. The vector or matrix does not have to be Kronecker structured, but the closer it is to such a structure, the more compact the tensor train representation is. The places in the index group are called *modes*.

A vector $\boldsymbol{v}$ or a matrix $\boldsymbol{M}$ represented as a tensor train is given in the following form using the multi-index notation:

$$\boldsymbol{v}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{V}_1(i_1) \cdot \boldsymbol{V}_2(i_2) \cdot \cdots \cdot \boldsymbol{V}_n(i_n)$$
$$\boldsymbol{M}[(i_1, i_2, \ldots, i_n), (j_1, j_2, \ldots, j_n)] = \boldsymbol{M}_1(i_1, j_1) \cdot \boldsymbol{M}_2(i_2, j_2) \cdot \cdots \cdot \boldsymbol{M}_n(i_n, j_n)$$

The components $\boldsymbol{V}_k$ and $\boldsymbol{M}_k$ are called *core tensors* or *cores* (they are also called tensor carriages, hence the name "tensor train", but core is more common). These are three way tensors, but they can be simply considered as arrays of matrices, and a matrix is chosen from each array by the corresponding index. Some TT articles define the cores as matrix functions with domain $\mathbb{N}$, as they take as input a natural number (two, in the case of matrices), and return a matrix. In others they are referred to as parameter-dependent matrices. Each core corresponds to a mode. The kth core consists of $r_{k-1} \times r_k$ matrices, with $r_0 = r_n = 1$, so that the result of the multiplications is a scalar. The integeres r_k are called the *ranks* or *TT-ranks* (to distinguish it from the rank commonly used in linear algebra) of the tensor train.

The format comes with efficient algorithms for basic linear algebra operations. The following list presents these. Most of these operations are applicable in the same way to vectors and matrices, so they are not presented separately. The matrix case can be reduced to the vector case by considering the kth row and column index as a single kth index (for example, a pair of a row index and a column index both with range 1 to 4 is treated as a single index with range 1 to 16).

- Product with a scalar: computing $\alpha\boldsymbol{A}$ can be simply done by multiplying each matrix in the first core of $\boldsymbol{A}$ by α

- Addition: given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ with the same dimensions in the TT format as $\boldsymbol{a}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{A}_1(i_1)\boldsymbol{A}_2(i_2)\ldots\boldsymbol{A}_m(i_m)$ and $\boldsymbol{b}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{B}_1(i_1)\boldsymbol{B}_2(i_2)\ldots\boldsymbol{B}_m(i_m)$, their sum $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$ can be computed in the TT format by setting its cores to:

$$\boldsymbol{C}_1(i_1) = \begin{bmatrix} \boldsymbol{A}_1(i_1) & \boldsymbol{B}_1(i_1) \end{bmatrix}$$
$$\boldsymbol{C}_k(i_k) = \begin{bmatrix} \boldsymbol{A}_k(i_k) & 0 \\ 0 & \boldsymbol{B}_k(i_k) \end{bmatrix} \; for\, k = 2, \ldots, m-1$$
$$\boldsymbol{C}_m(i_m) = \begin{bmatrix} \boldsymbol{A}_m(i_m) \\ \boldsymbol{B}_m(i_m) \end{bmatrix}$$

- Hadamard (element-wise) product: given two vectors of the same dimensions $\boldsymbol{a}$ and $\boldsymbol{b}$ in the TT format as $\boldsymbol{a}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{A}_1(i_1)\boldsymbol{A}_2(i_2)\ldots\boldsymbol{A}_m(i_m)$ and $\boldsymbol{b}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{B}_1(i_1)\boldsymbol{B}_2(i_2)\ldots\boldsymbol{B}_m(i_m)$, their Hadamard product $\boldsymbol{c} = \boldsymbol{a} \circ \boldsymbol{b}$, the cores of $\boldsymbol{c}$ in the TT format are

$$\boldsymbol{C}_k(i_k) = \boldsymbol{A}_k(i_k) \otimes \boldsymbol{B}_k(i_k),$$

  where $\otimes$ is the Kronecker product.

- The scalar product of two vectors of the same size, defined as

$$\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \sum_{i_1, i_2, \ldots, i_m} \boldsymbol{a}[(i_1, i_2, \ldots, i_m)]\boldsymbol{b}[(i_1, i_2, \ldots, i_m)]$$

can be efficiently computed in the TT format by using the following computations:

$$\boldsymbol{\Gamma}_k = \sum_{i_k} \boldsymbol{A}_k(i_k)\boldsymbol{B}_k(i_k)$$
$$\boldsymbol{v}_k = \boldsymbol{v}_{k-1}\boldsymbol{\Gamma}_k \text{ with } \boldsymbol{v}_1 = \boldsymbol{\Gamma}_1$$

then $\boldsymbol{v}_m$ is a scalar, and $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \boldsymbol{v}_m$.

- The Euclidean norm of a vector – which generalizes to the Frobenius norm in the matrix case – can be computed using the previously mentioned method for the scalar product:
$$||\boldsymbol{a}||_F = \sqrt{\langle \boldsymbol{a}, \boldsymbol{a} \rangle}$$

- Matrix-vector product: Given a matrix $\boldsymbol{M}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{M}_1(i_1)\boldsymbol{M}_2(i_2)\ldots\boldsymbol{M}_m(i_m)$, and a vector $\boldsymbol{x}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{X}_1(i_1)\boldsymbol{X}_2(i_2)\ldots\boldsymbol{X}_m(i_m)$ in the same tensor structure, their product $\boldsymbol{y} = \boldsymbol{M}\boldsymbol{x}$ can be computed in the TT format by computing its cores as
$$\boldsymbol{Y}_k(i_k) = \sum_{j_k} (\boldsymbol{M}_k(i_k, j_k) \otimes \boldsymbol{X}_k(i_k))$$

- Transposition: transposition can be done by simply rearranging the matrices in a core, so that the originally $(i_k \cdot \mathit{columns} + j_k)$th matrix of the core tensor becomes the $(j_k \cdot \mathit{rows} + i_k)$th one.

- Outer product of vectors: the outer product $\boldsymbol{X} = \boldsymbol{v}\boldsymbol{w}^T$ of two vectors represented using the above format can be computed in the TT matrix format described above using the following cores:
$$\boldsymbol{X}_k(i \cdot \mathit{columns} + j) = \boldsymbol{V}_k(i) \otimes \boldsymbol{W}_k(j)$$

- Matrix-matrix product: this operation should only be used when it is really necessary and unavoidable, as the cores of the resulting tensor train will be very large if used in an iteration step. The cores for the resulting matrix $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$ are:
$$\boldsymbol{C}_k(i, j) = \sum_{p} (\boldsymbol{A}_k(i, p) \otimes \boldsymbol{B}_k(p, j))$$

Apart from being able to represent structured matrices and vectors compactly, another advantage of the tensor train format is that the size of the representation can be automatically reduced even further if the representation does not need to be exact. This is done through the tensor train rounding algorithm. This algorithm takes as input a tensor train and a rounding tolerance, and returns a tensor train that has TT-ranks less than or equal to those of the original, and which represents a matrix/vector not further from the original one in Frobenius norm than the specified tolerance.

Given a vector $\boldsymbol{a}$ in the TT format a compressed version $\tilde{\boldsymbol{a}}$ with ε relative tolerance in Frobenius norm, meaning that $\frac{||\boldsymbol{a}-\tilde{\boldsymbol{a}}||_F}{||\boldsymbol{a}||_F} \leq \varepsilon$ can be computed using the following steps:

1. compute truncation parameter $\delta = \frac{\varepsilon}{\sqrt{m-1}}$ where m denotes the number of cores in $\boldsymbol{a}$

2. orthogonalize the tensor using QR decomposition of the cores

3. compute truncated SVD of the unfolding matrix of each core by dropping singular vector with corresponding singular value $\sigma_k < \delta$. For each core from left to right, with the truncated SVD decomposition denoted as $\tilde{\boldsymbol{U}}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{V}}$, keep $\tilde{\boldsymbol{U}}$ folded back as the new core, and merge $\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{V}}$ into the matrices in the next core before computing the next SVD.

