Symbolic on-the-fly analysis of stochastic Petri nets

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus/Senftenberg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

von
M.Sc. Informatik
Martin Schwarick

geboren am 02. 02. 1980
in Herzberg/Elster, Deutschland

Gutachter: Prof. Dr.-Ing. Monika Heiner
Gutachter: Prof. Dr. Susanna Donatelli
Gutachter: Prof. Dr. Peter Kemper

Tag der mündlichen Prüfung: 11. Juni 2014

Abstract

This thesis investigates the efficient analysis, especially the model checking, of bounded stochastic Petri nets (SPNs) which can be augmented with reward structures. An SPN induces a continuous-time Markov chain (CTMC). A reward structure associates a reward to each state of the CTMC and defines a Markov reward model (MRM). The Continuous Stochastic Reward Logic (CSRL) permits to define sophisticated properties of CTMCs and MRMs which can be automatically verified by a model checker.

CSRL model checking can be realized on top of established numerical analysis techniques for CTMCs which are based on the multiplication of a matrix and a vector. However, as these techniques consider a matrix and a vector at least in the size of the number of reachable states, it is still challenging to deal with the famous state space explosion problem. Several approaches, as for instance the use of Multi-terminal Decision Diagrams or Kronecker products to represent the matrix, have been investigated so far. They often enable the implementation of efficient CTMC analysis and are available in a couple of tools.

As an alternative to these established techniques I enhance the idea of an on-the-fly computation of the matrix entries deploying a symbolic state space representation. The set of state transitions defining the matrix will be enumerated by the firing of the transitions of the given SPN for all reachable states. The reachable states are encoded by means of Interval Decision Diagrams (IDD).

Further, I discuss crucial aspects for the implementation of the first multi-threaded symbolic CSRL model checker which is based on the developed technique and available in the tool *MARCIE*. An experimental comparison with the probabilistic model checker PRISM for a large number of experiments proves empirically the efficiency of the approach and its implementation, especially when investigating biological models.

Keywords Model Checking, Stochastic Petri nets, Rewards, Continuous-time Markov Chains, Markov Reward Models, Interval Decision Diagrams, Multi-threading, Continuous Stochastic Reward Logic

Zusammenfassung

Die vorliegende Dissertation betrachtet die effiziente Analyse, im Besonderen Modelchecking, von beschränkten stochastischen Petrinetzen (SPN), die um Rewardstrukturen angereichert werden können. Ein SPN induziert eine Zeit-kontinuierliche Markov Kette (CTMC). Eine Rewardstruktur assoziiert zu jedem Zustand einen Reward und definiert ein Markov Reward Modell (MRM). Die Kontinuierliche Stochastische Reward Logik (CSRL) erlaubt es, sehr spezielle Eigenschaften für CTMCs und MRMs zu definieren, die von einem Modelchecker automatisch überprüft werden können.

CSRL Modelchecking kann aufbauend auf etablierten numerischen Analysetechniken für CTMCs realisiert werden, die auf der Multiplikation einer Matrix mit einem Vektor basieren. Da diese Techniken eine Matrix und einen Vektor in der Grösse der Menge der erreichbaren Zustände berücksichtigen, ist es jedoch schwierig mir dem Phänomen der Zustandsraumexplosion umzugehen. Verschiedene Ansätze, wie die Verwendung Multi-terminaler Entscheidungsdiagramme (MTDD) oder Kronecker-Produkten zur Darstellung der Matrix, wurden bereits untersucht. Oftmals erlauben diese die Implementierung effizienter CTMC Analyse und sind in einer Reihe von Werkzeugen verfügbar.

Als eine Alternative zu diesen etablierten Techniken entwickle ich die Idee einer "on-the-fly" Berechnung der Matrixeinträge, unter Verwendung einer symbolische Darstellung des Zustandsraums, weiter. Die Menge der Zustandsübergänge, die die Matrix definieren, wird druch das Feuern der Transitionen des Petrinetzes in allen erreichbaren Zuständen aufgezählt. Die Menge der erreichbaren Zustände wird mittels Intervall-Entscheidungsdiagrammen (IDD) kodiert.

Weiter, diskutiere ich wesentliche Aspekte der Implementierung des ersten parallelisierten symbolischen CSRL Modelcheckers, der auf der entwickelten Technik basiert und Teil des Werkzeugs *MARCIE* ist. Ein experimenteller Vergleich mit dem probabilistischen Modelchecker PRISM für eine grosse Anzahl von Experimenten zeigt empirisch die Effizienz dieses Ansatzes und seiner Implementierung, insbesondere bei der Betrachtung biologischer Modelle.

Contents

List of Figures

List of Examples

List of Algorithms

List of Tables

List of Abbreviations

AKAP A-kinase Anchoring Protein - Case study.

BDD Binary Decision Diagram.

BSCC Bottom Strongly Connect Component.

CLOCK Circadian Clock - Case study.

CSL Continuous Stochastic Logic.

CSR Compressed Sparse Row.

CSRL Continuous Stochastic Reward Logic.

CTL Computation Tree Logic.

CTMC Continuous-time Markov Chain.

DD Decision Diagram.

DTMC Discrete-time Markov Chain.

ERK RKIP Inhibited ERK Pathway - Case study.

FMS Flexible Manufacturing System - Case study.

GS Method of Gauss-Seidel.

IDD Interval Decision Diagram.

JAC Method of Jacobi.

KANBAN KANBAN System - Case study.

LIDD Labeled Interval Decision Diagram.

LTL Linear Temporal Logic.

MDD Multi-valued Decision Diagram.

MRM Markov Reward Model.

MTBDD Mutli-terminal Binary Decision Diagram.

MTDD Mutli-terminal Decision Diagram.

MV Functor class - MatrixVector.

MVE Functor class - MatrixVectorEmbedded.

MxD Matrix Diagram.

OIDD Ordered Interval Decision Diagram.

OLIDD Offset-labeled Interval Decision Diagram.

OLROIDD Offset-labeled reduced ordered Interval Decision Diagram.

PGS Pseudo Gauss-Seidel.

PN Petri Net.

PSS Polling Server System - Case study.

ROIDD Reduced ordered Interval Decision Diagram.

RS Functor class - RowSum.

SCC Strongly Connected Component.

SPN Stochastic Petri Net.

SRN Stochastic Reward Net.

VM Functor class - VectorMatrix.

WC Workstation Cluster - Case study.

ZBDD Zero-suppressed Binary Decision Diagram.

List of Symbols

2^A The power set of the set A.

A^n The set of n-ary tuples containing elements of the set A.

C A CTMC.

$C^{\mathcal{A}}$ A CTMC approximating an MRM.

E The exit rates of a CTMC.

F The set of transition functions of a Petri net.

G_S The LDD representation of a set of states S.

I A CSRL time interval.

J A CSRL reward interval.

M An MRM defined by a CTMC C and a reward structure ϱ.

$N^{\mathcal{A}}$ An SPN inducing the approximating CTMC $C^{\mathcal{A}}$.

N^{ϱ} A Petri net representing a reward structure ϱ.

P The set of places of a Petri net.

T The set of transitions of a Petri net.

T_I The set of immediate transitions of a GSPN.

T_S The set of timed transitions of a GSPN.

V The set of arc weights of a Petri net.

V_I The set of inhibitor arc weights of a Petri net.

V_R The set of read arc weights of a Petri net.

X A stochastic process or a set of variables[1].

X_τ The value of the random variable with index τ.

A The CTL ALL-quantor.

[1] context-depending

E The CTL Exists-quantor.

F The Finally-operator.

G The Globally-operator.

$\hat{G}_{\mathcal{S}}$ The LIDD representation of the reachable states $\mathcal{S}$.

P The one-step probability matrix of a CTMC.

$\mathbf{P}^{\mathcal{U}}$ The discretized uniformization matrix of a CTMC and a constant λ.

Q The generator matrix of a CTMC.

R The rate matrix of a CTMC.

$\mathcal{P}_{\bowtie p}$ The CSRL proability operator.

$\upsilon^{M}_{\alpha,\tau,y}$ The distribution of the accumulated reward for the MRM with initial distribution α for te time point τ and the reward bound y.

π^{C}_{α} The steady state distribution of the CTMC C with initial distribution α.

$\pi^{C}_{\alpha,\tau}$ The transient distribution of the CTMC C with initial distribution α at time point τ.

$\rho_{G_S}(s)$ The path representing the state s in the IDD G_S.

Φ A CTL or CSRL state formula.

Ψ A CTL or CSRL state formula.

$\mathcal{RG}_N$ The reachability graph of a Petri net.

$\mathcal{R}_{Ns_0}$ The set of reachable states of a Petri net.

$\mathcal{S}$ The set of reachable states.

$\mathcal{S}_{\bowtie p}$ The CSRL proability operator.

S An arbitrary set of states.

S_N The potential states space of a Petri net.

U The Until-operator.

α A probability distribution.

χ_A The characteristic function of a set A.

$\iota^{C}_{\alpha,\tau}$ The cumulative sojourn times of the CTMC C with initial distribution α at time point τ.

f_t The function of a Petri net transition t which represents a rate for timed transitions

and a weight for immediate transitions.

$\hat{v}$ An LIDD node.

$\imath_s$ The lexicographic index of state s.

λ The uniformization constant.

$\mathbb{B}$ The set continaing 0 and 1.

$\mathbb{N}$ The natural numbers.

$\mathbb{R}_{\geq 0}$ The positive real numbers.

$\mathcal{B}_N$ The transition relation of the reachability graph.

$\mathcal{I}$ The set of intervals defined on $\mathbb{N}$.

$\vDash$ The satisfaction relation.

ϕ A CTL or CSRL state formula.

π The steady state distribution or the set of DD variables[1].

ψ A CTL or CSRL path formula.

σ A path in terms of a state sequence.

s A state of a Petri net.

$s(p)$ The number of tokens on place p in state s.

ϱ A reward function/structure.

${}^\bullet n$ The set of pre-nodes of node n.

$n^\bullet$ The set of post-nodes of node n.

t A Petri net transition.

v An IDD node.

y A reward bound.

1 Introduction

The world we are living in and which we are changing continuously by technical progress is inherently complex. Nevertheless or even for this reason humans are always eager to understand this complexity. On the one hand we want to understand natural phenomena and processes as the weather (to predict it) or biochemical systems as cells, organs or whole organisms (to heal diseases). On the other hand humans themselves develop and construct complex systems. By now our everyday life is indirectly or directly infused by them.

In most domains, as for instance energy supply, health-care or aviation, such systems have necessarily to be safe and dependable. Failures may cause huge economical or ecological damage or in the worst case let people die. However, it is challenging to achieve and prove safety and dependability requirements or at least to measure the risk of a system failure; there is still research in many directions. Formal verification is one of the techniques which may sometimes help to achieve this goal. Its core idea is to derive a model of the system to be studied or developed by abstracting details. The resulting model represents only a particular view to the system. An algorithm is then applied to determine, at best automatically, whether some specified properties are satisfied. In general there is a multitude of more or less formal description languages, which may be used dependent on the nature of the systems and the characteristics of the properties.

Continuous-time Markov chains (*CTMC*) represent a description formalism which may be used if the investigated model features the following properties:

- a discrete nature of states
- a continuous time evolution
- memorylessness.

Such systems can be found in many domains with practical relevance, as for instance performance or reliability evaluation and systems biology.

CTMCs are a well studied subject and due to their properties there is a batch of numerical analysis methods [112] which qualify them for formal verification. A CTMC represents the behaviour of the actual system model and in general one will not write it from scratch. An established approach is to generate the CTMC from a high-level

model of the system. Formalisms widely used for this purpose are generalized stochastic Petri nets (GSPN) [84], and stochastic process algebras [62] as for instance PEPA [64]. A comparison of PEPA and GSPN can be found in [48]. Petri nets enjoy the following nice properties making them very interesting for modeling and analysis of concurrent systems:

- The graphical representation of Petri nets allows even non-experts an easy and intuitive modelling. It is possible to execute a Petri net model by playing the token game. Although this can not be seen as a formal verification technique, it often increases the understanding of a model significantly.
- Petri nets represent a description language with formal semantics.
- There is a rich supply of analysis techniques based on the state space or on the actual net structure.
- There are several tools available supporting all these aspects.

Formal verification of a system requires further to provide a specification of the relevant system properties formulated in some formal language.

Temporal logics are such a family of languages. The technique to determine the satisfaction of a property specified in some temporal logic is known as model checking [38]. In the last 25 years temporal logics and related model checking algorithms have been developed for different types of models and applications, e.g. by Clarke and Emerson with the *Computation Tree Logic* (CTL) [36].

The bottleneck of this promising technique is the famous state space explosion, which is especially caused by the concurrency of single system components. For even small-sized models, the number of reachable states can be huge. It has be shown that in general the number of reachable states can not be bounded by a primitive recursive function with regard to the size of the Petri net [99].

Thus several methods have been developed which avoid the state space explosion by applying partial order or symmetry-based reduction techniques or which just accept the problem and make use of advanced data structures to compactly encode sets of states.

The latter techniques are called symbolic as they represent state sets by means of characteristic functions which are in turn represented by variants of decision diagrams. *Reduced Ordered Binary Decision Diagrams* (ROBDD) [18] are the first and probably the most famous class of decision diagrams which provide a canonical representation for Boolean functions. First model checking techniques which rely on the symbolic approach [86] have been proposed for CTL and allow a qualitative reasoning.

The investigation of quantitative model checking for Markov chains started in [54]

and [6] with probabilistic adaptions of CTL, namely the *Probabilistic Computation Tree Logic* (PCTL) and the *Continuous Stochastic Logic* (CSL). In the last 10-15 years dedicated model checking algorithms, especially for CTMCs and CSL, have been developed and implemented. Most of them are based on numerical standard techniques for Markov chains and thus on an in principle very simple operation, the multiplication of a matrix and a vector. However, in this setting the mentioned state space explosion is tightened by two aspects:

1. The real-valued vectors have the size of the set of system states and we are talking of millions or billions of states (depending on the analysis method several vectors are required).
2. The matrix itself is a quadratic real-valued matrix in the same dimension. Although in most cases the matrix is highly sparse (the majority of the possible entries is zero) we may have to deal with billions of non-zero entries.

Consequently researchers adapted symbolic techniques to address these problems. The matrix representation based on *Multi-terminal Binary Decision Diagrams* (MTBDD) [39] which are used for instance in the probabilistic model checker PRISM [78] can be mentioned as one successful example. However, there are scenarios where MTBDD-based techniques and also alternatives as Kronecker algebraic approaches suffer from the special system characteristics.

Motivation and contribution. In this thesis I develop an alternative approach to enumerate the entries of the rate matrix of a CTMC and apply it to its numerical solution and CSL model checking. I go one step further and consider also *Markov reward models* (MRM) in terms of *stochastic reward nets* (SRN) and the *Continuous Stochastic Reward Logic* (CSRL). The proposed approach is an on-the-fly computation of the matrix given a symbolic state space representation and a high-level description of the model.

The used high-level modeling formalism is basically stochastic Petri nets. The symbolic state space encoding is based on *Reduced Ordered Interval Decision Diagrams* (ROIDD), which generalize ROBDDs. They are proveningly qualified to compactly encode huge state sets of bounded Petri net models [115, 61, 110].

My thesis is especially motivated by research activities in the field of systems biology. Biological systems have been modeled as stochastic Petri nets and have been analyzed with PRISM in [51] and [23]. It turned out that for these models PRISM's hybrid MTBDD engine does not scale for the following reasons:

1. The number of decision diagram variables increases with the value range of the model variables due to the binary encoding of their values.

2. The compactness of the matrix representation and thus the efficiency of related operations is sensitive to the number of distinct matrix entries.

Alternatives as approaches based on Kronecker expressions may fail as well in these cases as they require structured models.

The contributions of my thesis are the following:

- I present a new efficient approach for the numerical analysis of stochastic Petri nets which is based on the on-the-fly computation of the rate matrix given a symbolic state space encoding and a Petri net specification of the model.
- I provide an implementation of the discussed approach resulting in the tool MARCIE. The distinguishing features of the tool are:
  - It outperforms related tools for many published case studies.
  - It supports the Continuous Stochastic Reward Logic as the only available symbolic model checker.
  - It is the only public, available symbolic tool offering multi-threaded numerical engines to analyse SPNs and SRNs and thus CTMCs and MRMs.
- To evaluate the implementation I provide a considerable comparison with the probabilistic model checker PRISM. Here I show that my approach outperforms that of PRISM for large state spaces.

The potential of the presented approach has been shown in [108, 105, 60, 109].

Organization The thesis is organized as follows:

Chapter 2 provides the necessary background material. Here I will give a brief and formal introduction to Place/Transition nets with extended arcs. I will sketch the basic ideas of symbolic state space analysis of bounded Petri nets. Sets of states will be encoded as Interval Logic Functions which in turn will have a canonical representation by means of *Reduced Ordered Interval Decision Diagrams*. This chapter presents further the basics of CTL model checking as an advanced evaluation technique being successfully applied for the introduced Petri net formalism.

Chapter 3 introduces (generalized) stochastic Petri nets and stochastic reward nets and their semantics: Continuous-time Markov chains and Markov reward models. I will further sketch a representation of stochastic reward nets which enables a simple approximation by stochastic Petri nets. After a brief overview of relevant numerical methods for the computation of important probability distributions, I present a CSRL model checking approach which maps the actual problem to CSL model checking based on the presented SRN approximation. The ingredients of

the approach are not knew, but they have not been combined and formalized in the way I will present it here.
This part motivates in fact the data structures and algorithms which I will discuss in Chapter 4.
The CSRL model checking approach is the first contribution of my thesis.

Chapter 4 is devoted to advanced matrix representation techniques for Markov chains. I give first a short overview of existing approaches alleviating the famous state explosion problem in the context of exact numerical solution techniques for Markov chains, which are basically the use of Multi-terminal (Binary) Decision Diagrams and the use of Kronecker algebraic expression to encode the rate matrix.

The drawbacks of these techniques have motivated the main contribution of my thesis, which I present in the second part of this chapter; a new approach to enumerate on-the-fly the entries of the computation matrix, based on a symbolic states space encoding with ROIDDs. The idea of an on-the-fly computation of the matrix entries has been discussed in the literature and is the foundation of several implementations, but to the best of my knowledge there is no adaption to a symbolic setting.

Chapter 5 discusses some important implementation details of the CSRL model checker in MARCIE, which is based on the new on-the-fly technique, namely the generic design of the numerical solvers and the feature of multi-threading. This first implementation of a symbolic model checker for CSRL is the third contribution of this work.

Chapter 6 presents an experimental evaluation of the developed technique using technical and biological models. A large number of experiments has been made, proving empirically the efficiency of the approach. To allow a judgement of the results I provide a comparison with the widely used probabilistic model checker PRISM.

Chapter 7 This chapter summarizes the achieved results and concludes with some ideas for future research.

2 Petri Nets

Petri nets is a formalism to model a qualitative perspective of concurrent systems. They provide an intuitive, graph-based representation with formal semantics. A Petri net is a directed graph comprising places (graphic: ○), which store indistinguishable items, called tokens (graphic: •) and transitions (graphic: □), which destroy and create tokens on those places to which they are connected by arcs (graphic: ⟶) following the so-called firing rule. Places usually represent passive system components. Transitions are used to model active components or events. The theoretical foundations trace back to the thesis of Carl Adam Petri in 1962. In the past numerous extensions were published, which augment the classical Petri net formalism by new arc types, define different semantics by changing the firing rule, introduce a notion of time or even combine all these issues.

In this work I consider stochastic Petri nets (SPN) [93]. As an important high-level formalism for the specification of Continuous-time Markov chains (CTMC), they are frequently used in performance and reliability analysis [104]. Starting with [52] SPNs became also an important tool for the modeling and analysis of biological networks [23, 51, 94, 82].

Before introducing SPN I will give a definition of Place/Transition nets with extended arcs similar to [115]. In contrast to this work I will not consider reset arcs. In the following I call such nets simply Petri nets (PN), or nets for short.

2.1 Petri Nets

A Petri net is a tuple $N = [P, T, V, V_R, V_I, s_0]$ where P is a finite set of places, T a finite set of transitions, $V : P \times T \cup T \times P \to \mathbb{N}$ the set of arcs with their weights, $V_R : P \times T \to \mathbb{N}$ the set of read arc (graphic: —•) weights, and $V_I : P \times T \to \mathbb{N}$ the set of inhibitor arc (graphic: —○) weights. A zero weight means that the arc does not exist. The places are containers for indistinguishable items, called tokens. A mapping $s : P \to \mathbb{N}$ associates to each place an amount of tokens and represents a state (marking) of the Petri net. s_0 is the initial state.

A Petri net is a bipartite graph. The two different node types represent passive (places)

and active (transitions) elements of the model. For a node n we let ${}^\bullet n = \{n' \mid V(n',n) > 0 \vee V_I(n',n) > 0 \vee V_R(n',n) > 0\}$ denote its set of pre-nodes and $n^\bullet = \{n' \mid V(n,n') > 0 \vee V_I(n,n') > 0 \vee V_R(n,n') > 0\}$ its set of post-nodes. The consideration of V_R and V_I is only meaningful for transitions. We further generalize these definitions for a node set $\mathcal{N}$ as

$${}^\bullet\mathcal{N} = \bigcup_{n\in\mathcal{N}} {}^\bullet n \text{ and } \mathcal{N}^\bullet = \bigcup_{n\in\mathcal{N}} n^\bullet.$$

The transitions represent events which consume and produce tokens on the places, to which they are connected by an arc. This implies in general a state change. The arc weights given by V define the flow relation for the tokens. The occurrence of an event is represented by the *firing* of the related transition. When a transition fires, it consumes on all its pre-places as many tokens as defined by the related ingoing arc weights and produces as many new tokens as defined by the related outgoing arc weights. A transition is **enabled** if on all its pre-places there is a sufficient amount of tokens, regarding to the connected arcs. Inhibitor arcs and read arcs allow to define additional constraints concerning the enabledness of a transition without representing an actual flow of tokens. I will give a more formal description.

Transition firing. For a transition $t \in T$ the following mappings define the consumption and production behaviour and possible constraints, which can also be represented as vectors from $\mathbb{Z}^{|P|}$:

$$\begin{aligned} t^+(p) &= V((t,p)) \\ t^-(p) &= V((p,t)) \\ t^R(p) &= V_R((p,t)) \\ t^I(p) &= \begin{cases} V_I((p,t)) & \text{if } V_I((p,t)) > 0 \\ \infty & \text{otherwise} \end{cases} \end{aligned}$$

and $\Delta t(p) = t^+(p) - t^-(p)$. A transition t is enabled in a state s if

$$s \geq t^- \text{ and } s < t^I \text{ and } s \geq t^R,$$

where $<$ and $\geq$ represent the element-wise application of the related operation. $\text{enabled}(s) = \{t \in T \mid s \geq t^- \text{ and } s < t^I \text{ and } s \geq t^R\}$ is the set of transitions, enabled in state s. If t is enabled in state s, it **may** fire (there is no force to do so). The firing leads to the state $s' = s + \Delta t$ and we can observe the state transition $s \xrightarrow{t} s'$. We say that s' is reachable in one step from s. The reachability relation $s \xrightarrow{*} s'$ is the transitive closure of the one-step reachability. It defines $\mathcal{R}_N(s) = \{s' \mid s \xrightarrow{*} s'\}$, the set of states which are reachable from a given state s in the Petri net N. $\mathcal{R}_N(s_0)$ denotes the set of reachable states of the Petri net N.

Paths. A path $\sigma = s_0, s_1, s_2, \ldots$ is any finite or infinite sequence with $\forall s_i, \exists t \in T : s_i \xrightarrow{t} s_{i+1}$. $\sigma[i]$ gives the state s_i. The set $Paths_s = \{\sigma \mid \sigma[0] = s\}$ contains all paths starting in s. The reachability of a state s implies the existence of a path σ starting in s_0 containing the state s.

Conflicts and concurrency. It is possible that the firing of an enabled transition t in some state s destroys the enabledness of an other transition t'. This can happen if both transitions share at least one pre-place. In this case one speaks of a structural conflict. The transitions t and t' compete for the tokens on the shared pre-place(s). However, it is state-depending whether a real competition situation, a dynamic conflict, occurs. If enabled transitions do not affect each other, they fire concurrently. Formally, we consider the transitions t, t' to be

- in a **structural conflict** if:
$$^\bullet t \cap {}^\bullet t' \neq \emptyset.$$
- in a **free-choice** conflict if:
$$^\bullet t = {}^\bullet t'$$
If it holds that $V(p,t) = V(p,t')$ (homogeneity), t and t' are enabled or disabled simultaneously.
- in a **dynamic conflict** in state s if:
$$t, t' \in \text{enabled}(s) \wedge t' \notin \text{enabled}(s - t^-) \wedge t \notin \text{enabled}(s - t'^-).$$
- **concurrently enabled** in state s if:
$$t, t' \in \text{enabled}(s) \wedge t' \in \text{enabled}(s - t^-) \wedge t \in \text{enabled}(s - t'^-).$$

Reachability graph. The set of reachable states $\mathcal{R}_N(s_0)$ and the set of state transitions $\mathcal{B}_N = \{(s,t,s') \mid s, s' \in \mathcal{R}_N(s_0) \wedge s \xrightarrow{t} s'\}$ constitute the reachability graph $\mathcal{RG}_N = [\mathcal{R}_N(s_0), \mathcal{B}_N]$ which represents the so-called *interleaving semantics* of the Petri net. The interleaving semantics considers all possible orders of the firing of concurrently enabled transitions. It is not locally decidable at the first glance, whether the outgoing arcs of a state in the reachability graph represent concurrent or conflicting transitions. The consideration of all possible orders of events (transitions firing) is one source of the famous state space explosion problem.

Strongly connected components. A state set $S' \subseteq \mathcal{R}_N(s_0)$ is called strongly connected if for all states $s, s' \in S'$ it holds that $s \xrightarrow{*} s'$ and $s' \xrightarrow{*} s$. In this case S' is called a strongly connected component (SCC). S' is maximal, if for each $s'' \notin S'$ the component $S' \cup \{s''\}$ is not a SCC. There is often special interest in the bottom (or terminal) SCCs (BSCC). S' is a BSCC if once S' has been reached, it is impossible to leave S'. More formally: $\forall s \in S', \nexists\, s' \notin S' : s \xrightarrow{*} s'$. The set $\mathcal{B} = \{C \mid C \text{ is } BSCC\}$ contains all BSCCs of the $\mathcal{RG}_N$ and the states in $\bigcup_{C_i \in \mathcal{B}} C_i$ are called *recurrent*, all others *transient*.

Algorithm 1 (Reachability graph construction)

```
1  func constructRG(N = [P,T,V,V_R,V_I,s_0] : Petri net)
2     U := {s_0}
3     R_N(s_0) := {s_0}
4     B_N := ∅
5     while U ≠ ∅ do
6        s :=selectOneOf(U)
7        U := U \ {s}
8        forall t ∈ enabled(s) do
9           s' := s + Δt
10          if s' ∉ R_N(s_0) then
11             R_N(s_0) := R_N(s_0) ∪ {s'}
12             U := U ∪ {s'}
13          fi
14          B_N := B_N ∪ {(s,t,s')}
15       od
16    od
17    return [R_N(s_0), B_N]
18 end
```

Behavioural properties. The reachability graph represents the behaviour of the Petri net as it contains all thinkable executions of it. It an be used to determine important behavioural Petri net properties:

- If the $\mathcal{RG}_N$ is finite, the net is bounded. In this case there is an upper bound $k \in \mathbb{N}$ for the number of tokens for all places in all reachable states. We speak also of the **k-boundedness** of the Petri net. **In this thesis I will only consider bounded Petri nets.**
- If the $\mathcal{RG}_N$ does not contain nodes without outgoing arcs, the underlying Petri net is **free of dead states**. In this case the state transition relation is total.

- If the $\mathcal{RG}_N$ is strongly connected, it is always possible to return to the initial state. The property is known as **reversibility**.
- If the $\mathcal{RG}_N$ contains in all BSCCs for all Petri net transitions a state transition, the underlying net possesses the **liveness** property, as it is always possible to enable a transition in the future.

To determine reversibility and liveness we need to know the SCCs, in particular the set $\mathcal{B}$ of the reachability graph. A simple algorithm for this purpose is Tarjan's algorithm [114] which has, as the $\mathcal{RG}_N$ construction (Algorithm 1) itself, a complexity of $\mathcal{O}(|\mathcal{R}_N(s_0)| + |\mathcal{B}_N|)$. However, the linear effort should not hide the fact that in practice an explicit treatment of states and state transitions is not feasible due to the *state space explosion* problem.

Table 2.1 and Table 2.2 show the size of reachability graphs of several biological and technical case studies. For a brief description of the case studies, I refer to the Appendix A.2. The models can be easily scaled by changing the initial marking or the weights of inhibitor arcs modeling the capacity of certain places. Therefor one has to assign a value to the model parameter N. Each net specifies in fact a family [25] of Petri nets, sharing the same graph structure. An alternative approach to scale models is to change their actual graph structure, generally by duplicating certain net elements. A comfortable high-level formalism to specify models with this kind of scalability is colored Petri nets [82]. The *PSS* model exploits this type of scalability. The parameter N specifies in this case the number of structurally identical clients.

The figures in Table 2.1 and Table 2.2 show that even a moderate increase of the value of N causes an explosion of the states and state transitions in all models. An explicit storage and analysis of such graphs becomes quickly a challenging, in many cases an infeasible, problem. For this reason I will sketch in the next section a class of established encoding techniques which are known as symbolic state space techniques.

model	N	$\mid \mathcal{R}_N(s_0) \mid$	$\mid \mathcal{B}_N \mid$
$AKAP$	3	1,632,240	12,691,360
	4	15,611,175	141,398,580
	5	74,612,328	734,259,344
	6	386,805,104	4,116,788,172
	7	1,286,458,560	19,274,793,024
	8	4,729,951,950	75,196,829,970
ERK	20	1,696,618	15,609,594
	30	15,609,594	152,964,416
	40	79,414,335	795,599,588
	50	283,887,981	2,895,687,860
	60	811,375,152	8,377,511,982
	70	1,982,528,598	20,650,564,704
	80	4,313,721,069	45,232,648,776
$CLOCK$	10	644,204	6,766,320
	20	16,336,404	183,032,640
	30	114,516,604	1,312,110,960
	40	463,424,804	5,370,593,280
	50	1,380,101,004	16,104,351,600
	60	3,378,385,204	39,604,537,920
$MAPK(LEV)$	6	1,373,026	15,015,264
	8	10,276,461	125,012,862
	10	52,820,416	690,183,846
	12	210,211,339	2,891,933,226
	14	694,661,670	9,936,133,506
	16	1,992,860,377	29,387,897,076
	18	5,115,081,124	77,305,070,556

Table 2.1: Size of the reachability graphs for different configurations of Petri net models of the biological systems given in Appendix A.2.1.

model	N	$\mid \mathcal{R}_N(s_0) \mid$	$\mid \mathcal{B}_N \mid$
FMS	2	810	3,699
	6	537,768	4,205,670
	8	4,459,455	38,533,968
	10	25,397,658	234,523,289
	12	111,414,940	1,078,917,632
	14	403,259,040	4,047,471,180
	16	1,259,146,701	12,996,555,981
	18	3,497,140,570	36,917,371,811
$KANBAN$	2	4,600	28,120
	4	454,475	3,979,850
	6	11,261,376	115,708,992
	8	133,865,325	1,507,898,700
	10	1,005,927,208	12,032,229,352
	12	5,519,907,575	68,883,925,110
PSS	5	240	800
	10	15,360	89,600
	15	737,280	6,144,000
	20	31,457,280	340,787,200
	25	1,258,291,200	16,777,216,000
WC	8	2,125	12,930
	16	7,821	49,410
	32	29,965	193,026
	64	117,261	762,882
	128	463,885	3,033,090
	256	1,845,261	12,095,490
	512	7,360,525	48,308,226
	1024	29,401,101	193,085,442

Table 2.2: Size of the reachability graphs for different configurations of Petri net models of the technical systems given in Appendix A.2.2.

Example 1

It is time to present the small running example, which I will use throughout the thesis to explain the introduced formalisms and to illustrate the related analysis techniques. The Petri net in Figure 2.1 models a simple producer-

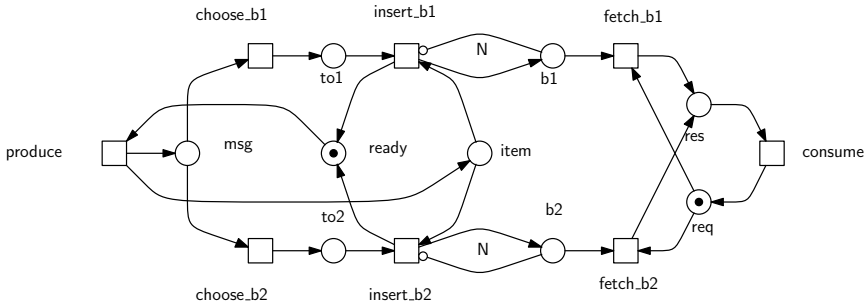


Figure 2.1: A simple producer-consumer model.

consumer system. The producer is represented by the transition *produce*, which hides a more complicated behaviour. The transition is enabled if the place *ready* carries a token. When firing, it produces a token on the place *item* and on the place *msg*. The token on *msg* signals a self-organizing storage instance the existence of the produced *item*. The storage instance comprises two different buffers, namely *b1* and *b2*. Which buffer will be used, is decided non-deterministically. When the decision has been made, the *item* is inserted into the chosen buffer, if the buffer is able to store it. The buffers have finite capacity modeled by the N-weighted inhibitor arcs connecting the places *bi* and the transitions *insert_bi*. If the chosen buffer is full, the producer has to wait.

The consumer is represented by the transition *consume*. After consuming an item the consumer sends a request (place *req*) to the storage instance, which chooses one of the buffers, *fetches* an item and provides it to the consumer on place *res*.

The storage instance is not really smart, as it chooses a buffer without checking its fill level. This may block the producer unnecessarily. Please note that the purpose of the model is only illustration, which will sometimes require certain local net structures. For this reason the model may appear somewhat artificial.

We can specify a value for N and can create the reachability graph using

Algorithm 1. For $N = 1$ the resulting graph has 32 nodes and 64 arcs. Since $\mathcal{R}_N(s_0)$ is a SCC and the state transitions are labeled with all Petri net transitions, we know that the Petri net is bounded, reversible and live. Table 2.3 shows the size of the reachability graph for different buffer capacities. For a value of 1000 we already run into trouble, if we use an explicit representation of the states and state transitions.

N	$\mid \mathcal{R}_N(s_0) \mid$	$\mid \mathcal{B}_N \mid$
1	32	64
10	968	2,530
100	81,608	223,210
1000	8,016,008	22,032,010

Table 2.3: Size of the reachability graphs for different buffer capacities.

★

2.2 Symbolic State Space Representation

A crucial issue in the remainder of my thesis are the compact storage of sets of states of bounded Petri nets and efficient operations to manipulate state sets. Symbolic states space representation techniques exploit compact canonical representations of the characteristic functions of state sets by means of Decision Diagrams (DD). They have been successfully applied in the field of formal verification. In this section I will give a short introduction to *Reduced Ordered Interval Decision Diagrams* [102, 113, 115] which generalize the well-known *Reduced Ordered Binary Decision Diagrams* [18]. They have been used to encode marking sets of k-bounded Petri nets. For an elaborated discussion, especially for the realization of specific ROIDD-operations, I refer to [115]. I will extend the following ROIDD definition in Chapter 4 to realize advanced operations for an efficient computation of the state transition relation of bounded Petri nets.

Characteristic functions. The characteristic function $\chi_A : A \cup \overline{A} \to \mathbb{B}$ for an arbitrary set A with $\chi_A(a) = 1 \Leftrightarrow a \in A$ maps exactly the elements of A to 1. In the scenario on hand, $A \subseteq \mathbb{N}^{|P|}$ represents a set of states of the Petri net $N = [P, T, V, V_R, V_I, s_0]$, for instance the set of reachable states $\mathcal{R}_N(s_0)$. What is needed is a way to formulate such characteristic functions and to encode them compactly.

2.2.1 Interval Decision Diagrams

Interval Logic Functions. Interval logic expressions have been introduced in [81] to describe state sets of Petri nets. Given a finite set of variables $X = \{x_1, \ldots, x_n\}$ we define interval logic expressions inductively:

$$\phi := x \in [a, b) \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$$

whereby $\in$ is treated as a symbol, x is a variable from X and $I = [a, b) = \{x \in \mathbb{N} \mid a \leq x < b\}$ an arbitrary interval on the natural numbers. The set of such intervals is $\mathcal{I}$.

For convenience we give the following short notations:

- $x = n \equiv x \in [n, n+1)$
- $x < n \equiv x \in [0, n)$
- $x > n \equiv x \in [n+1, \infty)$
- $x \neq n \equiv x \in [0, n) \vee x \in [n+1, \infty)$.

An interval logic expression g induces the interval logic function $f_g : \mathbb{N}^{|X|} \to \mathbb{B}$ which maps an n-ary tuple of natural numbers to 1 or 0 by replacing each variable x_i by

the element n_i and evaluating the logic operators. For an interval logic function $f = f(x_1, \ldots, x_n)$ and a variable x_i we define:

- the function $f_{x_i=c} = f(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n)$ as the cofactor which is the result of the substitution of x_i by the constant c.
- the interval $I \in \mathcal{I}$ is an independence interval if $\forall c, c' \in I : f_{x_i=c} = f_{x_i=c'}$. For such an interval the cofactor is the same for all its elements.
- the set $P = \{I_1, \ldots, I_k\}$ with $I_i \in \mathcal{I}$ is an independence interval partition if
  - $\forall I_i \in P : I_i$ is an independence interval
  - $\forall I_i, I_j \in P : I_i \cap I_j = \varnothing$
  - $\bigcup_{1 \leq i \leq k} I_i = \mathbb{N}$

An independence interval partition is called reduced if the union of neighboring intervals does not result in an independence interval. For an interval logic function f and a variable x_i there is a unique reduced interval independence partition P_r. P_r can be repesented by the ordered sequence $b_1, \ldots, b_k$ such that $P_r = \{[a_1 = 0, b_1), \ldots, [a_k, b_k)\}$.

Reduced Ordered Interval Decision Diagrams. An Interval Decision Diagram (IDD) is a rooted, directed and acyclic graph with nodes having an arbitrary number of outgoing arcs. Each arc is labelled with an interval from $\mathcal{I}$. The intervals of the outgoing arcs of each IDD node define an interval partition. There are two nodes without outgoing arcs: the terminal nodes, labelled with 1 and 0. Each IDD node gets associated a variable and we assume that the variables occur in the same order on each path from the root to a terminal node – we get ordered IDDs (OIDDs). If the outgoing arcs of each non-terminal node in an OIDD define a reduced independence interval partition and the OIDD does not contain isomorphic subgraphs we have a reduced ordered IDD (ROIDD).

I give a formal definition conform to [115]. For a set of variables $X = \{x_1, \ldots, x_n\}$, a *Reduced Ordered Interval Decision Diagram* is the tuple $G = [V, E, v_1, L, \pi]$ where:

- $V = V_N \cup V_T$ is a finite set of nodes possessing the non-terminal nodes V_N and the terminal nodes V_T with $(V_T \cap V_N = \varnothing)$.
- $L : (V_N \to X) \cup (V_T \to \mathbb{B})$ is a labeling function which assigns to each non-terminal node a variable from X and to the terminal nodes the values 0 and 1.
- $E \subset V_N \times \mathcal{I} \times V$ is a total relation specifying the arcs. Total means that each non-terminal node has at least one outgoing arc.
- $v_1 \in V$ is a unique node without ingoing arcs. v_1 is called the root.

- π is a fixed order of the variables in X.

Given the functions

- $part(v)$ returns the interval partition $P = \{I_1, \ldots, I_k\}$ induced by the labels of the outgoing arcs of v
- $part_j(v)$ returns the label of the j-th outgoing arc of v, the interval I_j
- $children(v)$ returns the target nodes $C = \{c_1, \ldots, c_k\}$ of the outgoing arcs of v
- $child_j(v)$ returns the target node c_j of the j-th outgoing arc of v

the following conditions must hold:

- the graph $[V, E]$ is a directed acyclic graph (DAG)
- for each path $v_1, \ldots, v_{n+1}$ with $v_{n+1} \in V_T$ it holds that $\forall 1 \leq i < n : (v_i, \cdot, v_{i+1}) \in E$ and $L(v_i) <_\pi L(v_{i+1})$
- for each non-terminal node v with $L(v) = x_i$

  $$f_v(x_i, \ldots, x_n) = \bigvee_{1 \leq j \leq k} x_i \in part_j(v) \wedge f_{child_j(v)}$$

  is an interval logic function and $part(v)$ is an independence interval partition with respect to f_v and x_i.
- the terminal node v defines the interval logic function $f_v = L(v)$
- the graph $[V, E]$ does not contain nodes v, v' which are isomorphic.

The given definition immediately answers the question how to generate the ROIDD F as the canonical representation of an interval logic function f. We have seen that the interval partition of a node induces an interval logic function f_v.

Using the *Bool-Shannon expansion*, each interval logic function f over the variables $X = \{x_1, \ldots, x_n\}$ can be rewritten as

$$f = \bigvee_{I_j \in P} x_i \in I_j \wedge f_{x_i \in I_j}$$

for a variable x_i subject to the constraint that P is a reduced independence interval partition.

For a fixed variable order π, the recursive application of the *Bool-Shannon expansion* for all variables creates an Ordered Interval Decision Tree from the bottom. If we include a procedure which prevents the insertion of nodes, which represent isomorphic subgraphs, we get an ROIDD. The procedure to build an ROIDD from an interval logic function is given in Algorithm 2. It performs such an exhaustive expansion. Figure 2.2 shows an interval logic function and its ROIDD representation.

Algorithm 2 (Build IDD)

```
1  UniqueTable := new(HashTable)
2  max := 2
3  func MakeNode (x : variable, P = {I_1, ..., I_k}, C = (c_1, ..., c_k))
4     while ∃ c_j, c_{j+1} ∈ C such that c_j = c_{j+1} do
5        C:= C \ c_{j+1}
6        I_j := I_j ∪ I_{j+1}
7        P:= P \ I_{j+1}
8     od
9     if |P| = 1 then return c_1 fi
10    res := UniqueTable[x, P, C]
11    if res ≥ 0 then return res fi
12    nodesInIDD := nodesInIDD + 1
13    UniqueTable[x, P, C] := nodesInIDD
14    return nodesInIDD
15 end
16
17 func Build(F,i)
18    if i > |π| then
19       if F = false then return 0 else return 1 fi
20    else
21       P := getPartition(F, i)
22       C := ()
23       for 1 ≤ j ≤ |P| do
24          v_j := Build(F(i, getNumberOf(P,j)), i + 1)
25          insert(C, v_j);
26       od
27       return MakeNode(i, P, C)
28    fi
29 end
30
31 proc BuildIDD(F : IntervalLogicFunction,π : variable order)
32    G_F := new(ROIDD)
33    G_F.root := Build(F, 1)
34    return G_F
35 end
```

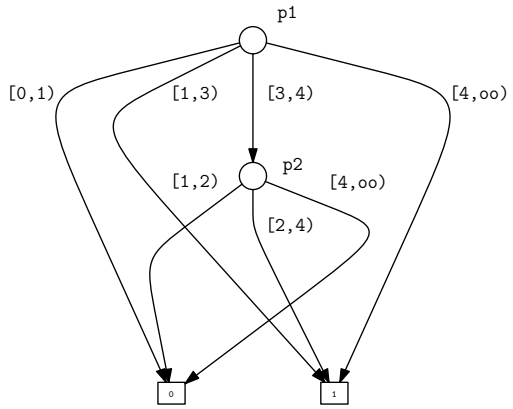


Figure 2.2: ROIDD representation of the interval logic function $f(p1, p2) = p1 \in [1, \infty) \vee p1 \in [3, 4) \wedge p2 \in [2, 4)$.

As an efficient implementation of a decision diagram package can bee seen as basic knowledge, I will not provide here a detailed discussion. For the ROIDD-specific details I refer again to [115].

However, I want to mention two crucial efficiency issues, which are important for the remainder:

- **Variable order.** It is well known that the variable order has an decisive impact on the size and thus on the efficiency of related operations for decision diagrams. Unfortunately, the problem of finding an optimal order is NP-complete. Hence a widely used strategy is to deploy heuristics to precompute a static variable order. The ROIDD package, my thesis is based on, considers the heuristic from [95].
- **Shared Decision Diagrams.** The idea of this technique has been originally introduced for ROBDDs in [17]. It merges the *UniqueTable* and the DAGs (actually the set of nodes) of several decision diagram instances. We can extend the given ROIDD definition as follows:
  - the DAG $[V, E]$ may contain several nodes without ingoing arcs
  - each node is a ROIDD root.

In the following I will use the term IDD for short, although I always consider shared, reduced and ordered IDDs.

2.2.2 State Space Representation for Petri Nets

In the remainder of this thesis, IDDs are used to encode state sets of bounded Petri nets. In this section I will sketch the basic ideas, algorithms and implementation concepts of an IDD-based state space analysis of bounded Petri nets, again conform to the presentation in [115]

The fundamental isomorphism. The base of the symbolic state space analysis approach is the isomorphism of a Boolean algebra $[B, \vee, \wedge, \neg, 0, 1]$, where B is an arbitrary set and a set algebra $[2^{S_N}, \cup, \cap, ^-, \varnothing, S_N]$, which is known as the Stone representation theorem for Boolean algebras. In the scenario on hand we consider a Petri net with n places and define B as the set of possible n-ary interval logic functions, where the places are the variables, and $S_N = \mathbb{N}^n$ is the set of all possible states of the net N. What is needed is an injective mapping from B to 2^{S_N} which maps an interval logic function $f \in B$ to $S \subseteq S_N$. We call the interval logic function f the characteristic function of S and denote it as χ_S. Therefor we can describe sets of states and especially related set operations by interval logic functions. For the state sets S_1 and S_2 we have

- $S_1 \cup S_2 \equiv \chi_{S_1 \cup S_2} = \chi_{S_1} \vee \chi_{S_2}$
- $S_1 \cap S_2 \equiv \chi_{S_1 \cap S_2} = \chi_{S_1} \wedge \chi_{S_2}$
- $\overline{S_1} \equiv \chi_{\overline{S_1}} = \neg \chi_{S_1}$.

I have recapped IDDs as a canonical representation of interval logic functions providing also the required unary and binary operations. However, an efficient state space analysis of bounded Petri nets has to consider also the transitions between the reachable states, which are defined by the firing of the Petri net transitions.

State transitions. If we use a state transition just to determine the related successor state we can think of an implicit usage. If we are interested in the state transition itself, for instance if some additional measure is associated to it, we use it explicitly.

A symbolic representation of the state transition relation which permits explicit access can be achieved through characteristic functions which encode a binary relation $R \subseteq S \times S'$. Therefor we consider two disjoint sets of variables $\{x_1, \ldots, x_n\}$ and $\{x'_1, \ldots, x'_n\}$ and define the characteristic function of R as

$$\chi_R(x_1, \ldots, x_n, x'_1, \ldots, x'_n) = 1$$
$$\Leftrightarrow \exists (s, s') \in R : \chi_s(x_1, \ldots, x_n) = 1 \wedge \chi_{s'}(x'_1, \ldots, x'_n) = 1.$$

I will come back to this idea in Section 4.2.2. For now we only require an implicit use of state transitions, as our concern is the state space construction.

We define the one-step reachability relation in terms of the firing of Petri net transitions. For a given state set S we are interested in S', the states reachable by firing the transition t in any state in S, and we define the operation

$$Fire(S,t) = \{s' \in S_N \mid s \in S : s \xrightarrow{t} s'\}.$$

We get the complete set of successor states of a state set S, if we take into account all transitions. The related operation is

$$Img(S) = \{s' \in S_N \mid s \in S, \exists t \in T : s \xrightarrow{t} s'\} = \bigcup_{t \in T} Fire(S,t).$$

The operation Fire. As we encode state sets and related operations using IDDs it is worth defining a special IDD operation for $Fire$. The idea of such a dedicated decision diagram operation was introduced in [119] for Zero-suppressed Binary Decision Diagrams (ZBDD) and also applied in [102] (BDD, IDD) and [95] (ZBDD). Algorithm 3 shows a simplified version of the operation for IDDs in [115], which does not consider reset arcs. I will describe it in more detail as it motivates the IDD operation which I will present in Section 4.3. The enabling conditions and the firing effect of Petri net transitions are encoded by means of so-called action lists. Each element of an action list is a tuple $al = (var, enabled, shift) \in X \times \mathcal{I} \times \mathbb{Z}$. The single entries of such a tuple al have the following semantics:

- $al.var \in X$ specifies the related variable
- $al.enabled$ defines the token interval which enables the transition with respect to this place
- $al.shift$ defines the token change on this place which is caused by the firing of the transition.

For each transition t with the environment $Env_t = {}^\bullet t \cup t^\bullet = \{p_{t_1}, \ldots, p_{t_k}\}$ we define the action list $al_t = \{al_{t_1}, \ldots, al_{t_k}\}$ with

- $al_{t_i}.var = p_{t_i}$
- $al_{t_i}.enabled = [max\{t^-(p_{t_i}), t^R(p_{t_i})\}, t^I(p_{t_i}))$
- $al_{t_i}.shift = \Delta t(p_{t_i})$

and assume that it respects the variable order π.

Algorithm 3 shows the implementation of the function *Fire*, whose arguments are the IDD G_S encoding a set S and the Petri net transition t. The auxiliary function *AuxFire* is used to traverse recursively the IDD, starting with the root node and the complete action list al_t of t.

For the current root v, the first argument, *AuxFire* looks to the associated variable and evaluates the enabledness of t and the effect of its firing. The required information is stored in the second argument, the current element al in action list of T. The following situations need to be distinguished:

1. The end of recursion has been reached ($al = \perp \vee v = 0$). Either the action list has been completely processed or the algorithm has reached the 0-terminal node. In both cases v is simply returned. In the first case, the firing of transition t does not affect the marking of places related to remaining IDD variables, which means that the IDD rooted by v will not be changed. In the second case, the extracted state does not enable t.
2. The current node represents a place in the environment of t ($a.var = var(v)$). The first step is the evaluation of the enabling values of the related place ($NewPart$). This is done by intersecting the interval partition associated to node v with the enabling interval. The second step is to compute the children of the new IDD node. If an interval in $NewPart$ intersects an interval of $part(v)$, *AuxFire* is called for the related child of v and the tail of the action list. The last step is to shift the computed interval bounds as defined by $a.shift$.
3. The value of $var(v)$ does not represent a place in Env_t, but the firing of the transition affects the value of at least one variable at a lower ROIDD level ($a.var > var(v)$). Thus the algorithm has to continue the traversal by calling *AuxFire* for all its children and the current action list element al.
4. A node with the variable $a.var$ does not occur in the current path and has been skipped in the variable order ($a.var < var(v)$). This situation represents the case that the related place is unbounded. As we consider only bounded Petri net models, this situation is irrelevant for the state space construction. However, it is similar to case (2) except that *NewPart* is simply $a.enabled$.

As in Algorithm 2, the function *MakeNode* ensures the construction of a reduced interval decision diagram. As every efficient decision diagram operation, *AuxFire* tries to reuse previously computed sub-problems. Therefor the function uses the look-up table *ResultTable* which is implemented as a fast hash map.