For details and pseudo code of the algorithm, see [22].

Most iterative algorithms for solving systems of linear equations can be implemented using only these operations. The rounding operation is needed because the size of the cores of the solution vector grows in each iteration (see the formula for addition and matrix-vector product), so the iteration vector needs to be compressed using the rounding algorithm described above.

The tensor train decomposition can be considered an extension of decomposing a vector or matrix into a Kronecker product of smaller components, as a Kronecker product can be represented as a tensor train with all ranks 1. To convert the Kronecker product into a tensor train, the components of the product must be simply flattened into cores consisting of 1×1 matrices. If the Kronecker product of the matrices $\boldsymbol{C}_k$ is computed, then the (i, j)th matrix in the kth core is just $\boldsymbol{C}_k[i, j]$.

Chapter 3

Related works

Calculating reliability and performance metrics of complex system by exploiting the inherent structure has been an area of ongoing research for a long time. A well-studied method of this principle is using Kronecker algebra for decomposing the infinitesimal generator matrix of a Markov chain into smaller matrices, and using iterative linear equation system solution methods without assembling the full matrix [5, 16].

The main disadvantages of such methods are that they are not applicable generally, only for system having exactly the necessary structure, they cannot be used when the state space is so large that vectors used in the analysis must also be compressed, and they can store the system only exactly, no memory reduction can be made even when only an approximate solution is sought.

Tensor representation methods, like the Hierarchical Tucker decomposition (HTD) [9, 10] or the Tensor Train decomposition (TT) [22] can overcome these problems. These methods exploit that state probabilities and rate matrices/infinitesimal generator matrices of a CTMC can be treated as a high-dimensional tensor (essentially, a multi-dimensional array) by decomposing the state space into domains of state variables. These tensors can then be represented using structured formats (HTD or TT), which can represent any matrix or vector exactly, although if it is not well structured, the decomposition might have a large size. There are approximation and rounding algorithm which can reduce the size of these representations, if exact representation is not required, making a trade-off between size and accuracy possible.

In [2], the authors propose an algorithm for the analysis of structured Markov-chains using compact tensor representations. The proposed algorithm uses block-Kronecker decomposition for the infinitesimal generator matrix of the system, along with a balanced HTD representation for the vector. Using HTD is similar to our case using TT, however, their method uses a separate subvector represented by its own HTD for each Descartes-product-structured partition of the reachable state-space. This can mean an exponential number of subvectors in the number of variables, which we aim to avoid. This representation also makes it impossible to "share information" between the elements of the solution vector when these elements are not in the same partition of the reachable state-space. In contrast, we aim to represent the whole solution vector by a single tensor train.

The algorithm is applicable to GSPNs, as one of the presented case studies is a GSPN model. However, the places are manually grouped into subsystems, such that each state variable corresponds to multiple places, leading to potentially very large domains for the state variables, and the representation cannot take advantage of the system's structure.

These groupings are chosen in a way such that the transition priorities do not have to be taken into account when creating the representation of the system matrix (as the block-Kronecker decomposition cannot handle this well), leading to large groups. In contrast, we keep the number of tokens in each place as its separate variable, and use the capabilities of the Tensor Train representation with the help of decision diagrams to take priorities into account in the structured representation of the infinitesimal generator.

[18] presents a method based on Tensor Trains for the mean time to absorption analysis of Superposed GSPNs, which means composing multiple GSPNs using synchronizing transitions. They consider a whole GSPN as a subsystem corresponding to a state variable, and as such, each TT core has to contain the whole transition matrix of one of the composed GSPNs, making this useful only when a large system is composed of a lot of small independent GSPNs. As priorities are handled in the rate matrix of each GSPN, the structured representation need not take them into account in their case either. Our method is aimed at the analysis of a single GSPN with a large number of places.

Moreover, their presented case study uses GSPNs where there is always exactly one token in the whole net, meaning that this model does not use one of greatest powers of GPSNs that they are very well suited for the modeling of distributed systems. The model uses GSPNs like state machines with each place corresponding to a state of the system.

Another limitation is that the method works only for the MTTA computation with a single absorbing state, while we aim for any set of absorbing states that can be compactly represented by a decision diagram.

[14] presents a TT-based method for the steady-state computation of communicating Markov processes, among them Stochastic Petri-nets. These are only regular SPNs, though, without immediate transitions and transition priorities.

To our best knowledge, ours is the first work addressing the solution of Generalized Stochastic Petri Nets without the aforementioned limitations using structured tensor representations.

Chapter 4

Computing GSPN metrics using Tensor Trains

4.1 Reducing GSPN metric computations to PSPN metric computations

In this report, we consider two types of reliability metrics: mean time to absorption and steady-state distribution.

Analyzing non-confused GSPNs is most commonly done by constructing its underlying CTMC and computing the desired metrics on it. This involves dropping the vanishing states from the state space [19, 1], which is not possible when using structured representations, like Tensor Trains.

Another possibility is treating the system's dynamics as a semi-Markov process with exponential and constant zero holding times [4], and performing the analysis through its embedded DTMC. We propose using a third approach, which is similar to this, but results in a Tensor Train with smaller TT-ranks for the coefficient matrix of the linear systems that are solved in the analysis.

Our approach consists of deriving a PSPN from the GSPN by turning the immediate transitions into timed ones, analyzing the PSPN, and correcting for the time spent in vanishing states in the metric computation formulae. The rates assigned in the PSPN to originally immediate transitions is a globally chosen arbitrary "immediate rate" constant multiplied by the weight of the immediate transitions. This constant should be chosen in a way that avoids making the Markov chain stiff (which means having transition rates orders of magnitude apart), if possible. One such way is using the average of the rates of the original timed transitions. The original transition priorities and the rates of originally timed transitions are kept from the GSPN. Multiplying a global constant by the weight ensures that the transition to fire is chosen from the fireable transitions according to the same distribution as in the original GSPN. This is because of the fact that for two random variables $x \sim \text{Exp}(\lambda)$ and $y \sim \text{Exp}(\mu)$, $P(\min\{x, y\} = x) = \frac{\lambda}{\lambda+\mu}$.

One metric of interest is the *mean time until first visitation* of a set of markings in the GSPN, which is equivalent to the *mean time until absorption* in a modified version of the GSPN's underlying Markov chain, where the states corresponding to the given markings are made absorbing.

This metric of the GSPN can be computed from the infinitesimal generator $\boldsymbol{Q}$ of the derived PSPN using the following formula:

$$\mathbb{E}(T_{absorb}) = \boldsymbol{\pi}_0 \hat{\boldsymbol{Q}}^{-1} \mathbf{1}_{\mathcal{T}}$$

where $\mathbf{1}_{\mathcal{T}}$ denotes the indicator vector of the tangible states of the GSPN and $\hat{\boldsymbol{Q}}$ is the infinitesimal generator of the PSPN's underlying Markov chain with the rows and columns corresponding to absorbing states omitted. As the ith element of $\boldsymbol{\pi}_0 \hat{\boldsymbol{Q}}^{-1}$ is the expected time spent in the ith state before absorption, multiplying by $\mathbf{1}_{\mathcal{T}}$ instead of $\mathbf{1}$ corrects for pretending that time also passes in vanishing states.

This is still not usable with the TT representation, though, as we cannot drop rows and columns of the infinitesimal generator if it is represented using a TT. In [27], we presented a formula which can be used to compute the mean time until absorption of a Markov chain using structured representations. The formula is stated in Theorem 1.

Theorem 1. Let $\boldsymbol{Q}$ be the infinitesimal generator of a Markov chain $\mathcal{M}$, $\mathcal{A}$ be a set of $\mathcal{M}$'s states, $\mathcal{O}$ be the set of absorbing states in $\mathcal{M}$, and $\mathcal{M}'$ be a Markov chain that is identical to $\mathcal{M}$ except that the states contained in $\mathcal{A}$ are also absorbing. Define the *modified generator* as:

$$\bar{\boldsymbol{Q}} = \boldsymbol{Q} - \boldsymbol{Q} \cdot \operatorname{diag}\{\mathbf{1}_{\mathcal{A}}\} - \operatorname{diag}\{\mathbf{1}_{\mathcal{A}}\} \cdot \boldsymbol{Q} + \gamma \cdot \operatorname{diag}\{\mathbf{1}_{\mathcal{O}}\}$$

where γ is some non-zero constant. If $\boldsymbol{\pi}_0^T \mathbf{1}_{\mathcal{A}} = 0$, then the time until absorption in $\mathcal{M}'$ with initial distribution $\boldsymbol{\pi}_0$ can be calculated as:

$$\mathbb{E}\{t_{absorb}\} = \boldsymbol{\pi_0} \bar{\boldsymbol{Q}}^{-1} \mathbf{1} \quad .$$

Proof. See [27]. .

This means that we can compute the expected time until a state (which means a given marking in the GSPN) in a given set is visited by deriving a PSPN from the GPSN, creating a TT representation of the infinitesimal generator of the PSPN's underlying Markov chain, and than computing $\boldsymbol{\pi_0} \bar{\boldsymbol{Q}}^{-1} \mathbf{1}_{\mathcal{T}}$, where $\mathcal{A}$ in $\hat{\boldsymbol{Q}}$'s formula is the set of states whose first visitation time we are interested in.

In the special case when the GPSN has deadlocks, the underlying Markov chain already has absorbing states (the deadlock states), which constitute the set $\mathcal{O}$ in the formula. By setting $\mathcal{A} = \mathcal{O}$, we can compute the mean time until the Petri-net deadlocks.

Another important property of a GSPN in extra-functional analysis is its steady-state distribution. A lot of metrics can be computed by computing the scalar product of the steady-state distribution vector and a reward vector which assigns rewards to each state. For example, the steady-state availability of the system can be computed by using the indicator vector of the operational states as reward vector, or the steady-state mean energy consumption can be computed by using the energy consumption for a unit of time spent in each state as rewards.

These reward vectors can often be easily represented as Tensor Trains, and the scalar product can be efficiently computed for two vectors in the TT representation.

The steady-state distribution $\boldsymbol{\pi_{ss}}$ of the GPSN can be computed using the infinitesimal generator $\boldsymbol{Q}$ of the PSPN in the following way:

$$\hat{\boldsymbol{\pi}}_{\boldsymbol{ss}}\boldsymbol{Q} = \mathbf{0}$$
$$\boldsymbol{\pi_{ss}} = \frac{\hat{\boldsymbol{\pi}}_{\boldsymbol{ss}} \circ \mathbf{1}_{\mathcal{T}}}{||\hat{\boldsymbol{\pi}}_{\boldsymbol{ss}} \circ \mathbf{1}_{\mathcal{T}}||_1}$$

The probability of finding the GSPN in a vanishing state at a given time must be zero, as it spends 0 time in a vanishing state. In the PSPN, however, the holding time of vanishing states is non-zero, so the steady-state distribution of the PSPN assigns non-zero probability to these states. This is corrected by first setting all the vanishing state probabilities to zero by computing the Hadamard-product with the indicator vector of the tangible state, and renormalizing the result so that the probabilities sum to 1.

As the time spent in a given tangible marking is the same in the PSPN as in the GSPN, and the next state is chosen according to the same distribution in them when a change occurs, the proportions of the steady-state probabilities between tangible states are the same in the GSPN and the PSPN.

The TT representation of the PSPN's rate matrix can be derived as described in Section 4.2. The computation of the desired metric can then be performed using TT-based linear system solvers and basic linear algebra operations that are available in the TT format.

4.2 Representing the Rate Matrix in the TT Format

The proposed approach for creating the Tensor Train representation of the rate matrix of a PSPN consists of the following steps:

1. Compute the reachable statespace of the PSPN.
2. Derive a Tensor Train representation of each individual transition's contribution to the rate matrix without taking other transitions into account.
3. For each priority level, sum the contributions of the transitions that have the given priority, and constrain the result to those starting states where exactly the given priority transitions fire.
4. Sum the contributions of the priority levels.

The following subsections detail these steps.

4.2.1 Reachable state space computation

The reachable state space is mainly needed to know the *effective capacity* of each place, meaning the highest reachable number of tokens in the given place. This is needed so that the mode lengths of the TT can be specified even if capacities are not explicitly given on the model. We use prioritized saturation with edge-valued interval decision diagrams (EVIDDs) [17] as an efficient way of this computation. The EVIDDs used in this step can be reused in step 3.

4.2.2 Contributions of individual transitions

If the marking-dependent rate expression of a transition k is a multilinear function of functions of a single place, the contribution of the transition can be represented in the TT format the following way:

We assume that the rate expression is in the following sum-of-products form:

$$R(k,m) = \sum_i \prod_j f_{i,j}(m(p_{i,j})).$$

First, a rate vector $\boldsymbol{r}_k$ is computed in the TT format, which specifies the rate of the transition for each potential marking. This is done by creating a TT vector for each $f_{i,j}$ whose cores all consist simply of 1×1 matrices with a single 1 element, except for the core corresponding to $p_{i,j}$, whose nth matrix has $f_{i,j}(n)$ as its single element, for each n between 0 and $p_{i,j}$'s effective capacity. From these vectors, the rate vector can be computed using Hadamard products and vector addition, both of which can be done in the TT format efficiently. Although the Hadamard product could make the TT-ranks grow (the cores of the Hadamard product are the Kronecker products of cores of the operands), this is not the case here, as the TT ranks of the Hadamard product's operands are all 1. This means that the rate vector can be represented using a TT with TT-ranks not greater than the number of terms in the sum. In the marking-independent case, this means a TT with TT-ranks 1.

If the rate expression is not a function of this form, then the TT representation of the rate vector is not so simple. However, it can be calculated using any general TT decomposition technique, like TT-SVD [22] or TT-Cross [21]. TT-SVD can give an exact representation, but needs the full rate vector to be available in explicit form, which might not be possible if there are too many places. In practice, a transition's rate depends only on some places, so the explicit rate vector can be created as if only these places existed, and the TT decomposition for the relevant subset of places can be performed on this much smaller vector. The resulting TT can then be extended with cores containing appropriately sized identity matrices for the remaining places.

The contribution $\boldsymbol{R}_k$ of the transition k to the rate matrix is represented by a TT with the following cores:

$$R_k^l[i,j] = \begin{cases} \boldsymbol{r}_k^l[i], & \text{if } j = i - A^-(k,p) + A^+(k,p) \\ & \quad \wedge\, i \geq A^-(k,p) \,\wedge\, i < A^\circ(k,p) \\ \mathbf{0}, & \text{otherwise} \end{cases}$$

where $\boldsymbol{r}_k^l$ is the lth core of the rate vector and R_k^l is the lth core of the transition's contribution.

4.2.3 Contributions of priority levels

After computing the contribution of each individual transition, those that belong to the same priority level can be summed. The result must be constrained to be non-zero only on those rows, which correspond to markings where transitions with the given priority can fire. This is done by "masking" the matrix through multiplying it from the left by

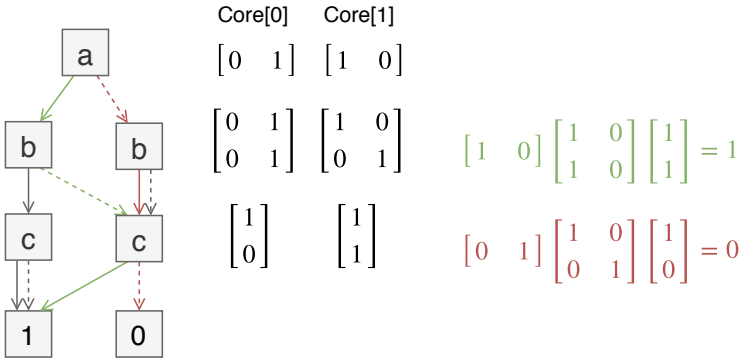


Figure 4.1: Example of the conversion between an MDD with binary variables and tensor train with two paths in the MDD and the corresponding element computation in the tensor train highlighted.