Forward state space construction. Based on the function *Fire* we can realize the functions *Img* and *FwdReach* given in Algorithm 4. *FwdReach* follows a breath first search strategy to compute the set of states, which are reachable from the set S. It distinguishes processed (*Old*) and unprocessed (*New*, initially the set S) states. In each iteration it takes the unprocessed states, computes the set of successor states using the function *Img* and determines which of them are newly explored. The current set of unprocessed states is added to the processed ones. The procedure terminates

Algorithm 3 (IDD operation – Fire)

```
1  func Fire (G_S: IDD, t : transition)
2     func AuxFire (v : unsigned, al : ActionList)
3        if al = ⊥ ∨ v = 0 then return v fi
4        a := head(al)
5        if ResultTable[v, a] ≠ ∅ then return ResultTable[v, a] fi
6        if var(v) = a.var then
7           NewPart := Intersect(part(v), a.enabled)
8           forall I_j ∈ NewPart, I_k ∈ part(v) do
9              if I_j ∩ I_k ≠ ∅ then
10                NewChild_j := AuxFire(child_k(v), tail(al))
11             od
12             Shift(NewPart, a.shift)
13          fi
14          CompletePartition(NewPart, NewChild)
15          res := MakeNode(var(v), NewPart, NewChild)
16       elseif var(v) > a.var then
17          NewPart_1 := a.enabled
18          Shift(NewPart, a.shift)
19          NewChild_1 := AuxFire(v, tail(al))
20          CompletePartition(NewPart, NewChild)
21          res := MakeNode(a.var, NewPart, NewChild)
22       else  /* var(v) < a.var */
23          NewPart := part(v)
24          forall I_j ∈ NewPart do
25             NewChild_j := AuxFire(child_j(v), al)
26          od
27          res := MakeNode(var(v), NewPart, NewChild)
28       fi
29    fi
30    ResultTable[v, a] := res
31    return res
32 end
33 begin
34    G_S' := new(IDD)
35    G_S'.root := AuxFire(G_S.root, t.al)
36    return G_S'
37 end
```

when all states returned by the function *Img* have already been explored in previous iterations. When calling *FwdReach* with $S = \{s_0\}$, it computes $\mathcal{R}_N(s_0)$.

It is well known that this naive implementation of *FwdReach* has several drawbacks [27]. On the one hand, it does not process immediately new states. On the other hand it considers in each iteration the complete set of Petri net transition ignoring the local effect of transition firing. Both aspects generally result in a high number of iterations whereby the intermediate state sets are often represented by large-sized Decision Diagrams (DD), although the final result may have a very compact DD representation.

Solutions for these problems are chaining [103] or Saturation [30, 115]. Transition chaining is the technique where the set of processed states is updated after applying *Fire* for each transition. In the Saturation technique described in [115], transitions are fired in conformance with the IDD, i.e. according to an ordering, which is derived from the variable order π. A transition is saturated if its firing does not add new states to the current state space (processed). Transitions are bottom-up saturated (i.e. starting at the terminal nodes and going towards the root). Having fired a given transition, all preceding transitions have to be saturated again, either after a single firing (single) or the exhausted firing (fixpoint) of the current transition.

Both techniques have in common that their efficiency depends on the used transition order.

Algorithm 4 (Forward state space construction)

```
1  func FwdReach (S : set of states)
2      Reached := S
3      New := S
4      repeat
5          Old := Reached
6          Reached := Reached ∪ Img(New)
7          New := Reached \ Old
8      until New = ∅
9      return Reached
10 end
```

```
1  func Img (S : set of states)
2      Res := ∅
3      forall t ∈ T do
4          Res := Res ∪ Fire (S, t)
5      od
6      return Res
7  end
```

Backward state space construction. The forward state space analysis is used to compute states, which are reachable from a given state set S and thus represent the possible future evolution of the model. Given a state set S, we can also reason about

the possible past evolution of the model. In this case we want to compute the set of states from which the states in S are reachable. This is based on the backward firing of the Petri net transitions. Analogously to the functions *Fire*, *Img* and *FwdReach* we can define the functions *RevFire*, *PreImg* and *BwdReach* with the following semantics:

- $RevFire(S,t) = \{s' \in S_N \mid s \in S : s' \xrightarrow{t} s\}$
- $PreImg(S) = \{s' \in S_N \mid s \in S, \exists t \in T : s' \xrightarrow{t} s\} = \bigcup_{t \in T} RevFire(S,t)$
- $BwdReach(S) = \{s' \in S_N \mid s \in S, \exists \sigma \in T^* : s' \xrightarrow{\sigma} s\}$

The function *RevFire* can be easily realized by using *Fire* taking the reversed action list $revAl_t$ with

- $revAl_{t_i}.var = al_{t_i}.var$
- $revAl_{t_i}.shift = -al_{t_i}.shift$
- $revAl_{t_i}.var = al_{t_i}.enabled + al_{t_i}.shift.$

State space construction in backward direction represents an important tool for the analysis techniques I will present in the remainder of this thesis. Please note that backwards directed state space generation may explore states which are **not** contained in $\mathcal{R}_N(s_0)$, and thus may require a special treatment.

Example 2

We can apply the symbolic state space construction to the running example with $N = 2$ and get the IDD given in Figure 2.3. Please note that the IDD graph structure is for this example independent of the actual value of N. The reason is the use of inhibitor arcs to model the capacity of the places *b1* and *b2*.

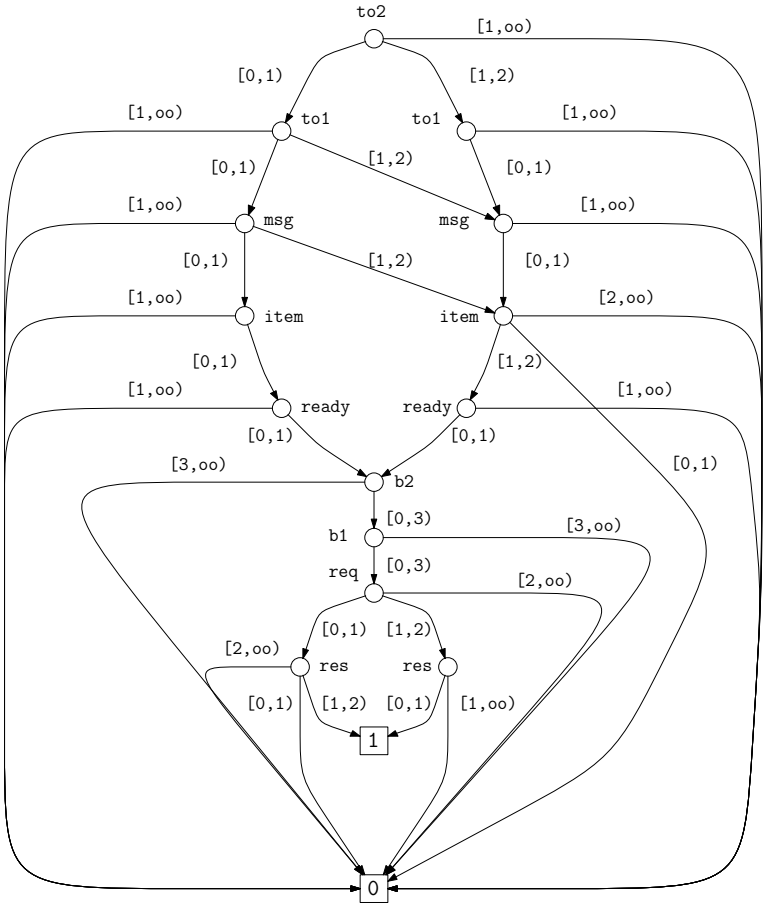


Figure 2.3: Interval Decision Diagram encoding the set of reachable states for the net in Figure 2.1 for $N = 2$.

★

2.3 CTL Model Checking

Model checking [36] denotes the application of an algorithm, the model checker, to evaluate for a formal model description the truth of a model property formalized using some propositional temporal logic in a fully automatic fashion. Under a temporal logic we understand an extension of a proposition logic by so-called temporal or tense operators. This allows to express qualitative properties as "eventually p holds in a state". Pnuelli considered temporal logics with a linear time semantics (LTL) to reason about the properties of reactive systems [98]. In the linear time model all possible executions of the model have to fulfill the given specification.

A different approach is to consider branching time, where it is possible in each state to chose between different future evolutions. To specify properties with branching time semantics Clarke and Emerson introduced the Computation Tree Logic (CTL) [36].

The semantics of such temporal logics is traditionally built upon a Kripke structure $K = [\mathcal{S}, R, L, s_0]$ with $\mathcal{S}$ a finite set of states, $R \subseteq \mathcal{S} \times \mathcal{S}$ a total transition relation (i.e. $\forall s \in \mathcal{S}, \exists s' \in \mathcal{S} : (s, s') \in R$), $L : \mathcal{S} \to 2^{AP}$ a labeling function which maps atomic propositions from AP to the states, and s_0 an initial state.

We can treat the reachability graph $\mathcal{RG}_N = [\mathcal{R}_N(s_0), \mathcal{B}_N]$ of a Petri net N as a Kripke structure if we set $\mathcal{S} = \mathcal{R}_N(s_0)$, disregard the labels of the transition relation $\mathcal{B}_N$ and just consider the existence of state transitions. Doing so we can define R simply as

$$\exists t \in T : s \xrightarrow{t} s' \Leftrightarrow (s, s') \in R$$

and

$$\mathcal{R}_N(s) = \{s\} \Rightarrow (s, s) \in R$$

which adds loops in the case of dead states to make the relation total.

Atomic propositions. The labeling function L maps a set of atomic propositions to each state. When the Kripke structure is constructed from a Petri net model, it makes sense to describe the atomic propositions naturally by relations of place markings. I consider an atomic proposition as a comparison of the results of two arithmetic functions using the place markings in a state as their arguments. I give an inductive syntax definition as

$$ap ::= (f \bowtie f)$$

with

$$\bowtie ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >,$$

and $f : \mathbb{N}^{|P|} \to \mathbb{R}$ an arithmetic function defined as

$$f ::= p \in P \mid n \in \mathbb{N} \mid f + f \mid f - f \mid f \,/\, f \mid f * f.$$

This style of specifying atomic proposition is more expressive compared to the approach in [115] which is restricted to atomic interval logic expression ($p \in I$) and does not allow to formulate propositions as for instance $p_1 = p_2 + p_3$. Unfortunately this higher expressiveness prohibits the construction of a general IDD representation of the satisfying states as it is possible with interval logic expressions and Algorithm 2. However, I assume to know the set of reachable states $\mathcal{S}$ and propose to accept this restriction. Therefore I need the function $extractAP(\mathcal{S}, ap = (f \bowtie f')) := \{s \in \mathcal{S} \mid f(s) \bowtie f'(s)\}$ which returns the states in $\mathcal{S}$ satisfying ap.

For *Functions* and *FunctionArguments* I assume the interface given in Algorithm 5.

Algorithm 5 (Interface – Functions)

```
1  struct FunctionArguments
2      map : map of unsigned
3
4      proc setArgument(var : Variable, value : unsigned)
5          map[var] := value
6      end
7  end
8  struct Function
9      impl : FunctionImpl
10
11     func operator()(args : FunctionArguments)
12         return impl.computeValue(args);
13     end
14
15     func maxVar()
16         return impl.maxVar();
17     end
18
19     func isArg(var: Variable)
20         return impl.isArg(var);
21     end
22
23     func createArgs()
24         return impl.createArgs();
25     end
26 end
27
```

A *Function* instance can be seen as a functor [1]. The function maxVar returns the maximal index of a variable contained in the function with respect to the IDD variable order π.

Algorithm 6 (Extract – Atomic proposition)

```
1  func ExtractAP (G : IDD , ap = (f ⋈ f') : atomic proposition)
2     func AuxExtractAP (v: unsigned, args, args' : FunctionArguments)
3        var := var(v)
4        if var > f.maxVar() & var > f'.maxVar() then
5           if f(args) ⋈ f'(args') then return v else return 0 fi
6        fi
7        forall I_j ∈ NewPart do
8           forall s_i ∈ I_j do
9              if f.isArg(var) then args.setValue(var, s_i) fi
10             if f'.isArg(var) then args'.setValue(var, s_i) fi
11             NewChild_j := AuxExtract(child_j(v))
12          od
13       od
14       res := MakeNode(var(v), NewPart, NewChild)
15       return res
16    end
17    begin
18       G_S' : = new(IDD)
19       G_S' := AuxExtractAP(G.root, f.createArgs(), f'.createArgs())
20       return G_S'
21    end
```

Given an IDD representation G of a state set, Algorithm 6 extracts all states fulfilling the atomic proposition $ap = (f \bowtie f')$. While traversing G it collects the values of the relevant variables in two *FunctionArguments* instances. When the set of function arguments is complete, the function values are computed and compared with respect to the operator $\bowtie$. If the result is true, the current IDD r is returned. Otherwise the traversal continues. The results of recursive calls to the auxiliary function AuxExtractAP are collected to create a new IDD node with a new interval partition and a new list of children. Again, the function MakeNode keeps the resulting IDD reduced. AuxExtractAP

[1] A functor is an object, which encapsulates the intended behaviour of a function. Its type definition overloads a dedicated operator and enables to call a function by the object's name. Therefor one speaks also of a callable object. An important advantage of this concept is the opportunity to encapsulate the related data next to a specific behaviour. In the programing language C++, the operator to be overloaded is *operator() < parameter list >* . In the remainder of this thesis I will make intensive use of this concept.

is not realized as a memory function but it terminates the recursive descent as soon as the maximal variable with respect to the variable order π of both functions has been reached.

For the set of reachable states $\mathcal{S}$ and a set of atomic proposition AP, the labeling function is

$$L = \{(s,p) \mid s \in \mathcal{S}, p \subseteq AP, \forall ap \in p : s \in extractAP(\mathcal{S}, ap)\}.$$

Besides such "natural" atomic propositions I will sometimes use $ap_{S'}$ to denote an atomic proposition which is mapped by L exactly to the states contained in S', e.g. ap_{init} to the initial state or ap_{dead} to all dead states.

2.3.1 Computation Tree Logic - CTL

Syntax. Given a set of atomic propositions AP, the syntax of CTL can be defined inductively given state formulas

$$\phi ::= true \mid ap \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathbf{E}[\psi] \mid \mathbf{A}[\psi]$$

and path formulas

$$\psi ::= \mathbf{X}\phi \mid \phi \mathbf{U} \phi.$$

This definition contains only the temporal operators Ne**X**t and **U**ntil. Often one can find the operators **F**inally and **G**lobally and the following relations can be applied:

$$\begin{array}{lcl} \mathbf{AF}\Phi & \equiv & \mathbf{A}[true\mathbf{U}\Phi] \\ \mathbf{EF}\Phi & \equiv & \mathbf{E}[true\mathbf{U}\Phi] \\ \mathbf{EG}\Phi & \equiv & \neg\mathbf{A}[\mathbf{F}\neg\Phi] \\ \mathbf{AG}\Phi & \equiv & \neg\mathbf{E}[\mathbf{F}\neg\Phi] \end{array}$$

Furthermore, the relations

$$\begin{array}{rcl} \mathbf{AX}\Phi & \equiv & \neg\mathbf{E}[\mathbf{X}(\neg\Phi)] \\ \mathbf{A}[\Phi\,\mathbf{U}\Psi] & \equiv & \neg\mathbf{E}[\neg\Psi\,\mathbf{U}(\neg\Phi \wedge \neg\Psi)] \wedge \neg\mathbf{E}[\mathbf{G}\neg\Psi] \end{array}$$

allow to convert every CTL formula to contain only **EG**, $\mathbf{E}[\mathbf{U}]$ and **EX**.

Semantics. The semantics of CTL formulas is defined with respect to a Kripke structure M (a Model) and the *satisfaction relation* $\models$ for states and paths. As CTL is a branching time logic, formulas are interpreted over the *computation tree (CT)* an infinite unfolding of the Kripke structure. Each tree node (state) may define different

choices for a future evolution of the system. State formulas are interpreted over the states of the CT. All states satisfy the constant proposition *true*. The atomic proposition *ap* is true in a state s, if it occurs in the set of labels, which is associated to s. The formula $\neg\phi$ is fulfilled if the state does not satisfy ϕ. The formula $\phi_1 \wedge \phi_2$ holds in s if it satisfies ϕ_1 and ϕ_2. The formula $\phi_1 \vee \phi_2$ holds in s if it satisfies ϕ_1 or ϕ_2. The CTL formula $\mathbf{E}[\psi]$ is true in state s if there is at least one path starting in s which satisfies the given path formula ψ. $\mathbf{A}[\psi]$ requires the satisfaction of ψ for all paths starting in s.

This can be formalized as follows:

$$\begin{array}{lcl} s \vDash ap & \Leftrightarrow & ap \in L(s) \\ s \vDash \neg\phi & \Leftrightarrow & s \nvDash \phi \\ s \vDash \phi_1 \wedge \phi_2 & \Leftrightarrow & s \vDash \phi_1 \wedge s \vDash \phi_2 \\ s \vDash \phi_1 \vee \phi_2 & \Leftrightarrow & s \vDash \phi_1 \vee s \vDash \phi_2 \\ s \vDash \mathbf{E}[\psi] & \Leftrightarrow & \exists \sigma \in Paths_s : \sigma \vDash \psi \\ s \vDash \mathbf{A}[\psi] & \Leftrightarrow & \forall \sigma \in Paths_s : \sigma \vDash \psi. \end{array}$$

As usual, $Sat(\phi) = \{s \mid s \in \mathcal{S} \wedge s \vDash \phi\}$ is the set of ϕ-states, i.e. all states fulfilling the state formula ϕ.

Path formulas are interpreted over the infinite paths of the CT. The formula $\mathbf{X}\Phi$ holds on a path $\sigma = s_0, s_1, s_2, \ldots$ if s_1 is a Φ-state. $\Phi\mathbf{U}\Psi$ holds on a path if it contains a Ψ-state and all preceding states are Φ-states, more formally:

$$\begin{array}{lcl} \sigma \vDash \mathbf{X}\Phi & \Leftrightarrow & \sigma[1] \vDash \Phi \\ \sigma \vDash \Phi\mathbf{U}\Psi & \Leftrightarrow & \exists i \geq 0, \forall 0 \leq j < i : \sigma[i] \vDash \Psi \wedge \sigma[j] \vDash \Phi. \end{array}$$

A path σ is a ψ-path if it satisfies the path formula ψ and for a state s I declare $Paths_{s,\psi} = \{\sigma \in Paths_s \mid \sigma \vDash \psi\}$ as the set of ψ-paths starting in s.

Example 3

We can use CTL to check the reversibility of net with the CTL formula

$$\mathbf{AG}\,\mathbf{EF}[ap_{init}],$$

and the liveness of a transition t with the CTL formula

$$\mathbf{AG}\,\mathbf{EF}[ap_{enabled_t}].$$

We can also probe whether it is possible that the producer has to wait unnecessarily because the chosen buffer is full while the other is not. We

define the atomic propositions $AP = \{to1 = 1, to2 = 1, b1 = N, b2 = N\}$ and the CTL formula

$$\mathbf{EF}[(to1 = 1 \wedge b1 = N \wedge \neg(b2 = N)) \vee (to2 = 1 \wedge \neg(b1 = N) \wedge b2 = N)].$$

★

2.3.2 Model Checking

CTL model checking [38] is the automatic procedure to determine whether a given Kripke structure M satisfies a given state formula ϕ, which is the case if the initial state s_0 is a ϕ-state, formally written:

$$M \vDash \phi \Leftrightarrow s_0 \in Sat(\phi).$$

The first CTL model checking algorithm has been proposed in [37]. The basic idea is to label all states with the sub-formulas they satisfy. Given the formula tree, each sub-formula is evaluated for all states starting with the inner most formulas and ending with the top formula. The most expensive operation is the evaluation of the until operator which is done by the construction of paths which either prove (witness) or disprove (counterexample) the truth of the path formula using depth-first search algorithms. After the labeling procedure is finished, it remains to check whether the initial state is labeled with the top formula. This complete model checking procedure has a time complexity of $\mathcal{O}(|\phi| \cdot (|\mathcal{S}| + |R|))^2$.

The state space explosion makes the application of this *naive* method often unfeasible. Symbolic CTL model checking algorithms [22, 86] have been developed to alleviate this problem. In this case we do not label single states, but compute the state set $Sat(f)$ for each sub-formula f, as it is shown in Algorithm 7. The core algorithm *check* checks the sub-formulas recursively. As mention earlier, it is sufficient to consider only the quantor-operator combinations $\mathbf{EX}, \mathbf{E}[\mathbf{U}]$, and $\mathbf{EG}$. The evaluation of these temporal operators is done by the related procedures *checkEX*, *checkEG*, and *checkEU*. *checkEG* and *checkEU* require to perform fixed point computations applying backward reachability analysis. For more details I refer to [115].

2.4 Summary

In this chapter I gave a brief introduction to the Petri net (PN) formalism. I recalled the symbolic representation of the state space of bounded Petri nets using Reduced

[2] $|\phi|$ is the number of sub-formulas.

Algorithm 7 (CTL model checking algorithm)

```
1  proc checkCTLFormula(φ : formula, K = [S, R, L, s_0] : model)
2
3     func check(f : formula)
4        if f ≡ true then Sat(f) := S
5        else if f ≡ ap then Sat(f) := extractAP(S, ap)
6        else if f ≡ Φ ∧ Ψ then Sat(f) := check(Φ) ∩ check(Ψ)
7        else if f ≡ Φ ∨ Ψ then Sat(f) := check(Φ) ∪ check(Ψ)
8        else if f ≡ ¬Φ then Sat(f) := S ∖ check(Φ)
9        else if f ≡ EXΦ then Sat(f) := checkEX(check(Φ))
10       else if f ≡ EGΦ then Sat(f) := checkEG(check(Φ))
11       else if f ≡ E[ΦUΨ] then Sat(f) := checkEU(check(Φ), check(Ψ))
12       fi
13       return Sat(f)
14    end
15
16    if s_0 ∈ check(φ) then
17       print(K ⊨ φ)
18    else
19       print(K ⊭ φ)
20    fi
21 end
```

Ordered Interval Decision Diagrams (ROIDD) and sketched the basic ideas of model checking the Computation Tree Logic (CTL).

3 Stochastic Petri Nets

The previously introduced Petri nets represent a powerful formalism to model qualitative aspects of concurrent systems. However, they provide no way to specify a notion of time, which is often an important requirement. Generally, there are different possibilities to augment Petri nets with time. In most of the cases time information is added to the Petri net transitions, which may represent

- working time: Timed Petri nets
- deterministic delays: Time Petri nets
- probabilistic delays: Stochastic Petri nets.

In the following I will consider the latter extension: stochastic Petri nets [93]. Their semantics is a well studied mathematical formalism, which provides a multitude of established techniques for quantitative analysis. SPN are widely used for system modeling and analysis in fields as performance evaluation – and to an increasing degree – in systems biology.

3.1 Stochastic Petri Nets

In a stochastic Petri net possibly state-dependent firing rates[1] are associated to each of the Petri net transitions. Formally I consider an SPN as a tuple $N_S = [N, F]$ with $N = [P, T, V, V_R, V_I, s_0]$ a Petri net and $F : T \to \mathcal{F}$, whereby $\mathcal{F} = \{f_t \mid t \in T, f_t : \mathbb{N}^{|P|} \to \mathbb{R}_{\geq 0}\}$ and $F(t) = f_t$. The behavioural properties of the Petri net N remain valid. We can change the reachability graph definition such that we replace the transition relation $\mathcal{B}_N$ by the relation $\mathbf{R} = \{(s, f_t(s), s') \mid s, s' \in \mathcal{R}_N(s_0) \wedge s \xrightarrow{t} s'\}$ which labels each existing state transition with the related firing rate. The firing rate of a transition t in a state s can be understood as the average number of firings of t in s which are observable during one time unit. The amount of time until leaving state s is known as the sojourn or residence time $\delta(s)$. If the sojourn time is treated as a random variable with a negative exponential distribution governed by the given firing rates, the resulting graph describes a Continuous-time Markov Chain (CTMC) [93], a simple

[1] Sometimes called functional rates.

and well-studied type of stochastic process.

Stochastic processes. A stochastic process $\{X_\tau : \tau \in \mathcal{T}\}$ is a set of random variables over a state space $\mathcal{S}$ and an index set $\mathcal{T}$, which is usually interpreted as time. $X_\tau = s$ denotes that the value of the random variable X_τ is s and we say that the process is in state s[2]. The probability for a random variable X_i to have the value x is

$$Pr\{X_i = x\}.$$

The conditional probability that X_i has the value x given that the random variable X_j has the value x' is

$$Pr\{X_i = x \mid X_j = x'\}.$$

A stochastic process features the *Markov* or *memoryless* property if its future evolution depends only on the current state, formally written as $\forall \tau_0 < \ldots < \tau_i < \tau_{i+1} \in \mathcal{T}, \forall s_0, \ldots, s_i, s_{i+1} \in \mathcal{S}$

$$Pr\{X_{\tau_{i+1}} = s_{i+1} \mid X_{\tau_0} = s_0, X_{\tau_1} = s_1, \ldots, X_{\tau_i} = s_i\} = Pr\{X_{\tau_{i+1}} = s_{i+1} \mid X_{\tau_i} = s_i\}.$$

We classify stochastic processes concerning the nature of the state space, the index set and of course whether the memoryless property holds or not. In the following three cases will be of interest:

- Discrete-time Markov Chains – discrete states, discrete indices (time), and memorylessness
- Continuous-time Markov Chains – discrete states, dense indices (time), and memorylessness
- Performance variables – real valued states, continuous time, and **non**-memorylessness.

Discrete-time Markov Chains (DTMC). A discrete-time Markov chain is a stochastic process with a discrete state space and a discrete time model featuring the memoryless property. In each time step (e.g. induced by a system clock) a state transition occurs with a specified probability. For each state, the probabilities of the possible state transitions sum up to 1. A DTMC $D = [\mathcal{S}, \mathbf{P}]$ is characterized by its state space $\mathcal{S}$ and the *transition probability matrix* $\mathbf{P}$. $\mathbf{P}$ is called a stochastic matrix and fulfills thus the following constraints:

$$\forall i, j \in [0, |\mathcal{S}|) : 0 \leq \mathbf{P}_{i,j} \leq 1$$

[2] Usually one uses X_t but I reserved t to denote a transition of a Petri net.

and

$$\forall i \in [0, |\mathcal{S}|) : \sum_{j \in [0,|\mathcal{S}|)} \mathbf{P}_{i,j} = 1.$$

We will deploy DTMCs as a means to compute measures of Continuous-time Markov Chains.