$\boldsymbol{Diag}(\mathbf{1}_{\mathcal{S}_n})$, where $\mathbf{1}_{\mathcal{S}_n}$ is the indicator vector of priority n states (markings), defined by:

$$\mathbf{1}_{\mathcal{S}_n}[i] = \begin{cases} 1, & \text{if in the } i\text{th marking, the highest priority} \\ & \text{enabled transition has priority } n \\ 0, & \text{otherwise.} \end{cases}$$

The product can be computed in the Tensor Train format if the indicator vector is also given as a Tensor Train. An efficient way to compute its TT representation is to use an MDD representing the set $\mathcal{S}_n$ with the same variable order as used for the individual rate matrix contributions. The TT representation can be derived from the MDD using the following method:

The tensor train for the indicator vector of the set described by an MDD can be assembled from the MDD by using the adjacency matrices between two levels as the matrices in the tensor cores. The matrix with index i in the core is the adjacency matrix of the i-edges between corresponding levels. The last core corresponds to the edges between the last variable's nodes and the terminal node 1.

Lemma 1. The tensor train whose cores are the adjacency matrices of the MDD representing the set $\mathcal{S}$ as described above represent the vector $\mathbf{1}_{\mathcal{S}}$. ▪

Proof. The computation of an element of the represented vector specified using the multi-index convention is done by going through the cores from left to right, choosing the matrix from the ith core according to the ith index in the index group, and multiplying these matrices. Going from left to right, the result of each multiplication before the last one has exactly one 1 in it, and the other elements are zero, because each node has exactly one i-edge.

The place of the 1 in each result specifies the index of the node chosen at that level. As the last core specifies which nodes of the last non-terminal level are connected to the 1 terminal node, the result of the last multiplication is 1 if the chosen path ends at the 1 terminal node and 0 otherwise. Figure 4.1 shows an example of this conversion. ▪

The MDD representation of a given priority state set can be efficiently obtained, for example, from the EVIDD used for representing the highest priority enabled transition in each marking in step 1 [17]. In our implementation, the priority MDDs are constructed from

the EVIDDs in such a way that they represent only reachable markings. This means that none of the contributions has non-zero elements on either the rows or the columns of unreachable markings. The rows are directly masked, and the columns would have non-zero elements only in those rows which are masked, as else the corresponding marking would be reachable from a reachable marking, making it also reachable. In explicit representations, the rate matrix is created only for the reachable states, but the TT representation can work only with product state spaces, so a lot of unreachable states are also contained in the represented matrix.

4.2.4 Summation

The result is then computed by summing the contributions of each priority level:

$$\boldsymbol{R} = \sum_{n} \left(\boldsymbol{Diag}(\mathbf{1}_{\mathcal{S}_n}) \sum_{\substack{k \in \mathcal{T} \\ \Pi(k)=n}} \boldsymbol{R}_k \right)$$

where n ranges through the priorities.

In the marking-independent case, this results in a Tensor Train with TT-ranks equal $\sum_p$ ([size of $\mathcal{S}_p$'s MDD] $\cdot$ [# priority p transitions]).

Chapter 5

Solving the linear systems using Tensor Trains

5.1 Overview of the TT-based Computation

The previous chapter presented a way to use a PSPN's rate matrix to compute metrics for a GSPN it was derived from, and an algorithm that constructs a Tensor Train representation of this rate matrix from the PSPN model. The formulae of the computations involve the solution of a linear system whose coefficient matrix is either the infinitesimal generator, computed from the rate matrix as $\boldsymbol{Q} = \boldsymbol{R} - \mathrm{Diag}(\boldsymbol{R} \cdot \mathbf{1})$, or another modified matrix computed using the formula from Theorem 1, both of which can be computed while staying in the Tensor Train representation.

The construction of the TT representation makes the rows and columns of the rate matrix corresponding to the unreachable states zero. There are two options regarding these:

- They can remain full zero, which means that the solution vector can take any value on the non-reachable states. The advantage of this is that the TT-ranks of the solution can be smaller. On the other hand, this introduces a lot of new dimensions to the null-space, which can make the solution algorithm unstable.

- The diagonal of these states can be filled with a dummy value. This constrains the exact solution to be zero on the unreachable states, removes the null-space directions introduced by the reducibility of the potential state-space. However, this means that the solution technically has to contain the information of what the reachable state space is, making the TT-ranks needed to accurately represent it much larger.

Tensor Train rounding using TT-SVD (see Section 2.6) may be applied after this to make the cores smaller, speeding up the subsequent computations. Unfortunately, in the case of large cores, this operation itself can become the bottleneck of the process, which we observed in our experiments.

The resulting matrix must be given to a TT-based linear equation solver as input along with the right-hand side of the system, which provides a Tensor Train representation of the system's solution as output. This chapter discusses one such a solver that can be considered state-of-the-art and the best scaling among its kind, and some modifications to it that we implemented in order to improve its performance.

5.2 The Alternating Minimal Energy (AMEn) solver

5.2.1 Alternating Least Squares for Tensor Trains

State-of-the art iterative linear equation solvers for Tensor Trains are based on the *Alternating Least Squares* approach proposed in [23] and [12]. The main idea of this method is optimizing each core tensor individually, while fixing the others. Although the global optimization problem is highly non-linear, the local problem involving the optimization of only a single core can be reduced to the solution of a small (compared to the global system) linear system of equations.

The ALS scheme consists of *sweeps*, and each sweep consists of optimizing each core in sequence. The stopping criterion can be checked after each update, but as its computation might be expensive, it may be a better choice to check it only after each sweep.

The vectors which can be represented by Tensor Trains with all but the kth core equal the cores of the current solution $\boldsymbol{x}$ constitute a subspace. The local system for the optimization of the kth core is mathematically derived from the global system using the kth *frame matrix* $\mathcal{X}_{\neq k}$ of the current solution, whose columns generate this subspace.

Let $\mathrm{vec}\left(C^{(k)}\right)$ for a core tensor $C^{(k)}$ denote its row-major vectorization, meaning that it is a vector constructed by stacking the row-major vectorizations of the matrices in the core below each other in increasing order of their index. The frame matrix is constructed such that $\mathcal{X}_{\neq k}\mathrm{vec}\left(X^{(k)}\right) = \boldsymbol{x}$, so the frame matrix can be used to translate between vectorizations of the kth core and the vector represented by the full Tensor Train.

The frame matrix is very large for large systems, as its number of rows equals the size of the global system (which the size of the potential state space in our application), so explicit computations with the frame matrix are intractable. However, its number of columns is small (equal to the number of elements in the kth core), and matrix-vector products involving the transpose of the frame-matrix can be efficiently implemented if the vector is also in the Tensor Train format using the cores of the Tensor Trains (see [23] for details of the formula and efficient implementation with reshapings and matrix products).

The local systems might be small compared to the global system, but they can still be quite large if the solution is sought with high TT-ranks for the sake of accuracy. In practice, when the local systems become large, they are solved using iterative linear equation system solvers instead of direct solvers, like GMRES or BiCGStab [24]. Our implementation uses BiCGStab for the local systems.

5.2.2 Rank-adaptive ALS using local enrichments

Simple ALS has the disadvantage of being fixed-rank, so the ranks of the solution have to be estimated a priori. There is no general technique to do that, so a better approach is making ALS able to adapt the rank of the solution. [23] and [12] proposed equivalent algorithms, called DMRG in one and MALS in the other for rank adaptation: instead of optimizing a single core in one ALS step, the core to optimize and the next core are merged into a "supercore", this supercore is optimized, and then decomposed again into two cores using a rank-revealing dyadic decomposition, like SVD.

Dolgov et al. proposed an algorithm in [6] called *Alternating Minimal Energy (AMEn)*, which has better scaling properties, as it does not need the optimization of potentially large supercores, and is less prone to stagnation.

The main idea of AMEn is applying an enrichment to the cores after optimization, so that the optimization of the next core can work with a larger subspace. This enrichment could be random, but a much better way is using a steepest descent scheme, which means adding the direction of the residual to the subspace. The enrichment is applied such that the current solution approximation does not change, so when kth core is extended with the enrichment columns, the next core is extended with full zero rows. This way the next ALS step can use these directions as well by filling the previously zero rows with optimized values.

There are different variants of AMEn based on how the enrichment is computed from the residual, and the best-scaling version is AMEn-ALS. In this version, a low-rank approximation to the residual with fixed TT-ranks equal to the enrichment size is maintained and updated through an auxiliary ALS iteration after each ALS step on the solution.

The auxiliary ALS iteration is an ALS approximation for $\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}$, so unlike ALS approximation for the solution of a linear system, it involves only projections to update the current core, no solution of a local linear system is needed. The kth core of the approximate residual is updated after the optimization of the kth core. The enrichment is computed by projecting the approximate residual to the subspace of vectors that can be represented with Tensor Trains whose first $k-1$ cores are the same as the already optimized one. This means that we use a frame matrix in this computation for which the first $k-1$ cores come from the current solution's TT, and the remaining cores come from the residual approximation. This projection gives the enrichment columns which are appended to the optimized core.

Algorithm 1: Alternating Minimal Energy (AMEn)

Data: $\boldsymbol{A}$ (coefficient matrix), $\boldsymbol{y}$ (right-hand side) and $\boldsymbol{x}_0$ (initial guess) in the TT format, $\varepsilon \in \mathbb{R}_+$, $r \in \mathbb{N}$
Result: $\boldsymbol{x}$ in the TT format, such that $||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}|| < \varepsilon$
$\boldsymbol{x} \leftarrow \boldsymbol{x}_0$;
$\boldsymbol{d} \leftarrow$ #cores in $\boldsymbol{x}$;
$\boldsymbol{z} \leftarrow$ random TT vector with all TT-ranks k;
while $||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}|| > \varepsilon$ **do**
  make the TTs of $\boldsymbol{x}$ and $\boldsymbol{z}$ right-orthogonal through QR factorizations;
  for $k = 1$ **to do**
    update $X^{(k)}$: solve $\mathcal{X}_{\neq k}^T \boldsymbol{A} \mathcal{X}_{\neq k} \mathrm{vec}(X^{(k)}) = \mathcal{X}_{\neq k}^T \boldsymbol{y}$;
    use SVD to remove redundant ranks of $X^{(k)}$;
    update $Z^{(k)}$: $\mathrm{vec}(Z^{(k)}) = \mathcal{Z}_{\neq k}^T \boldsymbol{y} - \mathcal{Z}_{\neq k}^T \boldsymbol{A} \mathcal{X}_{\neq k} \mathrm{vec}(X^{(k)})$;
    if $k < d$ **then**
      $\mathcal{W}_{\neq k} \leftarrow k$th frame matrix of a tensor train having the $k-1$ left cores of $\boldsymbol{x}$ at the beginning and the $d-k$ right cores of $\boldsymbol{z}$ at the end;
      compute enrichment core $W^{(k)}$:
      $\mathrm{vec}(Z^{(k)}) = \mathcal{W}_{\neq k}^T \boldsymbol{y} - \mathcal{W}_{\neq k}^T \boldsymbol{A} \mathcal{X}_{\neq k} \mathrm{vec}(X^{(k)})$;
      add the columns of $W^{(k)}$ to $X^{(k)}$;
      add r full zero rows to $X^{(k+1)}$;
    end
  end
end

5.2.3 Using AMEn for steady-state

When computing the steady-state of a continuous-time Markov chain, a non-zero vector of the left null-space of the infinitesimal generator. When using simple explicit representations, this is mostly done by substituting one row in the equation system with a normalization constraint that the elements of the solution sum to one.

In the case of structured representations, this is not possible, as a single row cannot be replaced by the normalization constraint. However, in the case of the ALS scheme, the normalization can be included in the local systems, as was done in [14]. A normalizer vector can be computed for the local system, whose scalar product with the local solution vector gives the sum of the elements in the global solution vector. This is added as a new row to the local system's coefficient matrix, and a 1 is appended to the right-hand side of the local system. In order to make the local system square, the normalizer vector extended with a 0 is also added as a new column.

In our case, a modification is needed for the normalization if the solution is kept unconstrained on the unreachable states. In this case, it may be better to make the solution on the reachable state-space sum to one, instead of the whole vector, as the reachable state-space can be orders of magnitude smaller than the potential, so renormalization after computing the solution can make the remaining error very large. This can be done by changing the normalizer vector: when computing the normalizer, instead of using the currently not optimized cores of the solution, we use their Kronecker products with the cores of the reachable state indicator vector $\mathbf{1}_{\mathcal{R}}$, and then multiply the result from the right with the linear operator representation of $\mathbf{1}_{\mathcal{R}}$'s core corresponding to the currently optimized core.

This modification is not needed when using the Constrained AMEn algorithm presented in Section 5.3.3, as in that case the solution is constrained to have non-zeroes only on the reachable states.

5.3 Improvement ideas

5.3.1 TT-SVD with iterative solvers

Based on some measurements, the main bottleneck of the computation process seemed to be the TT-SVD-based rounding of the infinitesimal generator applied before giving the matrix to the linear solver, as the cores created using the algorithm presented in Section 4.2 are too large, making full SVD computation very slow. [21] proposes another method for TT rounding called TT-cross in place of TT-SVD, but this method is not applicable in our case: the TT-ranks must be specified beforehand, instead of adaptively choosing them based on a threshold, and it can easily get stuck in local optima, which might not approximate the original matrix well. This is not so much of a problem in the area it was originally proposed for, where TTs represent discretizations of continuous functions and operators, but our case involves adjacency matrices of graphs corresponding to the reachable state space, so cross approximation could lead to dropping important states from the reachable set.

The cores appear to be well compressible, but full SVD computation is still slow, as it computes all the singular values and vectors at the same time, even if only some of them are needed. For this reason, our first idea was to compute only the largest singular values and the corresponding singular vectors using an iterative eigenpair computation algorithm, like

the power method or the Lanczos algorithm (see for example [25] for a detailed discussion of these and other iterative algorithms for eigenpair computation).

Such algorithms can be used by computing the dominant eigenpair of a matrix, and than applying deflation techniques to derive a new matrix whose dominant eigenpair is the dominant among the remaining eigenpairs of the original. These steps are iterated until enough pairs are computed so that the truncated decomposition approximates the original matrix within a given threshold.

The trace of a matrix equals the sum of its eigenvalues, and it can be easily and quickly computed. This makes it possible to know the sum of the remaining eigenvalues without computing all of them. Thus, we can use it to decide when to stop the computation, by subtracting each computed eigenvalue from the trace, and comparing the result to the rounding threshold.

Computing the truncated SVD of a matrix $\boldsymbol{A}$ with rounding error threshold ε involves computing the truncated eigendecompositions of $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}^T$ where the eigenvalues under ε^2 can be dropped.

Unfortunately, as truncated SVD computation with ε threshold needs eigenpair computation with ε^2 threshold, the eigenpair computation can fail to converge to the eigenvalues precisely enough, which we observed for both the power method and the Lanczos algorithm. If the error of the large eigenvalues exceeds any of the smaller eigenvalues that is still needed, than deflation fails to make that eigenvalue dominant when that eigenvalue comes next. As the large eigenvalues can be orders of magnitude away from the smaller, but still needed ones, the iterative algorithms are often not able to reduce the error enough.

5.3.2 Sparse AMEn-ALS

Another possibility to do away with the rounding bottleneck is to omit rounding the system matrix altogether. This solves another problem as well: when rounding is applied, the threshold must be specified somehow such that the solution of the rounded system is close enough to the solution of the original one, which is far from trivial.

The AMEn-ALS variant of AMEn, which uses an auxiliary ALS iteration for approximating the residual with low TT ranks does not need the cores of the system matrix to be given explicitly, only as abstract linear maps. This means that the sparsity of the TT cores can be utilized to make the computation faster, and our approach for deriving the TT representation results in very sparse cores. By implementing every operation with the cores this way, we can also avoid ever explicitly storing matrices with as many rows and columns as the ranks of the system matrix.

One problem here is that even though the iterations themselves might be fast, the residual norm computation after each sweep, which is used for stopping becomes the new bottleneck. As norm computation of Tensor Trains involves computing the Kronecker products of cores with themselves, even if this is also implemented using abstract linear maps, the resulting vector can easily become so large that even computing matrix-vector products with them becomes slow.