Continuous-time Markov Chains (CTMC). A CTMC is a stochastic process with a discrete state space in continuous time featuring the memoryless property. The memoryless property and the continuous time require the sojourn times to have a negative exponential distribution. As I consider only CTMCs originating from an SPN I will give a net-related definition. Nevertheless I may also use X to name a CTMC.

Given the SPN $N_S = [N, F]$, the induced CTMC is the tuple $C_N = [\mathcal{S}, \mathbf{R}, s_0]$, where $\mathcal{S} = \mathcal{R}_N(s_0)$ is the set of reachable states, $\mathbf{R} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$ the transition rate relation, and s_0 the initial state of N. Assuming that a state transition is given by a unique Petri net transition[3] the entry $\mathbf{R}(s, s')$ is defined as:

$$\mathbf{R}(s, s') = \begin{cases} f_t(s) & \text{if } \exists t \in T : s \xrightarrow{t} s' \\ 0 & \text{otherwise .} \end{cases}$$

The total (or exit) rate $E(s) = \sum_{s' \in \mathcal{S}} \mathbf{R}(s, s')$ is the sum of the rates of the transitions leaving state s. From the state transition relation $\mathbf{R}$ we derive the *generator matrix* $\mathbf{Q}$

$$\mathbf{Q}(s, s') = \begin{cases} \mathbf{R}(s, s') & \text{if } s \neq s' \\ -E(s) & \text{otherwise .} \end{cases}$$

If s is a dead state, $E(s)$ is $= 0$ and we call s also an absorbing state. Otherwise a transition will fire, which leads to some state s'. The negative exponentially distributed delay of this event $\delta(s)$ is governed by the firing rate of the responsible transition. The probability to leave s within τ time units is

$$Pr\{\delta(s) < \tau\} = 1 - e^{-E(s) \cdot \tau}.$$

The probability of the state transition $s \xrightarrow{t} s'$ is

$$f_t(s)/E(s) = \mathbf{P}(s, s').$$

Its probability within τ time units is

$$\mathbf{P}(s, s') \cdot (1 - e^{-E(s) \cdot \tau}).$$

[3] There are no transitions t, t' with $t^- = t'^- \wedge t^+ = t'^+ \wedge t^I = t'^I \wedge t^R = t'^R$

The relation $\mathbf{P}$ describes the so-called embedded discrete-time Markov chain.

The rates $\mathbf{R}$ and the one-step probabilities $\mathbf{P}$ define $|\mathcal{S}| \times |\mathcal{S}|$ matrices over $\mathbb{R}_{\geq 0}$. If the rates are time-independent, the CTMC is called homogeneous. An inhomogeneous CTMC is defined by several time-dependent transition relations.

Paths. It is necessary to augment the definition of paths from Section 2.1 in the context of CTMCs. A path $\sigma = (s_0, \delta(s_0)), \ldots, (s_i, \delta(s_i)) \ldots\ldots$ is any finite or infinite sequence with $\mathbf{R}(s_i, s_{i+1}) > 0$. $\sigma[i]$ gives state s_i, $\sigma(\tau)$ the index of the state, occupied at time τ.

Probability distributions of CTMCs. Given a CTMC C an interesting question is "what is the probability to be in state s at time instant τ when starting at τ_0 in state s_0?", formalized as

$$\pi^C(s, \tau, s_0) = Pr\{X_\tau = s \mid X_0 = s_0\}.$$

In this case, the system is initially in s_0 with a of probability one. We can generalize this for a probability distribution

$$\alpha : \mathcal{S} \to [0,1] \text{ and } \sum_{s \in \mathcal{S}} \alpha(s) = 1.$$

$\pi^C(s, \tau, \alpha)$ is the probability to be in state s at time instant τ given the initial probability distribution α. Let $\pi^C_{\alpha,\tau}$ denote the *transient probability distribution* at time instant τ given the initial distribution α whereby $\pi^C_{\alpha,\tau}(s) = \pi^C(s, \tau, \alpha)$. In the case of a single initial state s_0 with $\alpha(s_0) = 1$, I may also write $\pi^C_{s_0,\tau}$.

Often Markov chains feature the property that they finally reach a probability distribution, which will not change in the future evolution. This distribution is called a limiting distribution. Finite and irreducible CTMCs (the inducing SPN is bounded and reversible) are called ergodic and possess a unique limiting distribution which is independent of the initial distribution, denoted as the steady state distribution[4].
π^C_α denotes the *steady state distribution* given as $\lim_{\tau \to \infty} \pi^C_{\alpha,\tau}$.

In what follows there is often a high interest in reaching a designated set of states $S' \subseteq \mathcal{S}$. Thus I define also

$$\pi^C_{\alpha,\tau}(S') = \sum_{s \in S'} \pi^C_{\alpha,\tau}(s') \text{ and } \pi^C_\alpha(S') = \sum_{s \in S'} \pi^C_\alpha(s').$$

[4] Ergodicity is not a requirement throughout the remainder, although all considered case studies (See Appendix A.2) are ergodic.

Further I declare $\underline{\pi}^C_{\tau,S'}$ and $\underline{\pi}^C_{S'}$ as the $|\mathcal{S}|$-vectors which store for each state $s \in \mathcal{S}$ the probability $\pi^C_{\alpha,\tau}(S')$ and $\pi^C_{\alpha}(S')$ respectively, given that $\alpha(s) = 1$. Note that $\underline{\pi}^C_{\tau,S'}$ and $\underline{\pi}^C_{S'}$ do **not** represent a probability distribution.

The transient probabilities represent an instantaneous measure for a given time point. In addition we can consider the *cumulative transient probabilities*

$$\iota^C_{\alpha,\tau} = \int_0^\tau \pi_{\alpha,u} du,$$

which yield for each state the cumulative sojourn time in the interval $[0, \tau]$.

If C and α are uniquely determined by the context, I may lazily use π_τ, π, ι_τ and respectively.

3.2 Extensions

In this Section I will describe briefly two extensions of SPN which

- offer more convenient modeling capabilities: generalized stochastic Petri nets (GSPN),
- facilitate the easy definition of additional measures: stochastic reward nets (SRN).

In particular I concentrate on subclasses (not necessarily the largest one) of GSPN and SRN which can be reduced to SPNs or at least can be easily approximated by SPNs. This will finally enable to apply the evaluation techniques I consider in this thesis.

3.2.1 Generalized Stochastic Petri Nets

In SPNs the firing of transitions occurs always after a time delay. However, in some situations it is useful to force a state transition to take place immediately. The sojourn time in a state with such a state transition is zero and for an external observer this state is not visible. States with zero sojourn time are thus called *vanishing* states. All other states are *tangible*. On the Petri net level the described extension is known as *generalized stochastic Petri nets* (GSPN) and has been introduced in [85]. A recommendable textbook on this topic is [84].

In the GSPN formalism, the set of transitions T is divided into a set T_S of *stochastic* (usually called *timed*) transitions (graphic: □) and a set T_I of *immediate* transitions (graphic: ■). As enabled immediate transitions fire without delay, they have a higher priority than timed transitions. A state which enables at least one immediate transition is *vanishing*. In contrast to [84] I do not consider immediate transitions with different priority levels. I further confine myself to bounded and confusion-free[5] GSPNs.

The different priority levels of timed and immediate transitions usually affect the structure and the behaviour of the originating Markov chain as follows:

- The reachability graph of the underlying PN may not represent the semantics of the GSPN directly, as timed transitions lose the enabledness in vanishing states.
- A stochastic transition which has the same preconditions as one of the immediate transitions becomes dead.
- Conflicts between several enabled immediate transitions have to be resolved by specifying the probabilities of firing the single transitions. I use the approach given in [85]. For a vanishing state s_v which enables the set of immediate transitions $T_I' = \text{enabled}(s_v) \setminus T_S$, a random switch as the pair (s_v, sp) where

[5] A confusion is a situation where immediate transitions destroy the locality of a conflict.

$sp : T'_I \to [0,1]$ defines for each transition $t_i \in T'_I$ the probability to fire as

$$f_{p_i} = \frac{f_{t_i}}{\sum_{t_j \in T'_I} f_{t_j}}, \tag{3.1}$$

where f_t defines a possibly state-dependent weight. Two states s and s' are sp-equivalent $s \overset{sp}{\sim} s'$ if they define the same random switch. In Chapter 5 I will discuss a iterative probability propagation deploying the concept of random switches.

- The reachability graph of the GSPN contains vanishing and tangible states and the state transitions are labeled with probabilities and rates, respectively. This extended reachability graph (ERG) can not be directly interpreted as a CTMC.

Example 4

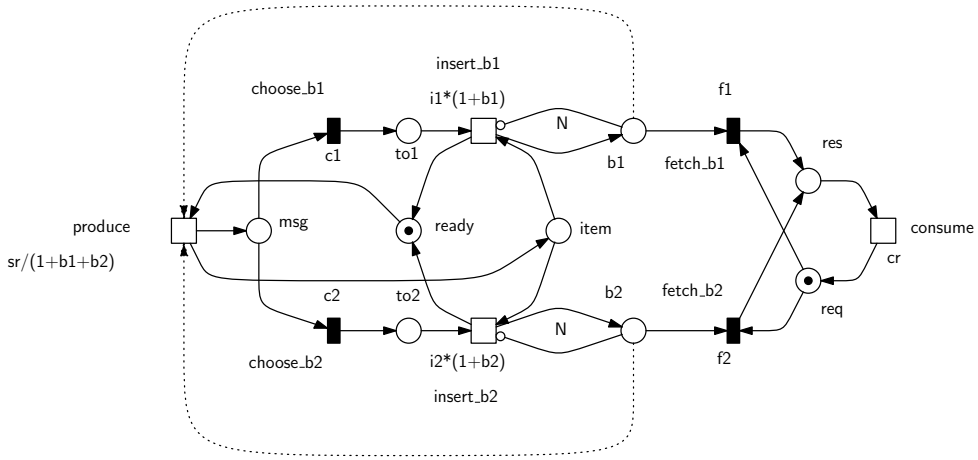


Figure 3.1: GSPN of the running example

Lets define a time behaviour for the example given in Figure 2.1 extending it to a GSPN. Therefor I specify the type of transitions including firing rates and weights as follows:

- The transitions *choose_b*1 and *choose_b*2 fire immediately. As the transitions are in a free-choice conflict we have to specify weights to resolve the conflict. The weights are defined by the constants $c1 : 0 < c1 < 1$ and $c2 : 1 - c1$.
- The transitions $fetch\_b1$ and $fetch\_b2$ fire immediately, too. The transition are in structural conflict which may become a dynamic one or

not. We define the weights to resolve possible conflict situations by the constants $f1 : 0 < f1 < 1$ and $f2 : 1 - f1$.

- The transition *consume* fires independently of the state with a constant rate cr.
- The transition *produce* fires with the state-dependent rate $sr/(1+b1+b2)$ where sr is some constant and $b1$ and $b2$ variables yielding the token values on the related places. The rate decreases with increasing fill level of the storage instance. The dashed arcs, which we call *modifier arcs*, highlight that the transition *produce* considers places in its state-dependent weight although these places are not its pre-places.
- The transitions $insert\_b1$ and $insert\_b2$ fire with the state-dependent rates $i1 * (1 + b1)$ and $i2 * (1 + b2)$ where $i1$ and $i2$ are also constants.

★

The stochastic process which originates from the GSPN is called a Semi-Markov process and can be reduced to a CTMC [85]. The general idea is to replace all possible state transition sequences $\sigma_v = s_0, r_0, s_1, w_1, \ldots, s_{n-1}, w_{n-1}, s_n$ where s_0 and s_n are tangible states, r_0 is the rate of a timed transition and the intermediate states are *vanishing*, by a single state transition (s_0, r, s_n) with rate $r = r_0 \cdot \prod_{1 \leq i < n} w_i$. This requires, that there are no loops containing only vanishing states. This is an applicable approach, if the deployed matrix storage scheme supports efficient modifications in terms of removing and inserting single matrix entries. It may drastically decrease the size of the transition matrix, as it is shown in Table 3.1. However, the number of vanishing states is often as huge that it prohibits the state space construction and a subsequent reduction.

For the on-the-fly computation of the transition matrix which I will introduce in Chapter 4, the reduction can not be applied. In this case it remains to eliminate the immediate transitions at the net level. Each bounded GSPN model can be reduced to a stochastically equivalent SPN model [25]. Furthermore, the authors of [25] present a structural "GSPN to SPN" reduction algorithm for a subset of GSPN families, whereby the considered restrictions can be further relaxed. Here I want to sketch situations which have also been intuitively illustrated in [84].

The general idea is to replace stepwise **each** combination of the firing of a timed transition and the subsequent firing of an immediate transition by a new timed transition interpreting both steps into one step. The weight of the immediate transition depends on the set of simultaneously enabled immediate transitions and the reduction procedure has to consider all possible state-dependent situations. This increases in general the number of transitions in the net significantly. When applying the reduction e.g. to

Table 3.1: The evolution of the state space of the FMS (see A.2.2) and the WC (A.2.2) model given as SPN and GSPN. $|\mathcal{S}_Q|$ denotes the number of states when completely considered qualitatively, i.e. immediate transitions are not priorized over stochastic transitions. A priorization of immediate transitions as it is the case in the GSPN semantics yields the set of reachable states $\mathcal{S}$ which consists of the vanishing states S_V and the tangible states S_T.

model	N	$\lvert\mathcal{S}_Q\rvert$	$\lvert\mathcal{S}\rvert$	$\lvert S_V\rvert$	$\lvert S_T\rvert$
FMS	2	3,444	2,202	1,392	810
	4	438,600	138,060	102,150	35,910
	6	15,126,440	2,519,580	1,981,812	537,768
	8	248,002,785	–	–	4,459,455
	10	2,501,413,200	–	–	25,397,658
WC	8	876	2,772	647	2,125
	16	10,132	10,132	2,311	7,821
	32	38,676	38,676	8,711	29,965
	64	151,060	151,060	33,799	117,261

the workstation cluster (see Appendix A.2.2), the number of timed Petri net transitions increases from 16 (GSPN) to 270 (SPN).

To illustrate the procedure let us reduce the GSPN model of the running example in Figure 3.1 to the stochastically equivalent SPN in Figure 3.3.

Free-choice conflicts. The first situation on hand is if two or more immediate transitions are in conflict and no side conditions are specified as it is the case for the transitions *choose_b*1 and *choose_b*2.

Let us first consider the general case, where we assume to have a place p with m pre-transitions, which can be timed or immediate transitions, $T_1, \ldots, T_m$, and the n immediate post-transitions $t_1, \ldots, t_n$. In this case each combination of the firing of an input transition T_i and an immediate output transition t_j is represented by a new transition $T_i t_j$ with the type of transition T_i having the same input arcs as T_i and the same output arcs as t_j. Its weight or firing rate, depending on the type of the input transition, is given as

$$f_{T_i t_j} = f_{T_i} \cdot \frac{f_{t_j}}{\sum_{k=1,\ldots,n} f_{t_k}}. \tag{3.2}$$

In the reduced net, the place p and its complete environment disappear. The free-choice-conflict situation is easy because the number of tokens on p enables either all or none of transitions in $p^\bullet$.

Figure 3.2 shows the GSPN model after the elimination of the free-choice conflict. In this first step the vanishing place msg has been removed from the net.

However, the immediate transitions $fetch\_b1$ and $fetch\_b2$ remain. These transitions may be in conflict situations whose occurrence and resolution depends on the marking of the places $b1$ and $b2$. This brings us to the second situation, non-free-choice conflicts.

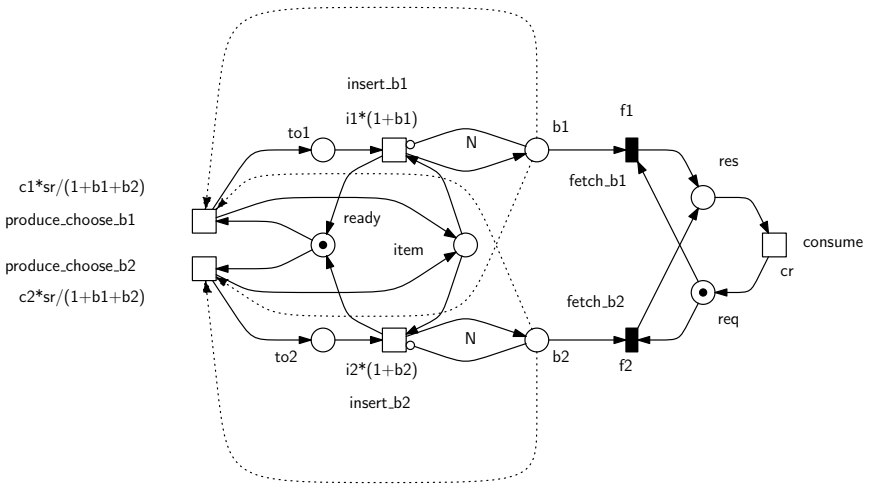


Figure 3.2: The GSPN after the reduction of the free-choice conflict.

Non-free-choice conflicts. In the non-free-choice case, the conflicts have to be resolved by considering all possible situations of the enabledness of the involved transitions. In the reduced model (Figure 3.2) the enabledness of the transitions $fetch\_b1$ and $fetch\_b2$ depend on the marking of the places req, $b1$ and $b2$. Thus we have to consider the following situations:

1. The transition *consume* creates a token on place req. Both buffers are empty and the transitions $fetch\_b1$ and $fetch\_b2$ are disabled. The rate of *consume* is cr. This is a special situation as the immediate transitions can not be removed. They become enabled when one of the *insert*-transitions creates a token on the related buffer. We have to create two additional transitions, namely

   a) $insert\_b1\_2$ which transfers the token from place $to1$ to place res and removes the token on req with rate $i1 \cdot (1 + b1) \cdot cr$. The transition represents the transition sequence $insert\_b1, fetch\_b1$ in the original model. The rate of the new transition is the product of the participating transitions and is thus state-dependent. As the transition is only enabled in the case that the place $b1$ is empty, the rate function is just $i1 \cdot cr$.

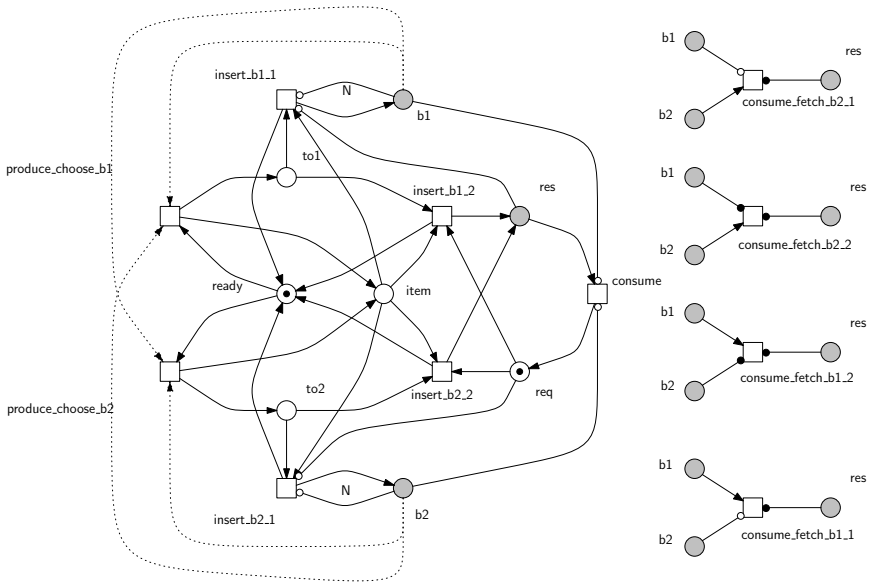


Figure 3.3: The SPN model as the result of eliminating immediate transitions from the GSPN in Figure 3.1. The grey colored places are called *logic* places. All logic places with the same name represent one physical place. The concept of logic nodes is offered by the Petri net editor Snoopy [58] to achieve clear layouts.

b) Analogously we create $insert\_b2\_2$ which transfers the token from place $to2$ to place res and removes the token on req with rate $i2 \cdot cr$.

2. The transition *consume* creates a token on place req and at least one of the $fetch$-transitions is enabled. We have to insert new timed transitions for the following four cases:

   a) Only transition $fetch\_b2$ is enabled as place $b1$ is empty. We insert the transition $consume\_fetch\_b2\_1$ with rate cr.

   b) Only transition $fetch\_b1$ is enabled as place $b2$ is empty. We insert the transition $consume\_fetch\_b1\_1$ with rate cr.

   c) Both transitions are enabled. This is an interesting situation, which must be modelled by two transitions. The resulting timed transitions are enabled and stand in a free-choice conflict. We compute firing rates using Equation 3.2 and create the transitions:

      i. $consume\_fetch\_b1\_2$ with rate $cr \cdot \frac{f1}{f1+f2} = cr \cdot \frac{f1}{1} = cr \cdot f1$.

ii. $consume\_fetch\_b2\_2$ with rate $cr \cdot \frac{f2}{f1+f2} = cr \cdot \frac{f2}{1} = cr \cdot f2$.

In summary we have to introduce five different *consume*-transitions and two additional *insert* (which are actually transfer) transitions. The original transition *consume* and the transitions $fetch\_b1$ and $fetch\_b2$ disappear. The result SPN of the described transformation, is shown in Figure 3.3.

Example 5

Lets have a closer look on the SPN of the running example in Figure 3.3. For $N = 1$, the CTMC has 15 states and 31 states transitions.

Table 3.2: A possible mapping of states to indices.

state	to2	to1	item	ready	b2	b1	req	res
0				1				1
1				1			1	
2				1		1		1
3				1	1			1
4				1	1	1		1
5		1	1					1
6		1	1				1	
7		1	1			1		1
8		1	1		1			1
9		1	1		1	1		1
10	1		1					1
11	1		1				1	
12	1		1			1		1
13	1		1		1			1
14	1		1		1	1		1

If we map each state to an index as can be seen in Table 3.2 and consider the constant values $cr = 1, f1 = 0.8, c1 = 0.9, i1 = 2, i2 = 3, sr = 1$ we obtain the rate matrix

$$\mathbf{R} = \begin{pmatrix}
 & 1.000 & & & & 0.900 & & & & 0.100 & & & \\
 & & & & & 0.900 & & & & & 0.100 & & \\
1.000 & & & & & & 0.450 & & & & & 0.050 & \\
1.000 & & & & & & & 0.450 & & & & & 0.050 \\
 & & 0.200 & 0.800 & & & & & 0.300 & & & & 0.033 \\
 & & 2.000 & & & & 1.000 & & & & & & \\
2.000 & & & & & & & & & & & & \\
 & & & & & 1.000 & & & & & & & \\
 & & & & 2.000 & 1.000 & & & & & & & \\
 & & & & & & 0.200 & 0.800 & & & & & \\
 & & & 3.000 & & & & & & & 1.000 & & \\
3.000 & & & & & & & & & & & & \\
 & & & & 3.000 & & & & & 1.000 & & & \\
 & & & & & & & & & 1.000 & & & \\
 & & & & & & & & & & & 0.200 & 0.800
\end{pmatrix}$$

and the exit rate vector

$$E^T = (2.00, 1.00, 1.50, 1.50, 1.3\overline{3}, 3.002.001.003.001.004.003.004.001.001.00).$$

We can further compute the transient probability distributions (see Section 3.3.1) for the time points 0.1, 1 and 5

$$\pi_{0.1} \approx \begin{pmatrix} 0.008850488316432 \\ 0.905137707825312 \\ 0.000012992107382 \\ 0.000002123550978 \\ 0.000000001250583 \\ 0.000257140502110 \\ 0.077509121062112 \\ 0.000000118541720 \\ 0.000000018815276 \\ 0.000000000004489 \\ 0.000027884219777 \\ 0.008202375837119 \\ 0.000000012543517 \\ 0.000000002160340 \\ 0.000000000000499 \end{pmatrix} \pi_1 \approx \begin{pmatrix} 0.214274920873051 \\ 0.449022518044711 \\ 0.024626351276071 \\ 0.003531239841160 \\ 0.000199752793848 \\ 0.046102341389217 \\ 0.236666146599075 \\ 0.002545065839736 \\ 0.000268383198995 \\ 0.000009763435379 \\ 0.004237629349748 \\ 0.018295531162148 \\ 0.000177711666609 \\ 0.000041545647520 \\ 0.000001084826153 \end{pmatrix} \pi_5 \approx \begin{pmatrix} 0.254109119912367 \\ 0.260129300391418 \\ 0.124101580941124 \\ 0.017398055417151 \\ 0.007359708315539 \\ 0.094049695903364 \\ 0.165667848740787 \\ 0.051668525860243 \\ 0.003006101487222 \\ 0.001802920631375 \\ 0.006969557186351 \\ 0.011081322215262 \\ 0.001547483019053 \\ 0.000908398961109 \\ 0.000200324514597 \end{pmatrix},$$

and the probability distribution in the steady state (see Section 3.3.2)

$$\pi \approx \begin{pmatrix} 0.250589047451738 \\ 0.250589047451738 \\ 0.129039024684902 \\ 0.018699996552253 \\ 0.008943481952699 \\ 0.095884936476242 \\ 0.160707539591403 \\ 0.058604169495140 \\ 0.003520477332084 \\ 0.002683044585810 \\ 0.006965073053546 \\ 0.010674659266240 \\ 0.001627893597087 \\ 0.001173492444028 \\ 0.000298116065090 \end{pmatrix}.$$

★

We can conclude that GSPN provide in principle modelling comfort. A bounded GSPN can always be reduced to a stochastically equivalent SPN, which can be analysed in general more efficiently due to the reduced state space. I will prove this observation empirically in Section 6.6.

3.2.2 Stochastic Reward Nets

A stochastic Petri net $N_S = [N, F]$ inducing the CTMC C can be augmented by a reward function $\varrho : \mathcal{S} \to \mathbb{R}_{\geq 0}$, often also called a reward structure. The reward function ϱ defines for each state s the rate $\varrho(s)$, ϱ_s for short, at which s earns the reward. In this thesis a stochastic reward net is defined by the tuple $N_M = [N_S, \varrho]$. Its semantics $M = [C, \varrho]$ is called a *rate-based Markov reward model (MRM)* [57]. Of course more general types of Markov reward models exist. For instance, stochastic reward nets as defined in [28] possess impulse rewards which are associated to transitions and gained upon their firing.

The accumulated reward. As the CTMC evolves over time, the reward earned by occupying certain system states increases. This defines in turn a stochastic process $\{Y_\tau : \tau \in \mathcal{T}\}$ with domain $\mathbb{R}_{\geq 0}$ and the same index set as for the process X. The value of the random variable Y_τ, the accumulated reward at time instant τ, is

$$Y_\tau = \int_0^\tau \varrho(X_u) du.$$

The accumulated reward at time τ for a path σ is given by

$$Y_\tau(\sigma) = \sum_{i=0}^{\sigma(\tau)-1} \varrho_{s_i} \cdot \tau_i + \varrho_{s_{\sigma(\tau)}} \cdot (\tau - \sum_{i=0}^{\sigma(\tau)-1} \tau_i).$$

$\{Y_\tau : \tau \in \mathcal{T}\}$ is **not** Markovian, as it depends on the complete history of the underlying process X.

In the literature, Y is often referred to as a performance variable and the underlying model as performance model. There are basically three important types of measures which are of interest in this context.

1. **Expected instantaneous reward rate**: The expected value of the reward rate either in steady state

   $$E[\varrho(X)] = \sum_{s \in \mathcal{S}} \varrho(s) \cdot \pi(s) = \varrho \star \pi$$

   or at a specific time point

   $$E[\varrho(X_\tau)] = \sum_{s \in \mathcal{S}} \varrho(s) \cdot \pi_\tau(s) = \varrho \star \pi_\tau.$$

   In both cases the expectation is the scalar of the reward vector ϱ and a probability distribution of the underlying Markov chain.
   Applications:

- Average number of tokens: In systems biology one is often interested in the average number of molecules of some species A. To achieve this we define the reward structure $\varrho^A = \{(s, s(A)) \mid s \in \mathcal{S}\}$. The average number of molecules of species A is $E[\varrho^A(X)]$. This weights for each state the probability with the number of tokens and sums the result for all states.
- Indicator functions: Given a partition $\{S_0, S_1\}$ of the state space $\mathcal{S}$, an indicator variable maps the value 0 to the states S_0 and the value 1 to the system states S_1. When doing dependability analysis one could distinguish the states S_{up}, where the system is *up* and the states S_{down}, where the system is *down* which are used in the reward function

$$\varrho_s^{up} = \begin{cases} 1 & \text{if } s \in S_{up} \\ 0 & \text{otherwise .} \end{cases}$$

  The expected probability that the system is *up*, i.e. in one of the states S_{up}, at time τ (known as availability) is in this case $E[\varrho^{up}(X_\tau)]$.

2. **Expected accumulated reward rate**: The accumulated reward is a measure which weights the cumulative sojourn time of each state[6] with the related rate reward. Its expectation is defined as

$$E[Y_\tau] = \sum_{s \in \mathcal{S}} \varrho(s) \cdot \iota_\tau(s) = \varrho \star \iota_\tau.$$

   Applications:

   - Expected sojourn times: We can reuse the reward structure ϱ^{up} and get the expected up-time in the interval $[0, \tau]$ of the system under investigation as $\varrho^{up} \star \iota_\tau$.
   - Expected number of transition firings: For the transition t we can specify the reward structure

$$\varrho_s^{t_{freq}} = \begin{cases} f_t(s) & \text{if } s \in \mathit{enabled}(t) \\ 0 & \text{otherwise .} \end{cases}$$

     and get the expected number of firings of t in the interval $[0, \tau]$ as $\varrho^{t_{freq}} \star \iota_\tau$. In the biological context this can be interpreted as the expected number of occurrences of a certain type of reaction.

3. **Distribution of the accumulated reward**: The distribution of the accumulated reward is a special case of Meyer's performability [89]. In general performability, a made-up word from performance and dependability, measures a

[6] cumulative transient probabilities

"system's ability to perform in a designated environment" [89]. For a system $\mathcal{S}$ and some performance variable Y,

$$Perf(B) = Pr\{Y \in B\}$$

specifies the probability that $\mathcal{S}$ performs at a designated level B, where B is a subset of the range of Y.

In the scenario on hand the system is given as the Markov process $\{X_\tau : \tau \in \mathcal{T}\}$ and the performance variable as the process $\{Y_\tau : \tau \in \mathcal{T}\}$ defined by a reward structure. So we define the performability

$$\upsilon^{[C,\varrho]}_{\alpha,\tau,y}(s) = Pr\{X_\tau = s, Y_\tau \leq y\},$$

which is the probability that the stochastic process resides at time instant τ in state s and at most a reward of y has been accumulated in the interval $[0,\tau]$. If C and α are uniquely defined by the context, I may use for short $\upsilon^{\varrho}_{\tau,y}(s)$.
Applications:

- Survivability: In [40] performability is used to define a notion of survivability for critical systems. Given a state space partition $\{S_{up}, S_{down}\}$ and the reward structure

$$\varrho^{down}_s = \begin{cases} costs(s) & \text{if } s \in S_{down} \\ 0 & \text{otherwise .} \end{cases}$$

  which maps costs to the *down*-states, survivability for a *down*-state s_{down} can be specified as

$$Pr\{X^{[up]}_\tau \in S^{up}, Y_\tau \leq y \mid X^{[up]}_0 = s_{down}\}.$$

  This means that with a given probability the system is able to be recovered from the *down*-state within τ time units and recovery costs of at most of y. The process $X^{[up]}$ is derived by making all *up*-states absorbing.

In the following I will discuss the representation of a stochastic reward net in more detail. There are two reasons for that:

- Users who model and analyze systems are generally not interested in specifying explicitly the vector which stores the state-specific rewards. Of course we want to have a high-level description for reward structures.
- So far I didn't consider the computation of the presented probability distributions and related measures. I will do this briefly in Section 3.3. The computation of transient and steady state probabilities is expensive in terms of memory and runtime. The computation of the distribution of the accumulated reward is even

much more expensive. I will present in Chapter 4 an efficient symbolic approach for an on-the-fly generation of the Markov chain which allows to compute the discussed measures. To prepare the use of this approach also for the computation of performability, I will present now the construction of an SPN which approximates the original SRN.

Representation of rewards. So far we are able to use SPN as a high-level formalism to specify the state space and the time behavior of a CTMC. Of course we are interested in a comparable high-level specification of the reward structure. We chose a definition style similar to that in [76].

For this purpose I define a reward structure ϱ by a set $\mathcal{R}$ of reward items

$$\mathcal{R} = \{(g, f) \mid g \subseteq \mathbb{N}^{|P|}, f : g \to \mathbb{R}_{\geq 0}\}.$$

A reward item consists of a guard g and a reward function f. The guard specifies the set of states, a subset of the potential state space, to which we associate the rate reward f. To define the guard, I use interval logic expressions (Section 2.2.1). Of course it may happen that the guards of two or more reward items define non-disjoint state sets. For these states the actual reward is the sum of the related reward functions

$$\varrho_s = \sum_{\{i=(g_j,f_j)|i\in\mathcal{R}\wedge s\in g_j\}} f_j.$$

Although the guard of a reward item may specify an arbitrary set of states, we will be interested in subsets of the reachable states and consider a reward item as a tuple $(S' \subseteq \mathcal{S}, \varrho_{S'})$. This specification style may raise the question for the internal representation of rewards. I propose to encode the reward structure as a Petri net, in particular by a set of transitions [105]. The reasons for a transition-based encoding are the following:

1. The reward vector can be similarly computed as the exit rates by applying the on-the-fly matrix computation, which will be introduced in more detail in Chapter 4.
2. The approximation technique used to compute $v_{\alpha,\tau,y}^{[C,\varrho]}(s)$ is based on a transformation of an MRM to a CTMC. With a transition-based reward encoding, the specification of the inducing SPN is rather straightforward.

Therefor I will treat a reward structure ϱ as a Petri net $N^\varrho = [P^\varrho \subseteq P, T^\varrho, V^\varrho = \varnothing, V_R^\varrho, V_I^\varrho, F^\varrho, \varnothing]$, although the mapping F^ϱ specifies reward functions instead of rate functions. The net N^ϱ can be seen as an extension of the original net N as it shares a subset of its places, but defines a set of new transitions induced by the reward structure.

These additional transitions T^{ϱ} have pre-conditions in terms of read arcs or inhibitor arcs as they are possible for all other net transitions. They also have an associated function, but they are never connected with places by standard arcs ($V^{\varrho} = \varnothing$). Their firing will not affect the behaviour of the net, but enable the computation of the rewards for a set of states.

Example 6

We augment the SPN of the running example given Figure 3.3 by a reward structure which maps a reward of 1 to states where the producer has to wait because the chosen buffer is full, although the other buffer has free storage. We define a reward structure for the unnecessary waiting time

$$\varrho_s^{wt} = \begin{cases} 1 & \text{if } s(to1) > 0 \wedge s(b1) = N \wedge s(b2) < N \\ 1 & \text{if } s(to2) > 0 \wedge s(b2) = N \wedge s(b1) < N \\ 0 & \text{otherwise .} \end{cases}$$

We weight the vector ϱ^{wt} with the cumulative sojourn times at time point 1 (vector ι_1)

$$\varrho^{wt} \cdot \iota_1 \approx \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0.108773223626972 \\ 0.659750701950858 \\ 0.001011129896766 \\ 0.006879545880368 \\ 0.000037654671373 \\ 0.017469070461261 \\ 0.187289276651378 \\ 0.000062617187795 \\ 0.000551035648327 \\ 0.000001532939767 \\ 0.001672603732081 \\ 0.016450714203455 \\ 0.000009147095568 \\ 0.000041574917683 \\ 0.000000170326641 \end{pmatrix} \approx \begin{pmatrix} 0.000000000000000 \\ 0.000000000000000 \\ 0.000000000000000 \\ 0.000000000000000 \\ 0.000000000000000 \\ 0.000000000000000 \\ 0.000000000000000 \\ 0.000000000000000 \\ 0.000551035648327 \\ 0.000000000000000 \\ 0.000000000000000 \\ 0.000000000000000 \\ 0.000009147095568 \\ 0.000000000000000 \\ 0.000000000000000 \end{pmatrix}.$$

The result is the vector of the expected cumulative sojourn times in states where the producer has to wait unnecessarily because of a full buffer. The sum

$$E[Y_1] \approx 0.000551035648327 + 0.000009147095568 \approx 0.000560$$

of the entries yields the expected cumulative unnecessary waiting time up to time point 1.

★

Deriving the reward-representing transitions. Let us consider the reward item ($S' \subseteq \mathbb{N}^{|P|}, \varrho_{S'}$). The basic idea is to define a transition, which is exactly enabled in S' and whose associated function defines $\varrho_{S'}$. Unfortunately, a single reward item may define a set of states for which we have to create multiple reward transitions, e.g. if its guard

expression contains a disjunction. At this point we have to pay attention to create the correct set of transitions. Take for instance the guard of the reward item

$$(p1 > 1 \vee (p1 = 3 \wedge p2 \in [2,4)), reward)$$

which is given in disjunctive normal form. We may derive the transitions shown in Figure 3.4, but in states satisfying $p1 = 3 \wedge p2 \in [2,4)$ both transitions would be enabled. This would associate a reward of $2 \cdot reward$ to these states and contradict the reward item definition. What we need is a guard representation by transitions which

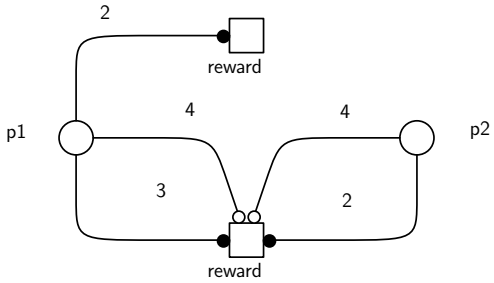


Figure 3.4: Wrong representation of the guard $p1 > 1 \vee (p1 = 3 \wedge p2 \in [2,4))$

fire always exclusively.

To specify the correct set of transitions I use the **canonical** IDD-representation of a guard, which can be derived using Algorithm 2. In an IDD (see Figure 2.2), each edge sequence reaching the 1-terminal node represents an interval logic expression just containing conjunctions of propositions $p \in [a,b)$, whereby p is the variable (a place name) of the current IDD node and $[a,b)$ the label of the chosen arc. For each of these conjunctions it is straightforward to create the pre-condition for a new transition, whereby a is represented by a read arc and b by an inhibitor arc. If $b \equiv \infty$, we can ignore it. We simply extract all paths leading to the 1-terminal node in the IDD encoding of the guards. For each path we create a new reward-induced transition. In general the number of such edge sequences may explode, but in praxis, assuming simple guard expressions, this number should be moderate. For the example above we get the transitions shown in Figure 3.5. Subsequently I will present a procedure which translates an SRN into an SPN approximating its behaviour. The transformation is based on an approximation technique for MRM, called Markovian Approximation [40], which has been proposed in the context of model checking.

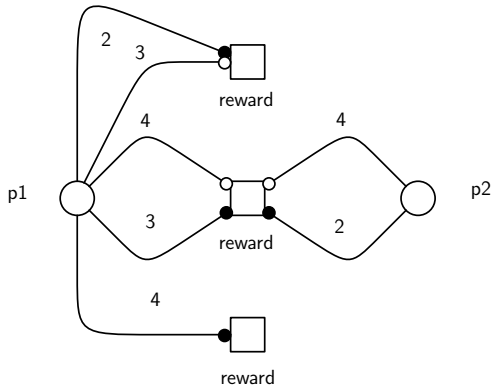


Figure 3.5: Correct representation of the guard $p1 > 1 \vee (p1 = 3 \wedge p2 \in [2, 4))$

Markovian Approximation. As the name may suggest, the basic idea of this method is to create a CTMC which approximates the behaviour of the actual MRM. The accumulated reward will be encoded in the discrete states. This requires to replace the continuous accumulation of reward by a discrete one. The discretized accumulated reward increases stepwise by a fixed value Δ. It requires y/Δ steps to approximately accumulate a reward of y. After n steps the actual accumulated reward lies in the interval $[n \cdot \Delta, (n+1) \cdot \Delta)$, which is the nth reward level. When the reward level is considered as a part of the model, the states of this CTMC are tuples (s, i) where $s \in \mathcal{S}$ is a state of the original model and $i \in \mathbb{N}$ the reward level. At level i all state transitions of the original model, given by the transition relation $\mathbf{R}$, are possible and now denoted as $\mathbf{R}_i$. The increase of the accumulated reward, the steps to the next level, must be realized by an additional state transition relation $\mathbf{R}_{i,i+1}$ which maps the reward function of the MRM to stochastic rates. During one time unit the original MRM accumulates a reward of ϱ_s in the state s. In the derived CTMC $C^{\mathcal{A}}$, we must observe $\frac{\varrho_s}{\Delta}$ transitions per time unit which increase the discretized reward level. For each state (s, i) there must be a state transition to the state $(s, i+1)$ with rate $\frac{\varrho_s}{\Delta}$.

The state space is infinite but countable as the reward growths monotonically without an upper bound. The resulting CTMC's rate matrix $\mathbf{R}^{\mathcal{A}}$ has the following structure and represents a special type of a so-called quasi birth-death process (QBD) [101].

$$\mathbf{R}^{\mathcal{A}} = \begin{pmatrix} \mathbf{R}_0 & \mathbf{R}_{0,1} & 0 & \dots & \dots & \dots & \dots \\ 0 & \mathbf{R}_1 & \mathbf{R}_{1,2} & 0 & \dots & \dots & \dots \\ \dots & 0 & \ddots & \ddots & 0 & \dots & \dots \\ \dots & \dots & 0 & \mathbf{R}_i & \mathbf{R}_{i,i+1} & 0 & \dots \\ \dots & \dots & \dots & \dots & \ddots & \ddots & \dots \end{pmatrix}$$

The probability $Pr_s\{X_\tau \in S', Y_\tau \in (j\Delta, (j+1)\Delta]\}$ to reach at time τ a state from S' while accumulating a reward of $[j\Delta, (j+1)\Delta)$ is approximated by

$$Pr_{(s,0)}\{X^{\mathcal{A}}_\tau \in \{(s', j) \mid s' \in S'\}\}.$$

Transient analysis of infinite QBDs is feasible [101] and requires only a finite number of n computation steps. This number only depends on the given time t, a constant λ, which should be at least the maximal exit rate, and an error bound ϵ. Thus only a finite number of states will be considered, in fact all the states which are reachable from the initial state by paths with a maximal length of n. For a QBD, n and the so-called level diameter, the minimal number of state transitions which are necessary to cross a complete level, can be used to determine in turn a finite set of levels, which have to be considered, see [101] for more details.
In the scenario on hand, the level diameter is always 1, as the level-increasing state transitions may be taken sequentially. This means to always consider the first n reward levels. However, to compute the joint distribution of time and reward we are only interested in states with an accumulated reward of at most y. Thus we can use y instead of t to define the finite subset of repeating levels. In [55], y is called the absorbing barrier. Let S_i denote the states at level i, $S_{>i}$ above and $S_{\leq i}$ below level i. For given y and Δ we define the absorbing level $r = \lfloor \frac{y}{\Delta} \rfloor$ representing the reward value y. States with a higher reward value than y can be abstracted to the set $S_{>r}$ and states with at most a value of y to the set $S_{\leq r}$. Starting at level 0, one has to cross $r+1$ levels (including level 0) to exceed the specified reward bound. Thus we can make all states in S_{r+1} absorbing and get a finite CTMC.

Stochastic Petri net interpretation. A high-level description of the rate matrix $\mathbf{R}^{\mathcal{A}}$ by means of an SPN permits its on-the-fly computation. I will sketch the obvious construction idea.

At first we extend the existing net by an additional place, whose marking represents the reward levels. The number of tokens on this additional place p_y increases only by the firing of the transitions which define the relation $\mathbf{R}_{i,i+1}$. I already specified this set of net transitions as T^{ϱ}. But so far these transitions only define a reward function for certain sets of states, but do not affect the set of reachable states. Two steps remain.

1. The firing of these transitions must increase the number of tokens on p_y.
2. The reward functions must be transformed to stochastic rates. For all reward transitions we redefine the associated function as $f_t' = f_t/\Delta$.

Given the original SRN as $[N_S, \varrho]$ with $N_S = [P, T, V, V_R, V_I, F, s_0]$ and the reward structure ϱ as the net $N^\varrho = [P^\varrho \subseteq P, T^\varrho, V^\varrho = \varnothing, V_R^\varrho, V_I^\varrho, F^\varrho, \varnothing]$, a specified reward step Δ, and a specified reward bound y, we define the SPN
$N^{\mathcal{A}} = [P^{\mathcal{A}}, T^{\mathcal{A}}, V^{\mathcal{A}}, V_R^{\mathcal{A}}, V_I^{\mathcal{A}}, F^{\mathcal{A}}, {s_0}^{\mathcal{A}}]$ with

1. $P^{\mathcal{A}} = P \cup \{p_y\}$
2. $T^{\mathcal{A}} = T \cup T^\varrho$
3. $V^{\mathcal{A}} = V \cup \{((t, p_y), 1) \mid t \in T^\varrho\}$
4. $V_R^{\mathcal{A}} = V_R \cup V_R^\varrho$
5. $V_I^{\mathcal{A}} = V_I \cup V_I^\varrho \cup \{(p_y, t, \lfloor \frac{y}{\Delta} \rfloor + 1) \mid t \in T^\varrho\}$
6. $F^{\mathcal{A}} = F \cup \{(t, f_t/\Delta) \mid (t, f_t) \in F^\varrho\}$
7. $s_0^{\mathcal{A}} = s_0$.

The set of inhibitor arc weights enforces the finite version of $C^{\mathcal{A}}$. Example 7 shows the elements which have to be added to the SPN in Figure 3.3 to create the approximating SPN concerning the reward structure given in Example 6.

The SPN $N^{\mathcal{A}} = [P^{\mathcal{A}}, T^{\mathcal{A}}, V^{\mathcal{A}}, V_R^{\mathcal{A}}, V_I^{\mathcal{A}}, F^{\mathcal{A}}, {s_0}^{\mathcal{A}}]$ allows us to approximate the distribution of the accumulated reward as

$$Pr\{Y_\tau < y\} = \sum_{0 \le i \le \lfloor \frac{y}{\Delta} \rfloor} Pr\{X_\tau^{\mathcal{A}} \in S_i\}.$$

Example 7

We create the approximating net $N^{\mathcal{A}}$ for the SPN in Figure 3.3 and the the reward structure in Example 6 by extending it by the subnet given in Figure 3.6.

The transient analysis for $N = 1, y = 0.01, \Delta = 0.01, \tau_1 = 0.1$ and $\tau_2 = 1$ gives vectors of size $|S| \cdot l$ from which we can derive the following vectors by applying

$$\upsilon_{\tau,y}^{wt}(s) = \sum_{0 \le i < l} \pi_\tau^{C^{\mathcal{A}}}((s, i))$$

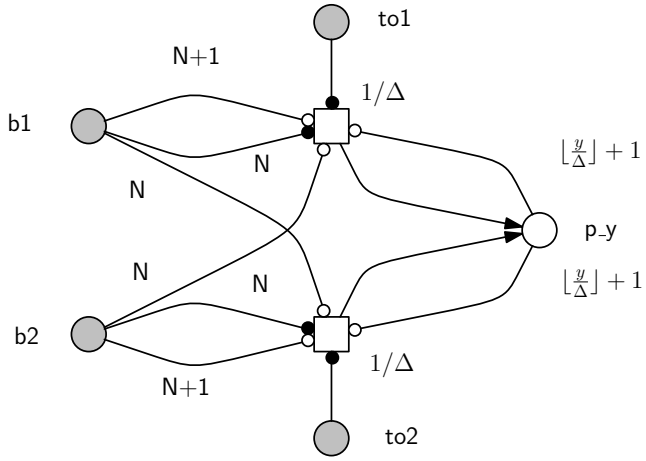


Figure 3.6: The subnet representing the reward accumulation for the SPN in Fig. 3.3 and the reward structure given in Example 6.

$$\upsilon^{wt}_{0.1,0.01} \approx \begin{pmatrix} 0.008850490961004 \\ 0.905137673206139 \\ 0.000012993220656 \\ 0.000002123704422 \\ 0.000000001272433 \\ 0.000257139739951 \\ 0.077509118491380 \\ 0.000000005624919 \\ 0.000000018834152 \\ 0.000000000005531 \\ 0.000027884274709 \\ 0.008202375484388 \\ 0.000000012555989 \\ 0.000000000102172 \\ 0.000000000000615 \end{pmatrix} \quad \upsilon^{wt}_{1,0.01} \approx \begin{pmatrix} 0.214249151450070 \\ 0.449019394879065 \\ 0.024520534866126 \\ 0.003528757543612 \\ 0.000199534351893 \\ 0.045746898760631 \\ 0.236616198739971 \\ 0.000011000580633 \\ 0.000268258860990 \\ 0.000009756293859 \\ 0.004232171095014 \\ 0.018294818705131 \\ 0.000177182831325 \\ 0.000000176746319 \\ 0.000001084032651 \end{pmatrix}$$

★

3.3 Numerical analysis

In this section I sketch numerical methods to compute transient and limiting probability distributions. An elaborated overview on solution techniques for Markov models would go beyond the scope of this thesis. Thus I concentrate on established iterative techniques and refer for further information on this topic to [112]. More or less general methods to compute the distribution of the accumulated reward are collected in [89, 42].

The presented methods have in common that they do not change the computation matrix. This characteristics is important if algorithms exploit sparsity or regularity as it is the case e.g. for symbolic representation techniques. In the work on hand the matrix entries will be generated on-the-fly by an emulation of the firing of Petri net transitions combined with a symbolic state space encoding. This prohibits the application of matrix-changing algorithms. The details of this new approach follow in Chapter 4.

3.3.1 Transient Analysis

I first sketch a numerical method for the computation of the transient probabilities $\pi^C_{\alpha,\tau}$ given the CTMC $C = [\mathcal{S}, \mathbf{R}, s_0]$ with generator matrix $\mathbf{Q}$. The transient distribution at time τ is given by

$$\pi^C_{\alpha,\tau} = \pi^C_{\alpha,0} \cdot e^{\mathbf{Q}\cdot\tau} = \pi^C_{\alpha,0} \cdot \sum_{k=0}^{\infty} \frac{(\mathbf{Q}\cdot\tau)^k}{k!}.$$

It is possible to compute the so-called matrix exponential $e^{\mathbf{Q}\cdot\tau}$ (see for [92] details) and to multiply it with the initial distribution. However, in practice one computes $\pi^C_{\alpha,\tau}$ directly. This is achieved by applying ordinary differential equation solvers, e.g. Runge-Kutta or projection methods as for instance the Krylov subspace method. The possibly most frequently used method for the computation of transient probabilities of Markov models is uniformization, also known as randomization or Jenson's method.

Uniformization. The general idea of this method is to derive from C a discrete-time Markov chain $\mathcal{D}^{\mathcal{U}} = [\mathcal{S}, \mathbf{P}^{\mathcal{U}}, s_0]$, with

$$\mathbf{P}^{\mathcal{U}} = \mathbf{I} + \frac{1}{\lambda}\mathbf{Q}$$

for some $\lambda > max\{E(s) \mid s \in \mathcal{S}\}$.

$D^{\mathcal{U}}$ defines the stochastic process $\{X^{\mathcal{U}}_k : k \in \mathbb{N}\}$ which will be embedded into a *Poisson* process $\{N_\tau : \tau \in \mathcal{T}\}$ with rate λ. The embedding gives the stochastic process $\{X^{\mathcal{U}}_{N_\tau} : \tau \in \mathcal{T}\}$, which is stochastically equivalent to $\{X_\tau : \tau \in \mathcal{T}\}$, and allows us to compute the transient probabilities at time instant τ as

$$\pi^C_{\alpha,\tau} = \sum_{k=0}^{\infty} e^{-\lambda\tau} \frac{(\lambda\tau)^k}{k!} \pi^C_{\alpha,0} (\mathbf{P}^{\mathcal{U}})^k. \tag{3.3}$$

The formula $e^{-\lambda\tau}\frac{(\lambda\tau)^k}{k!}$ specifies the probability that the *Poisson* process N makes k transitions within the time interval $[0, \tau]$.