For systems with a small reachable state space, this can be solved by explicitly iterating through the reachable states, computing the corresponding elements of the residual from the tensor trains, and summing their squares. However, this scales linearly with the size of the reachable state space, which is often exponential in the number of state variables

(number of places, in the case of GSPNs), and this is exactly what we want to avoid by using structured representations.

Another possibility to make the residual computation faster is using ALS-based matrix-vector products [20]. The problem with this is that this algorithm is prone to get stuck in local optima, and it is hard to decide when to stop the iteration computing the approximate matrix-vector product.

A seemingly obvious solution would be to use the ALS-based low-rank residual approximation used for the enrichments to approximate the norm of the exact residual. Unfortunately, we observed the norm of the approximation to be sometimes even orders of magnitude smaller than the real residual, especially in later sweeps.

Another problem is that the AMEn iteration was observed to diverge or stagnate. Often, the local iterative solvers were not able to converge to a local solution, and even by using direct solvers even for large local systems, the global iteration was observed to stagnate or diverge.

5.3.3 Local Kronecker constraints

To solve the convergence issue, we created a variant of AMEn-ALS which is constrained to provide a solution that is non-zero only on the reachable states. This is achieved by representing the solution cores as the Kronecker product of an unknown core and the constraint cores, which is the corresponding core of the indicator vector of the reachable state space. The motivation for this is the following:

- If the linear system is already constructed in such a way that it forces the solution to have zeros on the non-reachable states, than the constrained optimisation can be viewed as providing a priori information to the solver. As we represent the solution cores as Kronecker products of an unknown vector and a constraint core, and we know that the exact solution can be represented this way, the solver does not have to fit this information in the optimized variables, and can achieve a better solution in less iterations.
- If the linear system is constructed without restricting the solution on the non-reachable states, and the global system is singular (in our case often with a high-dimensional null space), than the local systems. are also singular. This can heavily degrade the performance the local iterative solvers. Krylov-subspace methods, for example, tend to oscillate between two solution directions with ever increasing solution norm if the nullspace of the system matrix is multi-dimensional. By using Kronecker constraints, we give a smaller system to the local solvers, which includes less redundant variables.

The modified version of AMEn-ALS computes the solution $\boldsymbol{x}$ of the linear system in the form of a Hadamard product $\boldsymbol{c} \circ \boldsymbol{x_l}$ of the constraint vector and a low-TT-rank representation, which is equivalent to having each core have the form of a Kronecker product.

When optimizing the ith core of the solution, the local system to solve becomes $\boldsymbol{A}_i \boldsymbol{K}_{\boldsymbol{C}^{(i)}} \boldsymbol{x}_{\boldsymbol{l}}^{(i)} = \boldsymbol{y}_i$, where $\boldsymbol{A_i}$ and $\boldsymbol{y}_i$ are the same local matrix and local right-hand side as in regular ALS, $\boldsymbol{K}_{\boldsymbol{C}^{(i)}}$ is the linear operator representation of the Kronecker product with the ith constraint cores (see Section 2.5), and $\boldsymbol{x_l}$ is the vectorization of the locally optimal core of the solution's unknown part. When computing the projected matrix and right-hand side, the cores of the full solution, not just the unknown part must be used.

As this system is rectangular, the local iterative solvers must use its normal equation as they work only with square systems.

The enrichments are applied to the low-rank unknown part. The ALS-based residual approximation is also kept in the same representation as the solution, so that its low-rank part can be used to compute the enrichment for the low-rank part of the solution.

In contrast to the regular AMEn-ALS, the full solution and residual cannot be kept TT-orthogonal here, only the low-rank parts, so the columns of the frame matrices are not orthogonal. This means that in the computation of the approximate residual's update and the enrichment, the pseudo-inverse of the frame matrix must be used instead of simply its inverse. If the frame matrix has full column rank, computing the product of its pseudo-inverse with a vector presents no computational problem, as both $\mathcal{X}_{\neq i}^T \mathcal{X}_{\neq i}$ and $\mathcal{X}_{\neq i}^T \boldsymbol{y}$ can be efficiently computed using the TT cores if $\mathcal{X}_{\neq i}$ is a frame matrix. The formula for this is the following:

$$
\begin{aligned}
\boldsymbol{L}_0 &= 1 \\
\boldsymbol{R}_{d+1} &= 1 \\
\boldsymbol{L}_i &= \boldsymbol{L}_{i-1}(\sum_j X^{(i)}[j] \otimes X^{(i)}[j]) && \text{for all } 0 < i < k \\
\boldsymbol{R}_i &= (\sum_j X^{(i)}[j] \otimes X^{(i)}[j])\boldsymbol{R}_{i+1} && \text{for all } k < i < d+1 \\
\mathcal{X}_{\neq k}^T \mathcal{X}_{\neq k} &= \boldsymbol{I}_{m_k} \otimes (reshape(\boldsymbol{L}, r_{k-1}, r_{k-1}) \otimes reshape(\boldsymbol{R}, r_k, r_k)),
\end{aligned}
$$

where d is the number of cores, r_{k-1} and r_k are the $k-1$th and kth ranks of the Tensor Train respectively (equivalent to the number of rows and columns of the kth core), and m_k is the length of the kth mode.

The frame matrices may not always have linearly independent columns, as masking the unreachable states might make some ranks redundant. However, as the frame matrices are as large as the potential state space, which is too large to explicitly store in the memory for large models, general pseudo-inverse formulas that use rank factorization or QR factorization are not applicable. Because of this, we resort to a heuristic approach in this case by using the same formula, but substituting the pseudo-inverse for the inverse. As $\mathcal{X}_{\neq i}^T \mathcal{X}_{\neq i}$ is small, its pseudo-inverse can be computed using standard techniques. The ALS-based residual approximation is already a heuristic approach in itself, and using this is still better than using random enrichments.

Unfortunately, the Constrained AMEn algorithm was still observed to stagnate when used with small enrichment ranks (1 or 2) or diverge when used with larger enrichment ranks (3 or 4) on our basic benchmark problem.

Algorithm 2: Constrained Alternating Minimal Energy (CAMEn)

Data: $\boldsymbol{A}$ (coefficient matrix), $\boldsymbol{y}$ (right-hand side), $\boldsymbol{c}$ (constraint vector) and $\boldsymbol{x}_0$ (initial guess) in the TT format, $\varepsilon \in \mathbb{R}_+$, $r \in \mathbb{N}$

Result: $\boldsymbol{x} = \boldsymbol{c} \circ \boldsymbol{x}_l$ in the TT format, such that $||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}|| < \varepsilon$

$\boldsymbol{x}_l \leftarrow \boldsymbol{x}_0$;

$\boldsymbol{d} \leftarrow$ #cores in $\boldsymbol{x}$;

$\boldsymbol{z}_l \leftarrow$ random TT vector with all TT-ranks k;

`/* if` $\boldsymbol{A}$ `has zero columns where` $\boldsymbol{c}$ `is zero, the Hadamard product computation can be dropped */`

while $||\boldsymbol{A}(\boldsymbol{c} \circ \boldsymbol{x}_l) - \boldsymbol{y}|| > \varepsilon$ **do**

  make the TTs of $\boldsymbol{x}_l$ and $\boldsymbol{z}_l$ right-orthogonal through QR factorizations;

  for $k = 1$ **to do**

    update $X_l^{(k)}$: solve $\mathcal{X}_{\neq k}^T \boldsymbol{A} \mathcal{X}_{\neq k} \boldsymbol{K}_{\boldsymbol{C}^i} \mathrm{vec}(X_l^{(k)}) = \mathcal{X}_{\neq k}^T \boldsymbol{y}$;

    use SVD to remove redundant ranks of $X_l^{(k)}$;