As the matrix $\mathbf{P}^{\mathcal{U}}$ is a stochastic matrix, the term $\pi^C_{\alpha,0}(\mathbf{P}^{\mathcal{U}})^k$ can be replaced by $\underline{\pi}^{(k-1)}\mathbf{P}^{\mathcal{U}}$ with $\underline{\pi}^0 = \pi^C_{\alpha,0}$, which reduces the problem to a repeated vector-matrix multiplication.

In practice we compute an approximation ${\pi^C_{\alpha,\tau}}^*$ by truncating the infinite sum. The method of Fox and Glynn [50] can be used to compute for a given error bound ϵ a left truncation point L, a right truncation point R, and the related *Poisson* probabilities $\{w_L, \ldots, w_R\}$, such that $|\pi^C_{\alpha,\tau} - {\pi^C_{\alpha,\tau}}^*| < \epsilon$ and

$$\pi^C_{\alpha,\tau} \approx \pi^C_{\alpha,0} \sum_{k=L}^{R} w_k (\mathbf{P}^{\mathcal{U}})^k = {\pi^C_{\alpha,\tau}}^*. \tag{3.4}$$

Algorithm 8 sketches the transient analysis based on uniformization.

Algorithm 8 (Uniformization)

```
1  func Uniformization(α : vector of double , ε, τ : double)
2     L, R : unsigned
3     w : vector of double
4     FoxGlynn(L,R, w, λτ,ε )
5     acc, π: vector of double
6     π := α
7     for k = 0 to k = R do
8        π := π · P^U
9        if k ∈ [L, R] then acc := acc + w[k] · π fi
10    od
11    return acc
12 end
```

Equation 3.4 computes the transient probability distribution at time instant τ given a certain initial distribution α. Usually $\alpha(s) = 1$ holds for a certain state s. If we want to compute $\underline{\pi}_{\tau,S'}$, as for instance in Section 3.4, we have to apply the uniformization method for each state $s \in S'$.

Fast backward analysis. In [68] the authors proposed a technique to compute $\underline{\pi}_{\tau,S'}$ in one pass. The idea is to initialize the goal states S' with a probability of 1 and to invert the direction of the probability propagation. This performs transient analysis backwards and computes for all states $s \in \mathcal{S}$ the probability to be for sure at time instance τ in one of the goal states under the assumption that s was the initial state.

This requires to replace the vector-matrix multiplication by a matrix-vector multiplication in line s in Algorithm 8. The vector $\underline{\pi}$ has to be initialized with 1 for all states $s \in S'$ and 0 otherwise. Please note that in this case neither the vectors $\underline{\pi}$ nor the result vector $\underline{acc}$ represent a probability distribution.

Cumulative transient probabilities. In [100] and [75] it was shown that uniformization can also be used to compute $\iota^C_{\alpha,\tau}$. The required change is a preparation of the used weights $\{w_L, \ldots, w_R\}$ by

$$w_i^c = 1 - \sum_{L \le j \le i} w_j.$$

3.3.2 Limiting Analysis

Next to the computation of transient probabilities, especially of the vector $\underline{\pi}_{\tau,S'}$, we will be interested in computing π^C_α and $\underline{\pi}_{S'}$. This is done by solving a system of linear equations in form of

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

Several methods exist for this purpose, which can be classified into *direct* and *iterative* methods. Direct methods as Gaussian elimination compute the solution in a fixed number of operations while iterative methods approximate the solution step-wise given an initial estimate and a specified convergence criterion. They rely on the multiplication of a matrix and a vector and do not change the matrix. Direct methods may produce a huge amount of *fill-in*, non-zero entries generated during the solution process, which is problematic concerning the implementation of efficient storage schemes and often results in extremely high memory consumption. In Markov analysis one applies traditionally iterative methods as they preserve the often huge, but in general extremely sparse matrices. I will now sketch two of these methods, namely the method of *Jacobi* and *Gauss-Seidel*. For an elaborated presentation I refer to [112].

In both cases the matrix $\mathbf{A}$ is split into two matrices

$$\mathbf{A} = \mathbf{M} - \mathbf{N} \text{ which gives } (\mathbf{M} - \mathbf{N})\underline{x} = \underline{b}$$

to finally derive the iteration scheme

$$\underline{x}^{k+1} = \mathbf{M}^{-1}\mathbf{N}\underline{x}^k + \mathbf{M}^{-1}\underline{b} = \mathbf{H}\underline{x}^k + c$$

with the iteration matrix $\mathbf{H} = \mathbf{M}^{-1}\mathbf{N}$. The iteration terminates, if the *difference* between the current and the previous approximation has become sufficiently

small. Of course there are several ways to specify the difference between two versions of the approximation. As in [96] I consider the relative measure

$$max_i\left(\frac{|\,\underline{x}^k(i) - \underline{x}^{k-m}(i)\,|}{|\,\underline{x}^k(i)\,|} \right) < \epsilon$$

with $m = 1$ and a default value of 10^{-6} for ϵ. In what follows we consider the matrix splitting

$$\mathbf{A} = \mathbf{D} - (\mathbf{L} + \mathbf{U})$$

with $\mathbf{D}$ as the diagonal, $\mathbf{L}$ as the strictly lower, and $\mathbf{U}$ the strictly upper triangular matrices. Further we assume that the diagonal does not contain zero elements, which is always guaranteed in our application scenario.

Jacobi. For the method of Jacobi the iteration matrix is derived by setting $\mathbf{M} = \mathbf{D}$ and $\mathbf{N} = (\mathbf{L} + \mathbf{U})$ which gives the iteration scheme

$$\underline{x}^{k+1} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\underline{x}^k + \mathbf{D}^{-1}\underline{b}$$

in scalar form

$$\underline{x}^{k+1}(i) = \frac{1}{d(i,i)}\left(\sum_{j \neq i} \Big(l(i,j) + u(i,j)\Big)\underline{x}^k(j) + \underline{b}(i) \right) , i = 1, \ldots, n.$$

As we need to remember the kth version of the approximation to compute the $(k+1)$th, two vectors in the size of the state space are required. The Jacobi method is slow as it requires in general many iterations. Furthermore there is no guaranty for convergence. However, as all elements of the current approximation are computed independently of each other the order in which matrix entries are accessed is irrelevant what makes a parallelization very easy.

Gauss-Seidel. Gauss-Seidel improves Jacobi by making use of recently computed approximations. Each time an element has been computed, the previous version will immediately be overwritten. Thus Gauss-Seidel converges much faster than Jacobi. Further it requires only one vector to store the approximation k and $k+1$.

Unfortunately, the immediate use of computed approximations introduces dependencies between the matrix elements. These dependencies require the matrix entries to be accessed row or column-wise, a fact which must be considered

when choosing the matrix storage scheme. A parallelization of Gauss-Seidel is not straightforward.

For the method of Gauss-Seidel the iteration matrix is derived by setting $\mathbf{M} = (\mathbf{D} - \mathbf{L})$ and $\mathbf{N} = \mathbf{U}$ which gives the iteration scheme

$$\underline{x}^{k+1} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\underline{x}^{k} + (\mathbf{D} - \mathbf{L})^{-1}\underline{b}$$

in scalar form

$$\underline{x}^{k+1}(i) = \frac{1}{d(i,i)}\left(\sum_{j=1}^{i-1} l(i,j)\underline{x}^{k+1}(j) + \sum_{j=i+1}^{n} u(i,j)\underline{x}^{k}(j) + \underline{b}(i) \right) , i = 1, \ldots, n.$$

It is further worth mentioning that Gauss-Seidel's convergence speed is affected by the initial order of states.

Reachability of states. A question for which one often wants to get the answer is: "How probable is it to finally reach a state in the state set S'?". If the underlying **bounded** Petri net is **reversible** and the induced Markov chain ergodic, this probability is always 1 and $\underline{\pi}_{S'} = \underline{1}$. However, numerical computations are required, if the Petri net is not reversible or if we forbid to visit a second set of states S''. For the computation of the vector $\underline{\pi}_{S'}$, the probabilities to finally reach a state from S', we have to solve

$$\mathbf{P}\underline{\pi}_{S'} = \underline{v} \text{ with } \underline{v}(s) = \begin{cases} 1 & \text{if } s \in S' \\ 0 & \text{otherwise .} \end{cases}$$

In most cases it is not necessary to consider the state transitions of all states, a circumstance which may decrease the computational effort and the memory consumption drastically. One can distinguish the following classes of states for which I want to give a characterization by means of CTL formulas:

- S_{no} : states with a probability to reach S' of zero.
$$S_{no} = Sat(\neg \mathbf{E}[\neg ap_{S''} \mathbf{U}\, ap_{S'}])$$
- S_{yes} : states with a probability to reach S' of one.
$$S_{yes} = Sat(\neg \mathbf{E}[\neg ap_{S''} \mathbf{U}\, ap_{S_{no}}])$$
- S_{maybe} : states whose a probability to reach S' is neither 0 nor 1 and has to be computed
$$S_{maybe} = \mathcal{S} \setminus (S_{no} \cup S_{yes}).$$

Consequently, S_{yes} and S_{no} states can be made absorbing.

The limiting distribution of reversible Petri nets. For the computation of the *steady state* probabilities π_α^C we have to solve

$$\pi \mathbf{Q} = 0,$$

rearranged to use matrix-vector multiplication

$$\mathbf{Q}^T \pi^T = 0.$$

When using Jacobi we have $\mathbf{M} = E$ and $\mathbf{N} = \mathbf{R}^T$, and obtain Algorithm 9. For

Algorithm 9 (Jacobi – Steady state)

```
1  func SteadyState(ε : double)
2     x^k, x^{k+1}: vector of double
3     x^k := 1/n // element-wise
4     x^{k+1} := 0
5     stop := false
6     while ¬stop do
7        stop := true
8        x^{k+1} := R^T x^k
9        for 1 ≤ i ≤ n do
10          x^k(i) := x^{k+1}(i)/E(i)
11          stop := stop& converged(x^k(i), x^{k+1}(i), ε)
12          x^{k+1}(i) := 0
13       od
14    od
15    return x^k
16 end
```

Gauss-Seidel we formulate Algorithm 10, but we can not easily map E and $\mathbf{R}$ to the matrix splitting $\mathbf{N}$ and $\mathbf{M}$ as the latter is the inverse of the difference of the diagonal and the strictly lower triangular matrix of $\mathbf{R}^T$.

The limiting distribution of non-reversible Petri nets. If the reachability graph of a given SPN is not reversible, the originating CTMC is not ergodic. The computation of the limiting distribution for such situation has been discussed in [11]. The probability for transient states is 0. The recurrent states yield the bottom strongly connected components $\mathcal{B}$. For a BSCC $C_i \in \mathcal{B}$, the distribution π^{C_i} can be computed with the methods described above, but has to be weighted

Algorithm 10 (Gauss-Seidel – Steady state)

```
1  func SteadyState(ε : double)
2     x: vector of double
3     x := 1/n // element-wise
4     stop := false
5     while ¬stop do
6        stop := true
7        for 1 ≤ i ≤ n do
8           r := 0
9           forall j ≠ i : R(j,i) > do
10             r := r + R(j,i)x(i)
11          od
12          r := r/E(i)
13          stop := stop& converged(x(i), r, ε)
14           x(i) := r
15       od
16    od
17    return x^k
18 end
```

with the probability to reach C_i. We have to compute

$$\pi_\alpha^C(s) = \begin{cases} \pi^{C_i}(s) \cdot \underline{\pi}_{\alpha,C_i}(s) & \text{if } \exists C_i \in \mathcal{B} : s \in C_i \\ 0 & \text{otherwise .} \end{cases}$$

3.4 CSRL Model Checking

In Section 2.3 I recalled the Computation Tree Logic (CTL) and the basic ideas of the related model checking procedure. CTL has no notion of explicit time although it is required in many application areas. Thus several quantitative extensions and related model checking algorithms have been proposed in the past. In the Real-Time Computation Tree Logic (RTCTL) [49] the tense operators are decorated with a time bound, whereby time is treated discretely. The Time Computation Tree Logic (TCTL) [2], syntactically equal to RTCTL, has indeed a continuous time semantics and is interpreted over structures where additional time information have been assigned to the state transitions as it is the case for instance for time Petri nets. The first probabilistic adaption of CTL was the Probabilistic Real Time Computation Tree Logic (PCTL) [54] which replaces the RTCTL path quantifier **E** and **A** by an operator to bound the probability of

the truth of path formulas. This allows to express properties as: "the probability of reaching within 10 time units a state where ϕ is true is greater than 0.5". The semantics of PCTL formulas are interpreted over DTMCs. The continuous adaption of PCTL for CTMC analysis is the Continuous Stochastic Logic (CSL), introduced in [5] and extended by a new operator to express steady state properties in [12]. The Continuous Stochastic Reward Logic (CSRL) [10] extends in turn CSL by additional reward bounds associated to the tense operators.

3.4.1 Continuous Stochastic Reward Logic

Syntax. CSRL can be seen as the counterpart of CTL for Markov reward models. The syntactical difference to CTL is that the temporal operators are decorated with a time interval I, and a reward interval J. Further, the path quantifiers $\mathbf{E}$ and $\mathbf{A}$ are replaced by the probability operator $\mathcal{P}_{\bowtie p}$, and there is the steady state operator $\mathcal{S}_{\bowtie p}$.

The CSRL syntax is defined inductively given state formulas

$$\phi ::= true \mid ap \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathcal{P}_{\bowtie p}[\psi] \mid \mathcal{S}_{\bowtie p}[\phi] \ ,$$

and path formulas

$$\psi ::= \mathbf{X}_J^I \phi \mid \phi \, \mathbf{U}_J^I \phi$$

with $ap \in AP$, $\bowtie \in \{<, \leq, \geq, >\}$, $p \in [0,1]$, and $I, J \subseteq \mathbb{R}_+$.

Semantics. CSRL state formulas are interpreted over the states, and path formulas over the paths of the MRM M, as we have seen it for CTL. The CSRL formula $\mathcal{P}_{\bowtie p}[\psi]$ is true in state s if the probability to choose a ψ-path is bounded by the probability p and the operator $\bowtie$. Given the initial distribution $\alpha(s) = 1$, the CSRL formula $\mathcal{S}_{\bowtie p}[\phi]$ is true in state s, if the probability to be in the steady state in a ϕ-state is bounded by $\bowtie p$.

For state formulas we define the satisfaction relation $\models$ as follows

$$\begin{aligned}
s \models ap &\Leftrightarrow ap \in L(s) \\
s \models \neg\Phi &\Leftrightarrow s \not\models \Phi \\
s \models \Phi \vee \Psi &\Leftrightarrow s \models \Phi \vee s \models \Psi \\
s \models \Phi \wedge \Psi &\Leftrightarrow s \models \Phi \wedge s \models \Psi \\
s \models \mathcal{S}_{\bowtie p}[\psi] &\Leftrightarrow \textstyle\sum_{s' \in Sat(\psi)} \pi_s^M(s') \bowtie p \\
s \models \mathcal{P}_{\bowtie p}[\psi] &\Leftrightarrow Prob_s^M(\psi) \bowtie p,
\end{aligned}$$

with $Prob_s^M(\psi) = Pr\{Paths_{s,\psi}\}$ as the probability to satisfy the path-formula Ψ starting in s. The set $Paths_{s,\psi} = \{\sigma \in Paths_s \mid \sigma \models \psi\}$ is measurable [40].

The satisfaction relation $\models$ for path formulas is defined over the set of paths of the Markov chain. The path formula $\psi = \mathbf{X}_J^I\Phi$ holds on a path starting in s if the successor state of s is a Φ-state and is reached within the time interval I. The accumulated reward in this moment must fall into the interval J. The formula $\psi = \Phi\mathbf{U}\Psi$ holds on a path if it contains a Ψ-state s' and all its predecessor states are Φ-states. The time point when s' is reached must fall into the interval I and the accumulated reward in this moment into the interval J. More formally:

$$\begin{aligned}
\sigma \models \mathbf{X}_J^I\Phi &\Leftrightarrow |\sigma| \geq 1 \wedge \delta_0 \in I \wedge \sigma[1] \models \Phi \wedge Y_{\delta_0}(\sigma) \in J \\
\sigma \models \Phi\mathbf{U}_J^I\Psi &\Leftrightarrow \exists \tau \in I : \sigma(\tau) \models \Psi \wedge \forall \tau' < \tau : \sigma(\tau') \models \Phi \wedge Y_\tau(\sigma) \in J.
\end{aligned}$$

Similar to CTL, we define the following relations to use the operators **F**inally and **G**lobally:

$$\begin{aligned}
\mathcal{P}_{\bowtie p}[\mathbf{F}_J^I\Phi] &\equiv \mathcal{P}_{\bowtie p}[true\mathbf{U}_J^I\Phi] \\
\mathcal{P}_{\bowtie p}[\mathbf{G}_J^I\Phi] &\equiv \mathcal{P}_{\bowtie (1-p)}[true\mathbf{U}_J^I\neg\Phi].
\end{aligned}$$

3.4.2 Model Checking

A CSRL model checker answers the question whether a Markov reward model $M = [C = [\mathcal{S}, \mathbf{R}, L, s_0], \varrho]$ satisfies a CSRL formula ϕ which is the case if the initial state is a ϕ-state, formally written:

$$M \models \phi \Leftrightarrow s_0 \models \phi.$$

The CSRL model checking algorithm is shown in Algorithm 11. It is basically inherited from that of CTL (see Algorithm 7). Of course, the functions checkEX, checkEG and checkEU have to be adapted. Further we have to consider the additional steady state operator.

Algorithm 11 (CSRL model checking algorithm)

```
1  proc checkCSRLFormula(φ : formula, M = [C, ϱ] : model)
2
3    func check(f : formula)
4      if f ≡ true then Sat(f) := S
5      elseif f ≡ ap then Sat(f) := abstractAP(S, ap)
6      elseif f ≡ Φ ∧ Ψ then Sat(f) := check(Φ) ∩ check(Ψ)
7      elseif f ≡ Φ ∨ Ψ then Sat(f) := check(Φ) ∪ check(Ψ)
8      elseif f ≡ ¬Φ then Sat(f) := S \ check(Φ)
9      elseif f ≡ S_⋈p Φ then
10       S' := check(Φ)
11       Sat(f) := select(p, ⋈, π_S')
12     elseif f ≡ P_⋈p[X^I_J Φ] then
13       r := checkX(check(Φ), I, J)
14       Sat(f) := select(p, ⋈, r)
15     elseif f ≡ P_⋈p[Φ U^I_J Ψ] then
16       r := checkU(check(Φ), check(Ψ), I, J)
17       Sat(f) := select(p, ⋈, r)
18     elseif f ≡ P_⋈p[G^I_J Φ] then
19       r := 1 − checkU(check(true), check(¬Φ), I, J)
20       Sat(f) := select(p, ⋈, r)
21     fi
22     return Sat(f)
23   end
24
25   func select(p :double, ⋈ : cmp, v : vector of double)
26     return {s | v(s) ⋈ p}
27   end
28
29   if s_0 ∈ check(φ) then
30     print(M ⊨ φ)
31   else
32     print(M ⊭ φ)
33   fi
34 end
```

Being a branching time logic, a path formula comes along with the operator $\mathcal{P}_{\bowtie p}$, which selects the satisfying states given the operator $\bowtie$ and the probability bound p. The selection is realized by the function select in Algorithm 11. The actual work is the computation of the probabilities for each state to take one of the fulfilling paths.

This evaluation step is affected by the values of the specified reward function ϱ, the time interval I, and the reward interval J. Often it is possible to derive intervals I' and J' which may reduce the computational effort or at least allow to solve the problem by an existing evaluation procedure. Take for instance the path formula $\psi = [\Phi \mathbf{U}^{[\tau,\tau']}_{(y,\infty)} \Psi]$. If we were choosing a $(\Phi \mathbf{U} \Psi)$-path whose states gain all the maximal possible reward value ϱ_{max}, the maximal possible accumulated reward is $\tau' \cdot \varrho_{max}$. In this case we can alter the formula to $\psi = [\Phi \mathbf{U}^{[\tau,\tau']}_{(y,\tau' \cdot \varrho_{max}]} \Psi]$.

If the rate reward function is total, we further check whether $y/\varrho_{min} > \tau$. Then we can replace τ by y/ϱ_{min} which may reduce the computational effort. If $y/\varrho_{min} > \tau'$ we skip any numerical computation as the given time and reward bounds are inconsistent with regard to the given rate reward function. However, the evaluation of the resulting formulas generally requires some more or less expensive numerical computations. In the following I will sketch the evaluation of the Ne**X**t -and the **U**ntil-operator. Depending on the time and reward bounds, we can apply the evaluation procedures introduced in [9, 11, 40].

X-operator

The evaluation of the formula $\mathcal{P}_{\bowtie p}[\mathbf{X}^I_J \Phi]$ requires to compute for each state s the probability $Prob^M_s(\mathbf{X}^I_J \Phi)$. The accumulated reward depends only on the reward gained in state s and the probability of a state transition to a Φ-state, which is given by the one-step probabilities $\mathbf{P}$. Further, the transition probability has to be weighted by a value derived from the given time and reward bounds. Let us first consider the special situation that the reward interval is $J = [0, \infty)$. Following [11], the probability of a state transition to a Φ-state within the time interval is

$$Prob^M_s(\mathbf{X}^I \Phi) = (e^{-E(s) \cdot \inf(I)} - e^{-E(s) \cdot \sup(I)}) \cdot \sum_{s' \in \Phi} \mathbf{P}(s, s').$$

In the case that $\inf(J) \neq 0$, the state s must yield a positive reward ϱ_s to fulfill $\mathcal{P}_{\bowtie p}[\mathbf{X}^I_J \Phi]$. Then it is possible to derive the time interval I_s by dividing all elements of J by ϱ_s. I_s represents the set of sojourn times in s which do not break the given reward constraints. This enables to map the problem to the previous scenario and we have

$$Prob^M_s(\mathbf{X}^I_J \Phi) = Prob^M_s(\mathbf{X}^{I \cap I_s} \Phi).$$

Table 3.3 shows the different intervals $I \cap I_s$ which have to be considered. The

Table 3.3: Secondary time intervals for the evaluation of the CSRL **X**-operator.

	$\varrho_s = 0$		$\varrho_s > 0$			
$I \setminus J$	$[0,\cdot]$	$(y,\cdot]$	$[0,\infty)$	$[0,y]$	$(y,y']$	(y,∞)
$[0,\infty)$	I	$\varnothing$	I	$[0,\frac{y}{\varrho_s}]$	$(\frac{y}{\varrho_s},\frac{y'}{\varrho_s}]$	$(\frac{y}{\varrho_s},\infty)$
$[0,t]$	I	$\varnothing$	I	$[0,min(t,\frac{y}{\varrho_s})]$	$(\frac{y}{\varrho_s},min(t,\frac{y'}{\varrho_s})]$	$(\frac{y}{\varrho_s},t]$
$[t,t']$	I	$\varnothing$	I	$[t,min(t',\frac{y}{\varrho_s})]$	$(min(t,\frac{y}{\varrho_s}),min(t',\frac{y'}{\varrho_s})]$	$(min(t,\frac{y}{\varrho_s}),t']$
$[t,\infty)$	I	$\varnothing$	I	$[t,\frac{y}{\varrho_s}]$	$(min(t,\frac{y}{\varrho_s}),\frac{y'}{\varrho_s}]$	$(min(t,\frac{y}{\varrho_s}),\infty)$

evaluation of the **X**-operator generally requires a single multiplication of a vector and the matrix $\mathbf{P}$.

U-operator

With regard to the **U**-operator, the characteristics of the time and reward intervals enable to distinguish the following four sub-logics of CSRL. For a detailed discussion I refer to [40].

Continuous Stochastic Logic (CSL). CSL is the CSRL subset with $J = [0,\infty)$ and $I \neq [0,\infty)$ meaning that there are no constraints concerning the accumulated reward. In this case I omit J for readability and do not consider the reward structure ϱ. The model M is characterized by the CTMC C.

CSL model checking can be completely reduced to the numerical computation of standard CTMC measures in a modified CTMC [9, 11]. Depending on the specified time bounds, certain states will be made absorbing by setting their rate to zero. Doing so, the probability to take one of the satisfying paths in the original model is equal to the probability of reaching a Φ-state in the derived model [9]. The computation of these probabilities can be realized with the standard methods I sketched in Section 3.3. In same cases the derived model becomes inhomogeneous and it is necessary to perform a two phase computation with different models. In the remainder, $M[S']$ denotes the Markov model derived from M by making all states in $S' \subset \mathcal{S}$ absorbing.

The following path formulas have to be considered:

1) $\mathbf{\Phi U^{[0,\tau]} \Psi}$: In this case it holds that

$$Prob_s^M(\Phi \mathbf{U}^{[0,\tau]} \Psi) = \sum_{s' \in Sat(\Psi)} \pi_{s,\tau}^{M[Sat(\neg\Phi \vee \Psi)]}(s'). \tag{3.5}$$

A simple transient analysis is necessary while making $(\neg\Phi \vee \Psi)$-states absorbing.

2) $\mathbf{\Phi U^{[\tau,\tau]} \Psi}$: Paths leading to $(\neg\Phi \wedge \neg\Psi)$-states are cut and it holds

$$Prob_s^M(\Phi \mathbf{U}^{[\tau,\tau]} \Psi) = \sum_{s' \in Sat(\Psi)} \pi_{s,\tau}^{M[Sat(\neg\Phi \wedge \neg\Psi)]}(s'). \tag{3.6}$$

3) $\mathbf{\Phi U^{[\tau,\tau']} \Psi}$: In this case we deal with an inhomogeneous CTMC, as the transition relation $\mathbf{R}$ is time-dependent. Before reaching the lower time bound, it is possible to leave a Ψ-state, provided that it fulfills Φ. After reaching the lower time bound, also Ψ-states become absorbing. The computation requires two steps. It holds that

$Prob_s^M(\Phi \mathbf{U}^{[\tau,\tau']} \Psi)$

$$= \sum_{s' \in Sat(\Phi)} \left(\pi_{s,\tau}^{M[Sat(\neg\Phi])}(s') \cdot \sum_{s'' \in Sat(\Psi)} \pi_{s',(\tau'-\tau)}^{M[Sat(\neg\Phi \vee \Psi)]}(s'') \right). \tag{3.7}$$

4) $\mathbf{\Phi U^{[\tau,\infty)} \Psi}$: This case is similar to the previous one as we deal again with an inhomogeneous CTMC. However the first step is now the computation of the limiting probabilities in the CTMC $M[Sat(\neg\Phi \vee \Psi)]$.