    $\boldsymbol{Z}^+ \leftarrow (\boldsymbol{K}_{\boldsymbol{C}^i}^T \mathcal{Z}_{\neq k}^T \mathcal{Z}_{\neq k} \boldsymbol{K}_{\boldsymbol{C}^i})^\dagger$;

    update $Z_l^{(k)}$: $\mathrm{vec}(Z^{(k)}) = \boldsymbol{Z}^+ (\boldsymbol{K}_{\boldsymbol{C}^i}^T \mathcal{Z}_{\neq k}^T \boldsymbol{y} - \boldsymbol{K}_{\boldsymbol{C}^i}^T \mathcal{Z}_{\neq k}^T \boldsymbol{A} \mathcal{X}_{\neq k} \boldsymbol{K}_{\boldsymbol{C}^i} \mathrm{vec}(X^{(k)}))$;

    if $k < d$ **then**

      $\mathcal{W}_{\neq k} \leftarrow k$th frame matrix of a tensor train having the $k-1$ left cores of $\boldsymbol{x} = \boldsymbol{c} \circ \boldsymbol{x}_l$ at the beginning and the $d-k$ right cores of $\boldsymbol{z} = \boldsymbol{c} \circ \boldsymbol{z}_l$ at the end;

      $\boldsymbol{W}^+ \leftarrow (\boldsymbol{K}_{\boldsymbol{C}^i}^T \mathcal{W}_{\neq k}^T \mathcal{W}_{\neq k} \boldsymbol{K}_{\boldsymbol{C}^i})^\dagger$;

      compute enrichment core $W^{(k)}$:

       $\mathrm{vec}(Z^{(k)}) = \boldsymbol{W}^+ (\boldsymbol{K}_{\boldsymbol{C}^i}^T \mathcal{W}_{\neq k}^T \boldsymbol{y} - \boldsymbol{K}_{\boldsymbol{C}^i}^T \mathcal{W}_{\neq k}^T \boldsymbol{A} \mathcal{X}_{\neq k} \mathrm{vec}(X^{(k)}))$;

      add the columns of $W^{(k)}$ to $X^{(k)}$;

      add r full zero rows to $X^{(k+1)}$;

    end

  end

end

Chapter 6

Evaluation

6.1 The long Kanban model

Our main benchmark model is a modified version of the commonly used Kanban model [3]. The original model is the GSPN model of a Kanban system with four phases, where the two middle phases can run in parallel. It has 16 places, 14 timed transitions and 2 immediate transitions.

The scaling of the original model is done by scaling the number of initial tokens on the "kanban" places, which model the number of available resources for each phase. This means scaling the effective capacity of the places, thereby expanding the domain of the state variables.

Our approach, however, is meant to scale on a different dimension: our aim is to efficiently analyse systems with a large number of variables, not a few variables having large domains. For this reason, we use a "long Kanban model", which means putting multiple copies of the original four-phase model (we call one copy a block) after each other. Figure 6.1 shows the resulting model.

We will use Kanban(M, N) to denote the long Kanban model with M blocks and N initial tokens on the kanban places.

To benchmark the MTTA computation, we compute the mean time until the first token is placed on the output place of the last block.

6.2 Evaluation results

Here, we discuss the results of our numerical experiments.

Regular AMEn with regular TT rounding The algorithm started to struggle with the rounding of the TT matrix for Kanban(2,2) and larger models. The rounding took some minutes for Kanban(2,2), and then AMEn seemed to converge for the steady-state computation in a single sweep. The rounding timed out for Kanban(3,1).

When the steady-state computation was performed with normalizing the whole solution and leaving the solution on the unreachable state space unconstrained, the algorithm seemed to converge in one sweep, which was very suspicious. The size of the reachable state space is orders of magnitude smaller, than the potential state space, but all of the

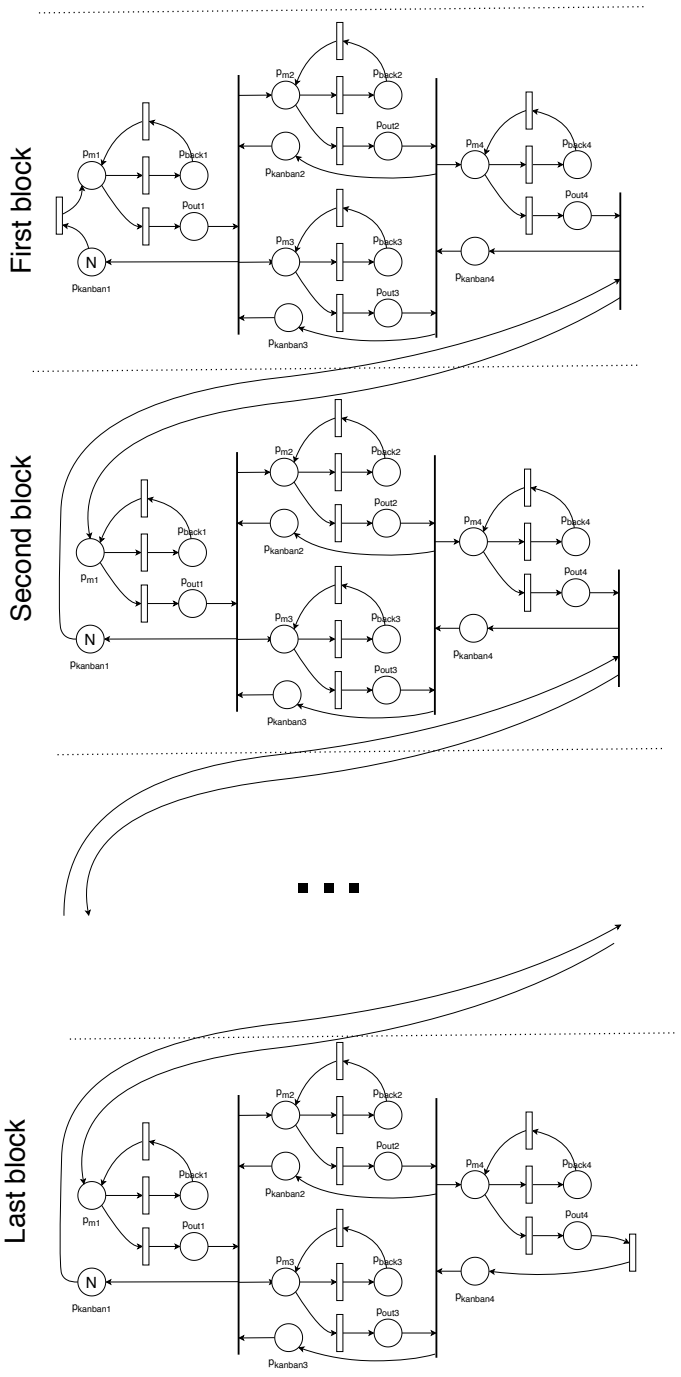


Figure 6.1: The long Kanban model

error is concentrated on it as all the other elements are multiplied only by 0. This means that renormalizing would make the error unacceptably large.

Increasing the non-reachable elements does not increase the residual norm. Therefore, the solver can make the residual arbitrarily small by making the reachable elements arbitrarily small, while increasing the non-reachable elements to account for the normalization constraint. This leads to a false solution.

As rounding was already a huge bottleneck, and we could not speed it up enough to make regular AMEn usable (see the next paragraph), we did not perform regular AMEn experiments with the false solution problem taken care of.

TT rounding with iterative eigensolvers Neither the power method nor the Lanczos algorithm was able to converge to a threshold small enough that the global relative rounding error is kept below 10^{-10}. We chose this threshold so that the rounding error can be near the same order as the errors of floating-point representation. Higher threshold would need to be justified by some kind of perturbation analysis which we were not able to do without knowing the condition number of the matrix.

The main problem seems to be that the dominant eigenvalues are orders of magnitude larger than the smaller, but still just needed ones, making it hard for the solver to approximate the large ones accurately enough for their error not to dominate the small ones.

Sparse AMEn We tried two ways to avoid the false solution:

- Filling the diagonal elements corresponding to the unreachable states with 1s to constrain the elements corresponding to unreachable states to be 0.
- Taking only the elements corresponding to reachable states into account for the normalization.

Even though these solved the problem of converging to a false solution, the sparse AMEn solver was not able to solve them even for Kanban(1,1). The first version was not able to converge in enough sweeps and timed out, as the computations in later sweeps become very demanding when the TT-ranks of the solution grow too large. In the second version, the elements of the cores started to diverge and grow towards infinity, most likely because of the null-space of the system matrix having too many dimensions.