$Prob_s^M(\Phi \mathbf{U}^{[\tau,\infty)} \Psi)$

$$= \sum_{s' \in Sat(\Phi)} \left(\pi_{s,\tau}^{M[Sat(\neg\Phi)]}(s') \cdot \sum_{s'' \in Sat(\Psi)} \pi_{s'}^{M[Sat(\neg\Phi \vee \Psi)]}(s'') \right). \tag{3.8}$$

Stochastic Logic (SL). In the Stochastic Logic there are no constraints concerning time and accumulated reward and I omit I and J. The evaluation is reduced to the computation of $\underline{\pi}_{S'}$ for $S' = Sat(\Psi)$ while making all $(\neg\Phi \vee \Psi)$-states absorbing (see Section 3.3.2).

Please note that SL is not equal to CTL. One could interpret $\mathbf{E}$ as $\mathcal{P}_{>0}$ and $\mathbf{A}$ as $\mathcal{P}_{\geq 1}$ but in general it holds that

$$A[\Phi \mathbf{U} \Psi] \not\equiv \mathcal{P}_{\geq 1}[\Phi \mathbf{U} \Psi],$$

as illustrated in Figure 3.7.

Figure 3.7: Due to the loops it is qualitatively possible to stay forever in the Φ-state. Without any time and reward constraints the probability to observe this single infinite path is zero and is independent of the actual transition rates. Each probable execution will stay some time in the Φ-state and will finally move to the Ψ-state. Thus the model fulfills the CSL formula $\mathcal{P}_{\geq 1}[\Phi \mathbf{U} \Psi]$ but not the CSL formula $\mathbf{A}[\Phi \mathbf{U} \Psi]$.

Continuous Reward Logic (CRL). CRL is the dual to CSL as time does not influence the truth of the given formula.

Under certain conditions it is possible to interchange the role of time and accumulated reward in the model and the formula. This enables to evaluate a CRL formula using the related CSL evaluation procedure and a modified MRM. The requirements for this approach, known as the principle of duality [40], is a **total** rate reward function, meaning that for each state the earned reward is positive. The accumulated reward of each path is uniquely determined by the sojourn times of its states. Thus, it is possible to interchange I and J in the CRL formula and to check the resulting CSL formula with respect to the model $C' = [\mathcal{S}, \mathbf{R}', L, s_0]$ where all transition rates are rescaled by the related rate reward: $\forall s \in \mathcal{S} : \mathbf{R}'(s, s') = \frac{\mathbf{R}(s,s')}{\varrho_s}$.

Continuous Stochastic Reward Logic (CSRL). The remaining types of formulas can be evaluated similar to CSL formulas but require to compute the performability measure [55, 40, 8]. The CSRL formula $\Phi \mathbf{U}_J^I \Psi$ is encoded into the model. The two-phase approach described by Cloth [40] considers the performability measure of a possibly time and reward inhomogeneous MRM $M\langle \Phi \mathbf{U}_J^I \Psi \rangle$. The transition relation changes after exceeding both the lower time bound and the lower reward bound. Starting from this point a path may fulfill the given formula. Before this happens (phase 1), only $\neg\Phi$ states become absorbing and their rate reward becomes zero. After exceeding the lower bounds (phase 2), Ψ-states become additionally absorbing. When starting phase 2, care must be taken with

$(\neg\Phi \wedge \Psi)$- states as they do not contribute to the probability mass. Cloth proposes to create copies, which are only reachable in phase 1. It remains to compute $\upsilon^{M\langle\Phi\mathbf{U}^I_J\Psi\rangle}_{s,\sup(I),\sup(J)}(Sat(\Psi))$ for all reachable states, as it holds (Theorem 3 in [40])

$$Prob^M_s(\Phi\mathbf{U}^I_J\Psi) = \sum_{s'\in Sat(\Psi)} \upsilon^{M\langle\Phi\mathbf{U}^I_J\Psi\rangle}_{s,\sup(I),\sup(J)}. \tag{3.9}$$

This general presentation leaves open how to compute the distribution of the accumulated reward, which is actually a difficult problem. In this context [42] discusses five more or less general algorithms. Markovian Approximation, which I sketched in Section 3.2.2, is one of the methods applicable also in the inhomogeneous case.

In the following I will describe an approach for the model checking of CRL and CSRL formulas which translates the problem specification to CSL and considers the SPN $N^{\mathcal{A}}$, which I specified in Section 3.2.2.

Model checking based on Markovian Approximation. The core idea of the Markovian Approximation is to encode a discretization of the accumulated reward induced by the MRM $M = [C, \varrho]$ into the states of the underlying CTMC C. In Section 3.2.2 I discussed how to derive a bounded high-level description of the resulting CTMC $C^{\mathcal{A}}$ by means of the SPN $N^{\mathcal{A}}$, where each marking of the place p_y represents a certain level of the discretized accumulated reward. We defined $l = \lfloor \frac{y}{\Delta} \rfloor + 2$ reward levels for a specified discretization constant Δ and a reward bound y. The lth level represents the reward interval $(> y, \infty)$.

With this background I propose to move the reward constraints J into the C(S)RL formula. To illustrate this simple transformation, let us consider the CSRL path formula $\psi = \Phi\mathbf{U}^{[\tau,\tau']}_{(y,y']}\Psi$. A ψ-path σ must fulfill the following three conditions:

1. σ is a $(\Phi\mathbf{U}\Psi)$-path.
2. The related Ψ-state is reached within the time interval I.
3. Its accumulated reward is in J
   a) when reaching the Ψ-state after exceeding τ, **or**
   b) when passing τ as far as $\sigma(\tau)$ is a $(\Phi \wedge \Psi)$-state.

With respect to Δ, the condition 3 can be encoded into the state formula Ψ by mapping y and y' to the markings (reward levels) of the place p_y. The boundary

level representing a reward greater than y' is $\lfloor \frac{y'}{\Delta} \rfloor + 1$. This gives the state formula

$$\Psi_J = \Psi \wedge p_y > \lfloor \frac{y}{\Delta} \rfloor \wedge p_y \leq \lfloor \frac{y'}{\Delta} \rfloor$$

which replaces Ψ in the original formula. Following this idea we can generally

CSRL	CSL
$\Phi \mathbf{U}^I_{[y,y]} \Psi$	$\Phi \mathbf{U}^I (\Psi \wedge p_y = \lfloor \frac{y}{\Delta} \rfloor)$
$\Phi \mathbf{U}^I_{[0,y]} \Psi$	$\Phi \mathbf{U}^I (\Psi \wedge p_y \leq \lfloor \frac{y}{\Delta} \rfloor)$
$\Phi \mathbf{U}^I_{(y,y']} \Psi$	$\Phi \mathbf{U}^I (\Psi p_y > \lfloor \frac{y}{\Delta} \rfloor \wedge p_y \leq \lfloor \frac{y'}{\Delta} \rfloor)$
$\Phi \mathbf{U}^I_{(y,\infty)} \Psi$	$\Phi \mathbf{U}^I (\Psi p_y > \lfloor \frac{y}{\Delta} \rfloor)$

Table 3.4: The CSL representation of relevant CSRL formulas for the approximating SPN $N^{\mathcal{A}}$.

remove J while replacing Ψ by Ψ_J. The different situations are shown in Table 3.4. The rest is CSL model checking!

Correctness. Let us take once again the CSRL formula $\mathcal{P}_{\bowtie p}[\Phi \mathbf{U}^{[\tau,\tau']}_{(y,y']} \Psi]$ to show exemplified that this CSL-based approach emulates that in [40]. We move the reward bounds into the state formulas and get the CSL formula

$$\mathcal{P}_{\bowtie p}[\Phi \mathbf{U}^{[\tau,\tau']} (\Psi \wedge p_y > \lfloor \frac{y}{\Delta} \rfloor \wedge p_y \leq \lfloor \frac{y'}{\Delta} \rfloor)].$$

To approximate the probability $Prob^M_s(\Phi \mathbf{U}^{[\tau,\tau']}_{(y,y']} \Psi)$, we apply the CSL model checking algorithm with the derived formula and the net $N^{\mathcal{A}}$. Therefor we replace in Equation 3.7 Ψ by the new state formula $\Psi_{(y,y']}$ and obtain
$Prob^{C^{\mathcal{A}}}_s(\Phi \mathbf{U}^{[\tau,\tau']} \Psi_{(y,y']})$

$$= \sum_{s' \in Sat(\Phi)} \left(\pi^{C^{\mathcal{A}}[Sat(\neg\Phi)]}_{s,\tau}(s') \cdot \underbrace{\sum_{s'' \in Sat(\Psi)} \pi^{C^{\mathcal{A}}[Sat(\neg\Phi \vee (\Psi \wedge p_y > \lfloor \frac{y}{\Delta} \rfloor \wedge p_y \leq \lfloor \frac{y'}{\Delta} \rfloor))]}_{s',(\tau'-\tau)}(s'')}_{phase\ 2} \right).$$

The term

$$\sum_{s'' \in Sat(\Psi)} \pi^{C^{\mathcal{A}}[Sat(\neg\Phi \vee (\Psi \wedge p_y > \lfloor \frac{y}{\Delta} \rfloor \wedge p_y \leq \lfloor \frac{y'}{\Delta} \rfloor))]}_{s',(\tau'-\tau)}(s'')$$

represents the second phase in Cloth's approach. $C^{\mathcal{A}}[\neg\Phi \vee (\Psi \wedge p_y > \lfloor \frac{y}{\Delta} \rfloor \wedge p_y \leq \lfloor \frac{y'}{\Delta} \rfloor)]$ approximates the MRM where in addition to the $(\neg\Phi)$-states all Ψ-states have become absorbing, as far as the lower reward bound has been exceeded, approximated by $p_y > \lfloor \frac{y}{\Delta} \rfloor$. The reward bound $\sup(J)$ in Equation 3.9 is approximated by $p_y \leq \lfloor \frac{y'}{\Delta} \rfloor$. That also the lower time bound has been exceeded can be seen when the transient analysis is done for the time $\tau' - \tau$. The computed probabilities are multiplied with the probability to be in a Φ-state at the end of phase one, where we consider the CTMC $C^{\mathcal{A}}[\neg\Phi]$. A separate consideration of $(\neg\Phi \vee \Psi)$-states is not necessary. This treatment of C(S)RL formulas yields naturally a model checking procedure for the formula $\Phi \mathbf{U}^{[\tau,\infty)}_{(y,\infty)} \Psi$ for which [40] does not present a solution. We can specify the CSL path formula $\Phi \mathbf{U}^{[\tau,\infty)}(\Psi \wedge p_y > \lfloor \frac{y}{\Delta} \rfloor)$ and have

$$Prob^M_s(\Phi \mathbf{U}^{[\tau,\infty)}_{(y,\infty)} \Psi)$$

$$\approx \sum_{s' \in Sat(\Phi)} \left(\pi^{C^{\mathcal{A}}[\neg\Phi]}_{s,\tau}(s') \cdot \sum_{s'' \in Sat(\Psi)} \pi^{C^{\mathcal{A}}[\neg\Phi \vee (\Psi \wedge p_y > \lfloor \frac{y}{\Delta} \rfloor)]}_{s'}(s'') \right).$$

In this case the second phase is the computation of the limiting probability to reach a Ψ-state in the subset of states which represents the equivalence class $S_{>y}$.

The proposed approach possesses several pros and cons which should be mentioned here. I will not provide solutions here for the cons.

Pros. Markovian Approximation has been shown to possess in general a good performance. It can be applied even for huge state spaces [41, 40, 107]. It does not require a total reward function. As the rewards are encoded into the model as well into the formula, the technique enables the use of existing CSL model checkers.
From my point of view the most important advantage is the "easy to get" high-level description by means of the SPN $N^{\mathcal{A}}$ which enables the efficient matrix representation which I will discuss in Chapter 4.

Cons. For the time being the error introduced by the Markovian Approximation is unknown [40]. The proposed encoding may introduce a further inaccuracy due to the discretization of the interval bounds. The encoding of the reward accumulation into the CTMC is expected to produce a highly stiff model, where the rates of the state transitions differ considerably. Stiffness affects the numerical

methods as uniformization with respect to the accuracy and the number of required iterations. A further disadvantage is the restriction to rate-based Markov reward models.

Example 8

We can use CSRL to formalize interesting questions concerning reward augmented SPNs (SRNs) as for instance:

- Availability - The CSL formula

  $$\mathcal{P}_{>p}[\mathbf{F}^{[\tau,\tau]} ap_{up}]$$

  asks: "For which states, when starting there, is the probability greater than some bound p to be at time point τ in an up-state?"

- Reliability - The CSL formula

  $$\mathcal{P}_{>p}[\mathbf{G}^{[0,\tau]} ap_{up}]$$

  asks: "For which states, when starting there, is the probability greater than some bound p to visit up to time point τ only up-states?"

- For the running example (See Figure 3.3) we ask for instance: "What is the probability to reach within time τ a state where the buffer $b1$ is full, whereby $b2$ remains empty and the number of consumptions is bounded by some B?" by using the reward structure

  $$\varrho_s^{consumption} = \begin{cases} cr & \text{if } s \in enabled(consume) \\ cr & \text{if } s \in enabled(consume\_fetch\_b2\_1) \\ (cr \cdot f2) & \text{if } s \in enabled(consume\_fetch\_b2\_2) \\ cr & \text{if } s \in enabled(consume\_fetch\_b1\_1) \\ (cr \cdot f1) & \text{if } s \in enabled(consume\_fetch\_b1\_2) \\ 0 & \text{otherwise .} \end{cases}$$

  and the CSRL formula

  $$\mathcal{P}_{=?}[b2 < 1 \mathbf{U}^{[0,\tau]}_{[0,B]} b1 = N].$$[7]

★

[7] The $\mathcal{P}_{=?}$ operator is not a standard CSRL operator. It is syntactic sugar to get the probability of the initial state.

3.5 Summary

In this chapter I recalled (generalized) stochastic Petri nets ((G)SPN), whose semantics are Continuous-time Markov chains (CTMC), and numerical methods to compute important probability distributions and secondary measures. As a useful extension of the SPN formalism I briefly presented rate-based stochastic reward nets (SRN) whose semantics is a special type of Markov reward model (MRM). The considered SRN can be approximated by SPN. I presented the Continuous Stochastic Reward Logic (CSRL) as an advanced analysis approach of (G)SPN and SRN. I sketched the basic ideas of CSRL model checking and proposed an approximative model checking approach which is purely based on CSL, a proper CSRL subset, and the SPN formalism.

4 Advanced Matrix Representation

The numerical methods presented in Section 3.3 have in common that they rely in some sense on the multiplication of an $|\mathcal{S}|\times|\mathcal{S}|$-matrix and an $|\mathcal{S}|$-vector, where $\mathcal{S}$ is the set of reachable states of the investigated model. Although this operation is very simple from an algorithmic view point, the state space explosion turns it in many cases into a serious challenge.

In this chapter I present the main contribution of this thesis, a symbolic on-the-fly approach to enumerate the entries of the rate matrix of a CTMC. The idea is to compute the matrix entries from the high-level model description and the related state space, instead of storing them in a dedicated data structure. It was first proposed in [45] for an explicit storage of the states and suffers from the state space explosion. For this reason I pick up the basic idea, but discuss it in a symbolic setting.

I have introduced the necessary ingredients in Chapter 2 and Chapter 3 - basically stochastic Petri nets as the high-level description of Continuous-time Markov chains and Interval Decision Diagrams as a data structure to represent symbolically sets of states.

Several advanced matrix representation techniques have been investigated in the past with notable results. Before I present the details of my approach, I briefly review some important encoding techniques of Markov chains, which often enable a compact matrix storage and which are deployed in public available tools.

4.1 Classical Sparse Matrix Representation

We have seen that the numerical analysis of Markov models can be carried out by applying matrix-vector and vector-matrix multiplications. The related matrices are generally extremely sparse, meaning that the vast majority of the potential $|\mathcal{S}|\times|\mathcal{S}|$ entries is zero. As I assume that the Markov model is induced by a Petri net, the number of non-zero entries per row is bounded by the number of Petri net transitions, which is typically much smaller than the number of reachable

states.

It is obviously worth considering this sparsity for the implementation of numerical Markov chain solvers. There are several established storage schemes for sparse matrices which differ in the internal representation and the supported operations, e.g. the modification of the matrix in terms of insert and delete operations or the support of efficient access to rows or columns.

A frequently used scheme is the *Compressed Sparse Row* (CSR) format. As the name may suggest, it enables an efficient access to the matrix rows. I will consider it also for an experimental comparison in Section 4.3.4.

For an $n \times m$-matrix $\mathbf{M}$ with nnz non-zero entries (in the scenario on hand, it holds $n = m = |\mathcal{S}|$), we can encode $\mathbf{M}$ using three arrays row, col and val. The array row stores at the ith position the position j of the array col. Starting with position j this array contains the column indices of the non-zero elements in row i. The related values are stored in the array val also starting at position j for the elements of the ith row. The number of non-zero elements in row i is $row[i+1] - row[i]$. This requires the arrays col and val to be of size nnz. The array row is of size $n+1$. As I assume the value type to be *double precision* and the index type to be *unsigned int*, this requires $nnz \cdot (sizeof(double) + sizeof(unsigned)) + (n+1) \cdot sizeof(unsigned)$ bytes to store $\mathbf{M}$. A dense encoding using a two-dimensional array would require $n \cdot m \cdot sizeof(double)$ bytes. Figure 4.1 shows the CSR encoding of the rate matrix given in Example 5.

pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
row	0	3	5	8	11	15	17	18	19	21	23	25	26	28	29																
col	1	5	10	6	11	0	7	12	0	8	13	2	3	9	14	2	6	0	5	4	5	7	8	3	11	0	4	10	10	12	13
val	1.0	0.9	0.1	0.9	0.1	1.0	0.45	0.05	1.0	0.45	0.05	0.2	0.8	0.3	1/3	2.0	1.0	2.0	1.0	2.0	1.0	0.2	0.8	3.0	1.0	3.0	3.0	1.0	1.0	0.2	0.8

Figure 4.1: The CSR format representation of the matrix $\mathbf{R}$ in Example 5.

Given a matrix $\mathbf{M}$ in CSR format, the multiplication $\underline{r} = \mathbf{M} \cdot \underline{v}$ can be realized as shown in Algorithm 12. If it is required to efficiently extract the columns of the matrix we can use the *Compressed sparse column* format, which is encoded analogously.

Algorithm 12 (Matrix-Vector multiplication – CSR)

```
1  func multiply(M : CSR matrix , v, r : vector of ValueT)
2      i, i_nnz : IndexT
3      for 0 ≤ i < M.n do
4          i_0 := M.row[i]
5          i_nnz := M.row[i + 1]
6          r[i] := 0
7          for i_0 ≤ j < i_nnz do
8              r[i] := r[i] + M.val[j] · v[M.col[j]]
9          end
10     end
11 end
```

4.2 State of the Art

CSR is a widely used scheme to store sparse matrices. However with regard to the encoding of large Markov models, the pure consideration of sparsity will not suffice. Regularity, which results in redundancy, must be taken into account. There are basically two relevant approaches to encode the rate matrix of CTMCs [90] addressing this aspect. They are either based on Kronecker representations encoded by means of Matrix Diagrams or on symbolic representations using Multi-terminal Decision Diagram s (MTDD).

4.2.1 The Kronecker Algebraic Approach

The general idea. The basic idea of the Kronecker algebraic approach for the analysis of large CTMCs is to store the rate (or generator) matrix $\mathbf{R}$ (or $\mathbf{Q}$) as a Kronecker algebraic expression where the operands are matrices characterizing the individual components of a structured model.

A structured model M is made by composition of its components $\{M_1, \ldots, M_K\}$, each with the state space $\mathcal{S}_i$. State transitions are induced by a set of events $\mathcal{E}$, which are either local to a certain component or synchronize several components. The composition of the components generates the potential state space $\hat{\mathcal{S}} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_K$ which is in general much larger than the set of reachable states of

M. The rate matrix over the potential state space $\hat{\mathbf{R}}$ can be specified as [91]

$$\hat{\mathbf{R}} = \sum_{e \in \mathcal{E}} \hat{\mathbf{R}}^e = \sum_{e \in \mathcal{E}} (\mathbf{W}_K^e \otimes \ldots \otimes \mathbf{W}_1^e) = \sum_{e \in \mathcal{E}} \bigotimes_{k=K}^{1} \mathbf{W}_k^e.$$

The matrix $\mathbf{W}_k^e$ represents the state transitions induced by the event $e \in \mathcal{E}$ with regard to the component M_k. For a local event e, which is responsible only for state transitions within the component M_i the matrix $\mathbf{W}_j^e$ equals the identity matrix I_n for all $j \neq i$, with n being the number of states in M_j . $\otimes$ is the Kronecker product defined for the matrices $\mathbf{A}^{k \times l}$ and $\mathbf{B}^{m \times n}$ as the $k \cdot m \times l \cdot n$ matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{pmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \cdots & a_{1,\ell}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \cdots & a_{2,\ell}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k,1}\mathbf{B} & a_{k,2}\mathbf{B} & \cdots & a_{k,\ell}\mathbf{B} \end{pmatrix},$$

as it is illustrated in Example 9. At first Plateau [97] used the Kronecker representation to compute the stationary distribution of stochastic automata networks (SAN). Later Donatelli [47] considered it for so-called superposed generalized stochastic Petri nets.

Example 9

For the matrices

$$\mathbf{M}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \mathbf{M}_2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ and } \mathbf{M}_3 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

we can specify, for instance, the Kronecker products

$$\mathbf{M}_4 = \mathbf{M}_1 \otimes \mathbf{M}_2 = \begin{pmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 & 0 \\ 7 & 8 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 4 & 5 & 6 \\ 0 & 0 & 0 & 7 & 8 & 9 \end{pmatrix}$$

and

$$\mathbf{M}_5 = \mathbf{M}_4 \otimes \mathbf{M}_3 = \mathbf{M}_1 \otimes \mathbf{M}_2 \otimes \mathbf{M}_3 = \begin{pmatrix} 1 & 0 & 2 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 4 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 10 & 0 & 12 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 8 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 14 & 0 & 16 & 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 4 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 5 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 10 & 0 & 12 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 8 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 14 & 0 & 16 & 0 & 18 \end{pmatrix}.$$

★

Improvements. The early attempts suffered from the fact that the Kronecker representation encodes the rate matrix indexed by the potential state space of the underlying model. Though this generally permits a very compact representation of huge matrices, numerical computation based on it requires computation vectors in the dimension of the potential state space, too.

Improvements [19, 70][1] enabled to consider just the reachable states $\hat{\mathcal{S}}$ but introduced a logarithmic runtime overhead to identify the actual model states using special search trees. This overhead was eliminated in [32] by replacing search trees with sparse **Multi-valued Decision Diagrams** (MDD) with offset-labels. In general, a Multi-valued Decision Diagram [91] is a data structure similar to an Interval Decision Diagram. The difference is that each edge is labeled with a natural number, representing the variable value. A non-terminal MDD node represents a value range $[0, r]$ for some variable with exactly $r + 1$ outgoing edges, even if neighboring arcs refer to the same child. The sparse variant contains only variable values where the related outgoing edge can be extended to a path reaching the 1-terminal node. An IDD with all edge intervals having a width of *one* can be seen as an MDD. Compared with the IDD representation for the reachable states of the running example in Figure 2.3, the MDD representation would contain $N + 1$ outgoing arcs for the nodes labeled with variables $b1$ and $b2$. An MDD with offset-labels stores for each edge additionally the number of sub-states, which are reachable over its left siblings. This enables to compute for

[1] not exhaustive

each state a lexicographic index as the sum of the edge-offsets of the related path. I will explain the idea of an offset-augmentation for Interval Decision Diagrams in Section 4.3.

In [32] the authors introduce further **Matrix Diagrams** (*MxD*) as a data structure to encode Kronecker expressions. Matrix diagrams are decision diagrams where matrices are associated to the nodes. In non-terminal nodes, each non-zero matrix element stores in addition to a real value a pointer to a node at the lower layer in turn representing a matrix. Matrix diagrams support operations as addition, sub-matrix extraction and column extraction. The extraction of sub-matrices considers two MDDs representing the set of column and row indices (states) which should be contained in the resulting *MxD*. This allows in a first step to generate an *MxD* representation of a Kronecker expression over the potential state space. In a second step one uses the offset-labeled MDD representation of the reachable states to extract an *MxD* representation containing only the rows and columns of the reachable states. For a detailed description of this approach I refer to [91]. The ability to extract columns allows the application of the method of Gauss-Seidel without further runtime overhead.

The Kronecker algebraic approach was first applied to compute stationary distributions using the Power method[2]. Later it was used to realize Jacobi and Gauss-Seidel solvers, for transient analysis and also for CTL [72] and CSL [20] model checking. Kronecker algebraic engines are part of tools as the APNN-Toolbox [21], SMART [29] and PEPS [15]. They are available in SMART and Möbius [43] in the form of matrix diagrams.

4.2.2 Multi-terminal Decision Diagram-based Approaches

A second important group of Markov chain representation techniques is based on so-called Multi-terminal Decision Diagrams (MTDDs). MTDDs generalize common decision diagrams such that they can possess more than two terminal nodes. In general a (reduced ordered) decision diagram over n variables represents a function

$$f^n : \mathcal{X}^n \to \mathcal{Y}$$

whereby

- $\mathcal{X} = \mathbb{B}$ and $\mathcal{Y} = \mathbb{B}$ for BDDs
- $\mathcal{X} = \mathbb{N}$ and $\mathcal{Y} = \mathbb{B}$ for IDDs and MDDs.