Apart from these problems, the computation of the residual became a bottleneck from the third sweep on because of the large ranks of the system matrix, which is multiplied by the solution rank when computing the exact residual.

Constrained AMEn Unfortunately, the constrained AMEn algorithm was able to solve the system for neither the steady-state nor the MTTA computation for even Kanban(1, 1). The solver started to stagnate after some sweeps when setting the enrichment rank to 1 or 2, meaning that it was not able to take advantage of the newly introduced ranks well enough, and they were dropped after the local optimization. The residual norm even started to oscillate when using 3 or 4-rank enrichments and the solution was not able to converge. Seeking deeper understanding of this behavior is part of our further research plans.

Chapter 7

Conclusions and future work

This work proposed a method for overcoming the problem of state space explosion when computing quantitative metrics for generalized stochastic Petri-net models, which are used in practice for the modeling of extra-functional properties of distributed systems.

The work consisted of two parts:

- In Chapter 4, we presented a method for deriving a Tensor Train from a GSPN model, that can be used to compute steady-state and mean time metrics for the model, and provided a mathematical description of how to compute them using the Tensor Train.
- The proposed method involves solving a linear equation system using the Tensor Train representation, which can be challenging in the case of large and complex models. In Chapter 5, we presented some possible modifications for a state-of-the-art TT-based linear equation solver algorithm to make it work in our computation algorithm. This part of the work is still in progress, as our currently implemented modifications were not able to solve the problems arising in the computation yet.

Chapter 6 discussed the evaluation of the proposed algorithm on a scalable benchmark model.

Apart from the theoretical contribution, our open-source prototype implementation[1] is also available to use for further research.

7.1 Future work

As the question of how to solve the large linear system in the proposed algorithm is still open, this is the most important direction for further research.

One possibility we plan to investigate is using "don't care"-based compaction [13] for MDDs in the derivation of the TT representation, making the TT-ranks smaller, hopefully small enough to make the TT-rounding procedure tractable.

The performance of ALS-based methods is strongly dependent on the capabilities of the local linear equation system solver. For this reason, it might be worthwhile to try other solvers e.g. GCROT [11].

[1] `https://github.com/szdan97/tensortrain`

Iterative solvers are often used with preconditioners in the explicit case [24]. We investigated some possibilities for preconditioning the global system using TT-structured preconditioners in [27], but the techniques we experimented with did not seem to improve performance. A potential research direction is finding techniques for TT-structured preconditioning which improve the performance of ALS-based solvers.

If a method for reliably solving the equation is found, there are several potential directions in the area. The TT-based computation could be adapted to transient metrics, like fixed mission-time reliability, for example using the numerical integration algorithm developed for tensor trains in [15]. Generalizing the method for GSPNs with proper non-determinism is another direction useful for areas where worst-case scenarios must be analyzed, like reliability analysis of safety-critical systems.

Acknowledgements

The results presented in this work were established in the framework of the professional community of Balatonfüred Student Research Group of BME-VIK to promote the economic development of the region. During the development of the achievements, we took into consideration the goals set by the Balatonfüred System Science Innovation Cluster and the plans of the "BME Balatonfüred Knowledge Center", supported by EFOP 4.2.1-16-2017-00021.

I want to say an e^{huge^2} thank you to my advisor, Kristóf Marussy, for all the help he provided. I am also grateful to everyone in the Fault Tolerant Systems Research Group who aided my work in any way. I would like to thank Katalin Friedl and László Kabódi for their insights on the topic. I want to express my gratitude to Péter Lantos for providing me the opportunity of an internship at Prolan Co., where I could see the practical side of reliability analysis through a real-world project.

I also want to thank God for giving me the opportunity for this research, leading me to the people I had to meet to make this happen, and making me capable of this work.

Bibliography

[1] Falko Bause and Pieter S Kritzinger. Stochastic petri nets: An introduction to the theory. *ACM SIGMETRICS Performance Evaluation Review*, 26(2):2–3, 1998.

[2] Peter Buchholz, Tugrul Dayar, Jan Kriege, and M Can Orhan. On compact solution vectors in kronecker-based markovian analysis. *Performance Evaluation*, 115:132–149, 2017.

[3] Gianfranco Ciardo and Andrew S Miner. Storage alternatives for large structured state spaces. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 44–57. Springer, 1997.

[4] Gianfranco Ciardo, Jogesh Muppala, and Kishor S Trivedi. On the solution of gspn reward models. *Performance evaluation*, 12(4):237–253, 1991.

[5] Tugrul Dayar. Analyzing markov chains based on kronecker products. *MAM*, pages 279–300, 2006.

[6] Sergey V Dolgov and Dmitry V Savostyanov. Alternating minimal energy methods for linear systems in higher dimensions. *SIAM Journal on Scientific Computing*, 36 (5):A2248–A2271, 2014.

[7] Christian Eisentraut, Holger Hermanns, and Lijun Zhang. On probabilistic automata in continuous time. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 342–351. IEEE, 2010.

[8] Christian Eisentraut, Holger Hermanns, Joost-Pieter Katoen, and Lijun Zhang. A semantics for every gspn. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 90–109. Springer, 2013.

[9] Lars Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2029–2054, 2010.

[10] Wolfgang Hackbusch and Stefan Kühn. A new scheme for the tensor representation. *Journal of Fourier analysis and applications*, 15(5):706–722, 2009.

[11] Jason E Hicken and David W Zingg. A simplified and flexible variant of gcrot for solving nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 32(3): 1672–1694, 2010.

[12] Sebastian Holtz, THORSTEN Rohwedder, and Reinhold Schneider. The alternating linear scheme for tensor optimisation in the tt format. *Preprint*, 71, 2011.

[13] Youpyo Hong, Peter A Beerel, Jerry R Burch, and Kenneth L McMillan. Sibling-substitution-based bdd minimization using don't cares. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(1):44–55, 2000.

[14] Daniel Kressner and Francisco Macedo. Low-rank tensor methods for communicating markov processes. In *International Conference on Quantitative Evaluation of Systems*, pages 25–40. Springer, 2014.

[15] Christian Lubich, Ivan V Oseledets, and Bart Vandereycken. Time integration of tensor trains. *SIAM Journal on Numerical Analysis*, 53(2):917–941, 2015.

[16] Kristóf Marussy, Attila Klenik, Vince Molnár, András Vörös, István Majzik, and Miklós Telek. Efficient decomposition algorithm for stationary analysis of complex stochastic petri net models. In *International Conference on Application and Theory of Petri Nets and Concurrency*, pages 281–300. Springer, 2016.

[17] Kristóf Marussy, Vince Molnár, András Vörös, and István Majzik. Getting the priorities right: saturation for prioritised petri nets. In *International Conference on Application and Theory of Petri Nets and Concurrency*, pages 223–242. Springer, 2017.

[18] Giulio Masetti, Leonardo Robol, Silvano Chiaradonna, and Felicita Di Giandomenico. Stochastic evaluation of large interdependent composed models through kronecker algebra and exponential sums. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 47–66. Springer, 2019.

[19] Andrew S Miner. Implicit gspn reachability set generation using decision diagrams. *Performance Evaluation*, 56(1-4):145–165, 2004.

[20] Ivan Oseledets. Dmrg approach to fast linear algebra in the tt-format. *Computational Methods in Applied Mathematics Comput. Methods Appl. Math.*, 11(3):382–393, 2011.

[21] Ivan Oseledets and Eugene Tyrtyshnikov. Tt-cross approximation for multidimensional arrays. *Linear Algebra and its Applications*, 432(1):70–88, 2010.

[22] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[23] Ivan V Oseledets and SV Dolgov. Solution of linear systems and matrix inversion in the tt-format. *SIAM Journal on Scientific Computing*, 34(5):A2718–A2739, 2012.

[24] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[25] Yousef Saad. *Numerical methods for large eigenvalue problems: revised edition*. SIAM, 2011.

[26] William J Stewart. *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton university press, 2009.

[27] Dániel Szekeres. Towards tensor-based reliability analysis of complex safety-critical systems. Technical report, 2019. URL `https://tdk.bme.hu/VIK/ViewPaper/Tenzorreprezentacios-modszerek-a-komplex`.