[2]The Power method is a simple iterative method. It can be seen as the uniformization method - without truncation and accumulation of Poisson weights.

With MTDDs, $\mathcal{Y}$ can be any finite set, for instance a finite subset of $\mathbb{R}_{\geq 0}$. In combination with the explicit state transition representation sketched in Section 2.2.2, it is possible to encode symbolically real-valued matrices. I will illustrate the underlying ideas with Multi-Terminal **Binary** Decision Diagrams (MTBDD). Binary Decision Diagrams (BDD) can be seen a special class of IDDs where each node labeled with a variable $x_l \in X$ represents the decision for x_l to be either $false/zero$ or $true/one$. Each non-terminal node has three outgoing arcs label with $[0,1), [1,2)$ and $[2,\infty)$. The latter refers always the 0-terminal node. A path from the BDD root to the 1-terminal node can be interpreted as a $|X|$-dimensional bit vector. MTBDDs can have more than two terminal nodes.

The general idea. The symbolic representation of vectors and matrices based on MTBDDs traces back to [39]. An n-dimensional vector $\underline{v} \in \mathcal{Y}^n$ is treated as a mapping from the set of integers (indices) $\{0,\ldots,n-1\}$ to $\mathcal{Y}$. For an index $0 \leq i < n$ the binary encoding is essentially a bit vector $\underline{v}^i \in \mathbb{B}^m$, with $m = \lceil ld(n-1) \rceil$ the smallest number of bits required for the binary encoding of $n-1$. This mapping can be encoded by a MTBDD over the set of variables $X = \{x_1,\ldots,x_m\}$. Each variable represents a bit. A path through the MTBDD gives the sequence of arc labels defining the binary encoding of some index $0 \leq i < n$. The related terminal node represents the vector entry $\underline{v}[i]$.

The idea to encode a matrix $\mathbf{M} \in \mathcal{Y}^{n \times n}$ is very similar[3]. $\mathbf{M}$ is treated as a mapping from $\{0,\ldots,n-1\} \times \{0,\ldots,n-1\}$ (index pairs) to $\mathcal{Y}$. This requires a second set of variables $X' = \{x'_1,\ldots,x'_m\}$. Now a path represents

1. the binary encoding of the row index i in terms of the sequence of the labels of arcs starting in nodes labeled with a variable from X (row variables),
2. the binary encoding of the column index j as the labels of arcs starting in nodes labeled with a column variable from X',
3. the actual matrix entry $\mathbf{M}(i,j)$ represented by the value of the reached terminal node.

Row and column variables are usually arranged in an interleaving order

$$\pi = x_1 < x'_1 < x_2 < x'_2 < \ldots < x_m < x'_m,$$

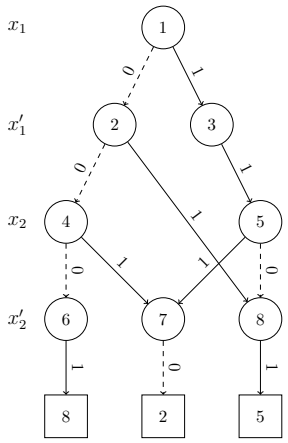
which

[3] The MTBDD-based encoding is not restricted to square-matrices. However, in this thesis matrices have this feature.

1. yields naturally a recursive matrix description [39], as it is illustrated in Example 10,
2. yields in general a compact MTBDD encoding [63].

Example 10

The MTBDD-based matrix representation of a sparse 4×4 matrix $\mathbf{M}$ taken from [96].



$$\mathbf{M} = \begin{pmatrix} 0 & 8 & 0 & 5 \\ 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 \end{pmatrix}$$

The path 1 -↠ 2 -↠ 4 -↠ 6 →$\boxed{8}$ represents the matrix entry $\mathbf{M}(0,1) = 8$. The outgoing arcs of the *row*-variables x_1 and x_2 represent the bit vector $(0,0)$, the out going arcs of *column*-variables x_1' and x_2' represent the bit vector $(0,1)$. The value of the matrix entry is the value of the reached terminal node.

Due to the interleaving of the *row* and *column*-variables, the node 4 represents the sub-matrix $\mathbf{M}_4 = \begin{pmatrix} 0 & 8 \\ 2 & 0 \end{pmatrix}$ and the node 3 the sub-matrix $\mathbf{M}_3 = \begin{pmatrix} 0 & 5 \\ 0 & 5 \\ 0 & 5 \\ 2 & 0 \end{pmatrix}$.

★

For the MTBDD encoding B_M of a matrix $\mathbf{M}$ and the MTBDD encoding B_v of a vector $\underline{v}$, the multiplication $\underline{r} = \mathbf{M} \cdot \underline{v}$ is realized by the standard BDD operation $APPLY$. The result is the MTBDD representation B_r of the vector $\underline{r}$.

Improvements. When encoding Markov chains by means of MTBDDs an important issue is the mapping of model variables to decision diagram variables, and thus indirectly the states to the indices.

Sophisticated approaches to derive the set of variables and their ordering in the context of Markov chain representation were studied in [63] and [44]. The key observation: it is worth considering the structural information of the high-level descriptions, in particular the components of process algebra descriptions [63] or Kronecker expressions [44]. In the latter work the authors propose a modular language where each model contains a set of integer variables. The actual model consists of the composition of the defined modules. Each integer variable var with range $\{0, \ldots, r\}$ is represented by a set of consecutive MTBDD variables $\{x_{var_1}, \ldots, x_{var_{\lceil ld(r) \rceil}}\}$, representing the binary encoding of possible values. This style of model representation yields variable orders which naturally consider dependencies between the individual model variables, an important aspect for the definition of static variable orders for decision diagrams. In this context the problem of the potential versus the reachable states arises similar to the Kronecker-based approach. If a model variable var with range $\{0, 1, 2\}$ is encoded by two Boolean MTBDD variables x_1 and x_2 the states with $var = 3$ are indeed unreachable states, but the MTBDD encoding of the rate matrix contains rows and columns which are related to such states. If the matrices as well as the computation vectors are encoded by MTBDDs, the unreachable states are represented by empty rows, empty columns and empty vector entries, respectively, and are ignored during the numerical computation.

However, Parker [96] showed that the pure MTBDD-based encoding of matrices and vectors performs only well in exceptional cases, were the computation vectors possess only a moderate number of distinct values and thus a moderate number of terminal nodes. As problem solution he proposed a hybrid approach, combining a MTBDD-based matrix encoding and an explicit storage of the computation vectors. Therefor the matrix-vector multiplication is based on a simple depth-first search traversal of the MTBDD. During the traversal the labels of the nodes and arcs are used to compute the index of the row and column index. When reaching a terminal node, the extracted value and the computed indices are passed to a generic function which reads and updates the dense vectors, depending on the applied operation. In this setting the computation vectors have a dimension in the size of the potential state space. To solve this second problem, Parker augmented the MTBDD with offsets[4] similar to [32] and improved the basic traversal algorithm such that it skips unreachable rows and columns during the index computation. As it extracts each individual matrix entry without exploiting the regularity in the data structure, the proposed traversal algorithm suffers from a very long runtime. Result caching, which usually guaranties the

[4] Offset-augmented decision diagrams fall into the class of Edge-valued decision diagrams studied in [80] (EVBDD) and [33] (EV+MDD).

efficiency of DD operations, is not applicable here. Parker solved this third problem by merging the MTBDD representation with an explicit storage of matrices.

His idea was to stop the MTBDD traversal at a predefined MTBDD layer. A pre-processing step converts the sub-matrices encoded by the MTBDD nodes of this layer to their explicit representation, using a sparse storage scheme as CSR. Thereby the related nodes become terminal nodes and their values are references to matrix instances. When the traversal algorithm reaches such a node, it computes the actual matrix entries by enumerating the entries in the referred sub-matrix and adding the computed row and column indices. With this approach there is a trade-off between memory and runtime costs depending on the chosen layer. See Chapter 6 for some figures. For a detailed description of the MTBDD-based approach I refer to [96].

MTBDDs are the most famous representatives of MTDDs. In the context of Markov chain analysis they were first used to compute limiting probability distributions in [53] (Power) and [63] (Jacobi and Gauss-Seidel). Their application in probabilistic model checking started with [7]. Further generalizations as MTMDDs [27][5] or Multi-terminal Zero-suppressed Binary Decision Diagrams [67] (MTZDDs) exist. MTDD-engines are used in tools as PRISM (MTBDDs) and SMART (MTMDDs).

Conclusions. Convinced of the potential of IDDs for efficient state space representation I could "invent" Multi-terminal IDDs to apply them to the numerical analysis of Markov chains. However, the following two reasons suggest to not follow this line:

- MTDDs are generally sensitive concerning the number of distinct matrix entries. The regularity in the decision diagram decreases with an increasing number of terminal nodes; the compactness of the data structures and thus the efficiency of related operations suffer. The source of a high number of terminal nodes may be
  - state-dependent (functional) rates, used for instance to describe kinetics in biochemical networks,
  - the use of secondary matrices as $\mathbf{P}$ or $\mathbf{P}^{\mathcal{U}}$, which often contain much more distinct values than the original rate matrix $\mathbf{R}$.
- The encoding of the row and column indices requires to double the number

[5] This depends on the definition of MDDs. In [27] MTMDDs are mentioned as special generalization. In [91] MDD can have multiple terminal nodes per definition.

of decision diagram variables. The consequences are again increasing size of the data structures and less efficient operations.

I will now present an alternative approach for the representation of the rate matrix of a CTMC $C_N = [\mathcal{S}, \mathbf{R}, s_0]$ induced by the SPN $N_S = [N, F]$. The approach builds on a ROIDD-based (Section 2.2) symbolic state space encoding of $\mathcal{S}$. An on-the-fly computation of the state transitions omits the doubling of decision diagram variables and the explicit storage of the distinct values of the matrix entries. It moves the whole computation of the matrix $\mathbf{R}$ to a DD traversal and exploits therefor the effects of firing the Petri net transitions T.

Crucial points of this approach are:

1. The enrichment of the previous ROIDD definition by index offsets. These offsets enable an efficient computation of the lexicographic state indices and are basically inspired by [32].
2. A special ROIDD operation generating for non-zero matrix entries the row and the column index as well as the value. This new operation is basically inspired by the function Fire in Section 2.2.2.
3. A strategy to reduce the overhead of redundant computations by truncating the traversal algorithm at a predefined layer. This is inspired by the approach reported in [96].

In the following section I will provide a detailed discussion of these three aspects.

4.3 IDD-based On-the-fly Matrix Generation

4.3.1 Enumeration of State Indices

Numerical computations with respect to the CTMC C_N require a mapping $\imath : \mathcal{S} \to \{0, \ldots, |\mathcal{S}| - 1\}$ of its state space to the index set. A depth-first search traversal of the ROIDD $G_{\mathcal{S}}$ representing $\mathcal{S}$ induces naturally a lexicographic indexing which meets our needs. However, an enumeration of the lexicographic predecessors when mapping state s to its index is of course not advisable. An efficient implementation of this mapping requires a slight extension of the basic data structure. For each arc of the ROIDD $G_{\mathcal{S}}$ it is necessary to remember the number of sub-states reachable over the previous sibling arcs. Similar to Offset-labeled MTBDDs [96], I denote the resulting data structure Offset-labeled Reduced Ordered Interval Decision Diagrams (OLROIDD). Beforehand I define

for a non-terminal ROIDD node v the following functions:[6]

1. $\mathrm{a}(v,c) \in E$ returns the j-th outgoing arc of v iff $c \in \mathrm{p}_j(v)$
2. $\mathrm{p}(v,c) \in I$ returns the interval of the arc $\mathrm{a}(v,c)$
3. $\mathrm{w}_j(v) \in \mathbb{N}_0,\ 1 \leq j \leq v_k$ returns the width of $\mathrm{p}_j(v)$
4. $\mathrm{c}(v,c) \in V$ returns $c_j(v)$ iff $c \in p_{x \geq j}(v)$
5. $\mathrm{r}_j(v) \in \mathbb{N}_0,\ 1 \leq j \leq v_k$ returns the number of sub-states reachable over the first $j-1$ outgoing arcs of v

$$\mathrm{r}_j(v) \quad = \quad \begin{cases} 0 & \text{if } j = 1 \\ \mathrm{r}_{j-1}(v) + \mathrm{w}_{j-1}(v) \cdot \mathrm{r}_{\mathrm{c}_{j-1}(v)_k}(\mathrm{c}_{j-1}(v)) & \text{otherwise.} \end{cases}$$

6. $\mathrm{r}(v) \in \mathbb{N}_0$ returns the number of all sub-states reachable from v

$$\mathrm{r}(v) \quad = \quad \begin{cases} 0 & \text{if } v = 0 \\ 1 & \text{if } v = 1 \\ \mathrm{r}_{v_k}(v) & \text{otherwise.} \end{cases}$$

7. $\mathrm{r}(v,c) \in \mathbb{N}_0$ returns the number of sub-states reachable over the outgoing arcs of v considering all assignments of the variable $var(v)$ **less** then c

$$\mathrm{r}(v,c) \quad = \quad \begin{cases} c \cdot r(c_1(v)) & \text{if } \mathrm{a}(v,c) = 1 \\ \mathrm{r}_{\mathrm{a}(v,c)-1}(v) + (c - \inf(\mathrm{p}(v,c))) \cdot r(c_{\mathrm{a}(v,c)}(v)) & \text{otherwise.} \end{cases}$$

I am now ready to extend the definition of ROIDDs (Section 2.2) by additional arc labels to enable the index computation for the encoded states. For the variables $X = \{ x_1, \ldots, x_n \}$ an OLROIDD is a tuple $\hat{G} = [G, O]$ where:

1. G is an ROIDD.
2. $O : E \to \mathbb{N}_0$ maps to the jth outgoing arc of node v the value $r_j(v)$, its index-offset.

Figure 4.2 shows an OLROIDD encoding the reachable states of the running example. The offsets are given next to the intervals indicated by the number sign.

[6] The functions make use of the functions defined in Section 2.2. The function names are abbreviated, e.g. $c_j(v)$ means $child_j(v)$.

States, paths and indices. For a finite set of states $S \subseteq \mathbb{N}^n$ with IDD representation G_S and a state $s = (s_1, s_2, \ldots, s_n) \in S$ the corresponding path $\rho_{G_S}(s)$ is the sequence $v_1 \xrightarrow{s_1} v_2 \xrightarrow{s_2} \ldots \xrightarrow{s_{n-1}} v_n \xrightarrow{s_n} 1$. The term $v_l \xrightarrow{s_l}$ provides the information that the outgoing arc $a(v_l, s_l)$ is selected with the variable assignment s_l. The lexicographic index of a state s is defined by its path $\rho_{G_S}(s)$ as

$$\imath(\rho_{G_S}(s)) = \sum_{1 \le l \le n} r(v_l, s_l). \tag{4.1}$$

Conventions. In what follows I am only interested in indices which are related to the set of reachable states $\mathcal{S}$ of a finite CTMC. So I assume that $\mathcal{S}$ is represented by the OLROIDD $\hat{G}_{\mathcal{S}}$.

Henceforth I consider only (Offset-labeled) Reduced Ordered Decision Diagrams and will use the abbreviation LIDD or IDD, respectively, for short. Further I will not distinguish between a state s, its path $\rho_{\hat{G}_{\mathcal{S}}}(s)$ in the LIDD $\hat{G}_{\mathcal{S}}$ and its index $\imath(\rho_{\hat{G}_{\mathcal{S}}}(s))$. I may use the terms as synonyms and abbreviate $\rho_{\hat{G}_{\mathcal{S}}}(s)$ as ρ_s and $\imath(\rho_{\hat{G}_{\mathcal{S}}}(s))$ as $\imath_s$.

Example 11

Figure 4.2 shows the LIDD encoding $\hat{G}_{\mathcal{S}}$ of the set of reachable states for the net in Figure 3.3 with $N = 2$ enriched with offsets.

Let us consider three different states, their paths in the LIDD and their lexicographic indices. The used variable order π is

$$to2 < to1 < item < ready < b2 < b1 < req < res.$$

For the state $s_1 = (0, 0, 0, 1, 0, 2, 0, 1)$ we extract the path

$$\rho_{s_1} = v_{15} \xrightarrow{0} v_{13} \xrightarrow{0} v_{11} \xrightarrow{0} v_9 \xrightarrow{1} v_8 \xrightarrow{0} v_6 \xrightarrow{2} v_5 \xrightarrow{0} v_2 \xrightarrow{1} 1,$$

and compute the lexicographic index

$$\imath_{s_1} = 0 + 0 + 0 + 0 + 2 + 1 \cdot 1 + 0 + 0 = 3.$$

For the state $s_2 = (1, 0, 1, 0, 0, 2, 0, 1)$ we extract the path

$$\rho_{s_2} = v_{15} \xrightarrow{1} v_{14} \xrightarrow{0} v_{12} \xrightarrow{1} v_{10} \xrightarrow{0} v_8 \xrightarrow{0} v_6 \xrightarrow{2} v_5 \xrightarrow{0} v_2 \xrightarrow{1} 1,$$

and compute the lexicographic index

$$\imath_{s_2} = 20 + 0 + 0 + 0 + 2 + 1 \cdot 1 + 0 + 0 = 23.$$

For the state $s_3 = (0,0,0,1,0,1,0,1)$ we extract the path

$$\rho_{s_3} = v_{15} \xrightarrow{0} v_{13} \xrightarrow{0} v_{11} \xrightarrow{0} v_9 \xrightarrow{1} v_8 \xrightarrow{0} v_6 \xrightarrow{1} v_5 \xrightarrow{0} v_2 \xrightarrow{1} 1,$$

and compute the lexicographic index

$$\imath_{s_3} = 0 + 0 + 0 + 0 + 2 + 0 \cdot 1 + 0 + 0 = 2.$$

★

Indexing of arbitrary states

In the following we will never compute the index for an isolated state. We will rather be interested in enumerating the indices of a given set of states. The offsets

Algorithm 13 (Enumerate – State indices)

```
1  proc AuxEnumerateIndices (v, v̂ : unsigned, index : IndexT, op : OP)
2     if v = 0 then return fi
3     if v = 1 then op(index) return fi
4     for 1 ≤ j ≤ |p(v)| do
5        for inf(I_j) ≤ s < sup(I_j) do
6           j' := a(v̂, s)
7           v̂' := c_j'(v̂)
8           if isValid(v̂') then
9              I_j' := p_j'(v̂)
10             index' := index + r_j'(v̂) + r(v̂') · (s − inf(I_j'))
                                 \_____ r(v̂,s) _____/
11             v' := c_j(v)
12             AuxEnumerateIndices(v',v̂',index',op)
13          fi
14       od
15    od
16 end
17 proc EnumerateIndices (G_S : IDD , op : OP)
18    AuxEnumerateIndices(G_S.root, Ĝ_S.root,0,op)
19 end
```

enable an efficient implementation of the function *EnumerateIndices* given in Algorithm 13. The function realizes the index computation for arbitrary subsets of $\mathcal{S}$ and is parameterized by the type of operation to be executed each time an index has been mapped to a state.

In particular the operation is represented by a functor.

The actual work is done in the auxiliary function *AuxEnumerateIndices* which extracts recursively all states encoded in the IDD G_S. The lexicographic index of a state is computed conform to Equation 4.1, where the single terms $r(v_l, s_l)$ are derived for each node v_l from the offsets of its outgoing arcs. The procedure extracts two paths for each state; one in the IDD representing an arbitrary set of states G_S and one in the LIDD $\hat{G}_{\mathcal{S}}$ representing $\mathcal{S}$. Note that S must not be contained in $\mathcal{S}$. Unreachable states are skipped during the index computation in line *8*. In general a reachable state s which is contained in different **finite**[7] state sets S and S' with different IDD representations is also represented by different paths. However, for each of these paths we can extract the unique representative $\rho_{\hat{G}_{\mathcal{S}}}(s)$ in the LIDD representation $\hat{G}_{\mathcal{S}}$.

Initialization of vector entries. An application of the procedure *EnumerateIndices* is the initialization of the solution vector in a linear system of equations which must be solved, e.g., to compute the limiting probabilities as discussed in Section 3.3.2. In this context we have to initialize all entries belonging to the states S_{yes} with 1. This can now be realized by the procedure *InitStates* and the related functor given in Algorithm 14.

Algorithm 14 (Init states)

```
1  struct InitStates
2      vec : vector of double
3      value : double
4
5      proc operator()(index : IndexT)
6          vec[index] := value
7      end
8  end
9
10 proc InitStates(G_S : IDD, vec : vector of double, value :double)
11     is : struct InitStates
12     is.vec := vec
13     is.value := value
14     EnumerateIndices(G_S,is)
15 end
```

[7] If an IDD represents a finite state space, each path to the 1-terminal node contains a node for each variable.

Selection of states. A further important application is the selection of states which are associated with certain values in a given vector; for instance all states with a probability below some threshold. We can extend the procedure *EnumerateIndices* for this purpose. The new function *SelectStates*, given in Algorithm 15, creates the IDD for the states contained in the state set G_S and fulfilling a specified predicate *op*. Here *op* is a functor realizing an arbitrary complex predicate[8]. The related recursive function *AuxSelectStates* returns a possibly newly created IDD node. When enumerating the outgoing arcs of an IDD node, the function fills two lists representing an interval partition and the related children, as we have seen in Algorithm 2. The use of *MakeNode* guaranties that the resulting IDD is reduced. Each time the traversal reaches the 1-terminal node, the function *AuxSelectStates* checks whether the extracted state fulfills the specified condition. If so, the return value is 1, otherwise 0. A naive implementation of this idea would consider each arc of the IDD, even if it is not possible to reach one of the satisfying states by passing it. Two simple checks before selecting an arc may prevent the Algorithm 15 from being exponential. Before calling the recursive auxiliary function we check whether all states reachable over the arc fulfill the required property (line *16*). If so, we just append its child, as it represents the related sub-states. Otherwise we further check whether there exists no satisfying state at all (line *18*). If so, we just add 0 to the list of children. If both checks fail, it is necessary to continue the traversal with the current arc (line *21*).

The precise realization of the related functions *checkAll* and *checkNone* depends finally on the selection policy, but means in most cases to evaluate the vector entries in the index interval $[index, index']$. To avoid the possibly expensive explicit evaluation of the index interval as long as possible we exploit the size of the maximal consecutive block and gap concerning the satisfying states. These information must be provided by a pre-processing step.

checkNone considers the maximal possible size of a gap between two satisfying states. If it is smaller than the number of reachable states of the current arc e_{v_j} (Figure 4.3b), at least one satisfying state (gray) must exist in the index interval. If the number of reachable states is indeed below the maximal size of a gap, it is worth checking all the reachable indices before continuing the traversal of the current arc (see Figure 4.3a).

Analogously, *checkAll* has to consider the maximal possible size of an index interval representing a block of satisfying states. If it is smaller than the number of reachable states of the current arc (Figure 4.4b), the check fails. Otherwise, each single vector entry in the interval $[index, index']$ must be tested explicitly.

[8] In C++, a functor whose operator() has a Boolean return value is called a predicate.

Algorithm 15 (Select states)

```
1  func AuxSelectStates (v, v̂ : unsigned , index: IndexT, op : OP)
2     if v = 0 then return 0 fi
3     if v = 1 then return op(index) fi
4     bounds,children : UList
5     for 1 ≤ j ≤ |p(v)| do
6        for inf(I_j) ≤ s < sup(I_j) do
7           bounds.append(s)
8           j' := a(v̂, s)
9           v̂' :=c_j'(v̂)
10          I_j' := p_j'(v̂)
11          index' := index + r_j'(v̂) + r(v̂') · (s − inf(I_j'))
                                     \_____ r(v̂,s) _____/
12          v' :=c_j(v)
13          child := 0
14          if not isValid(v̂') then
15             child := 0
16          elseif op.checkAll(index,index') then
17             child := v'
18          elseif op.checkNone(index,index') then
19             child := 0
20          else
21             child := AuxSelectStates(v',v̂',index',op)
22          fi
23          children.append(child)
24       od
25    od
26    return MakeNode(var(v),bounds,children)
27 end
28 func SelectStates (G_S : IDD , op : OP)
29    G_S' : new(IDD)
30    G_S'.root := AuxSelectStates(G_S.root,Ĝ_S.root,0,op)
31    return G_S'
32 end
```

The check is successful, if all entries satisfy the predicate (Figure 4.4a).

The number of states, which are reachable by passing IDD-arcs, decreases as the algorithm approaches the terminal nodes, and may fall below the maximal gap and block size, which increases at the same time the probability to perform the evaluation of the index interval $[index, index']$.

The gain of this optimization depends generally on the number and the distribution of the states satisfying the given property. The function *SelectStates* enables the implementation of functions as *StatesLessThan* in Algorithm 16. The function returns the subset of S represented by G_S for which the assigned value in vector v is less than l. Such functions are required to realize for instance the evaluation of the CSRL operators $\mathcal{P}_{\bowtie p}$ and $\mathcal{S}_{\bowtie p}$ (see Section 3.4).

Algorithm 16 (Select states – Less than)

```
1  struct Compare
2      vec : vector of double
3      value : double
4      op : OP
5
6      func operator()(index : IndexT)
7          return op(vec[index], value)
8      end
9
10     func checkAll()(index : IndexT, index′ : IndexT)
11         if index′ − index > blockSize then return false fi
12         forall i ∈ [index, index′] do
13             if not op(vec[index], value) then return false fi
14         od
15         return true
16     end
17
18     func checkNone()(index : IndexT, index′ : IndexT)
19         if index′ − index > gapSize then return false fi
20         forall i ∈ [index, index′] do
21             if op(vec[index], value) then return false fi
22         od
23         return true
24     end
25
26 end
27
28 func StatesLessThan(G_S : IDD, v : vector of double, l :double)
29     slt : struct Compare
30     slt.vec := v
31     slt.value := l
32     slt.op := <
33     return SelectStates(G_S,slt)
34 end
```

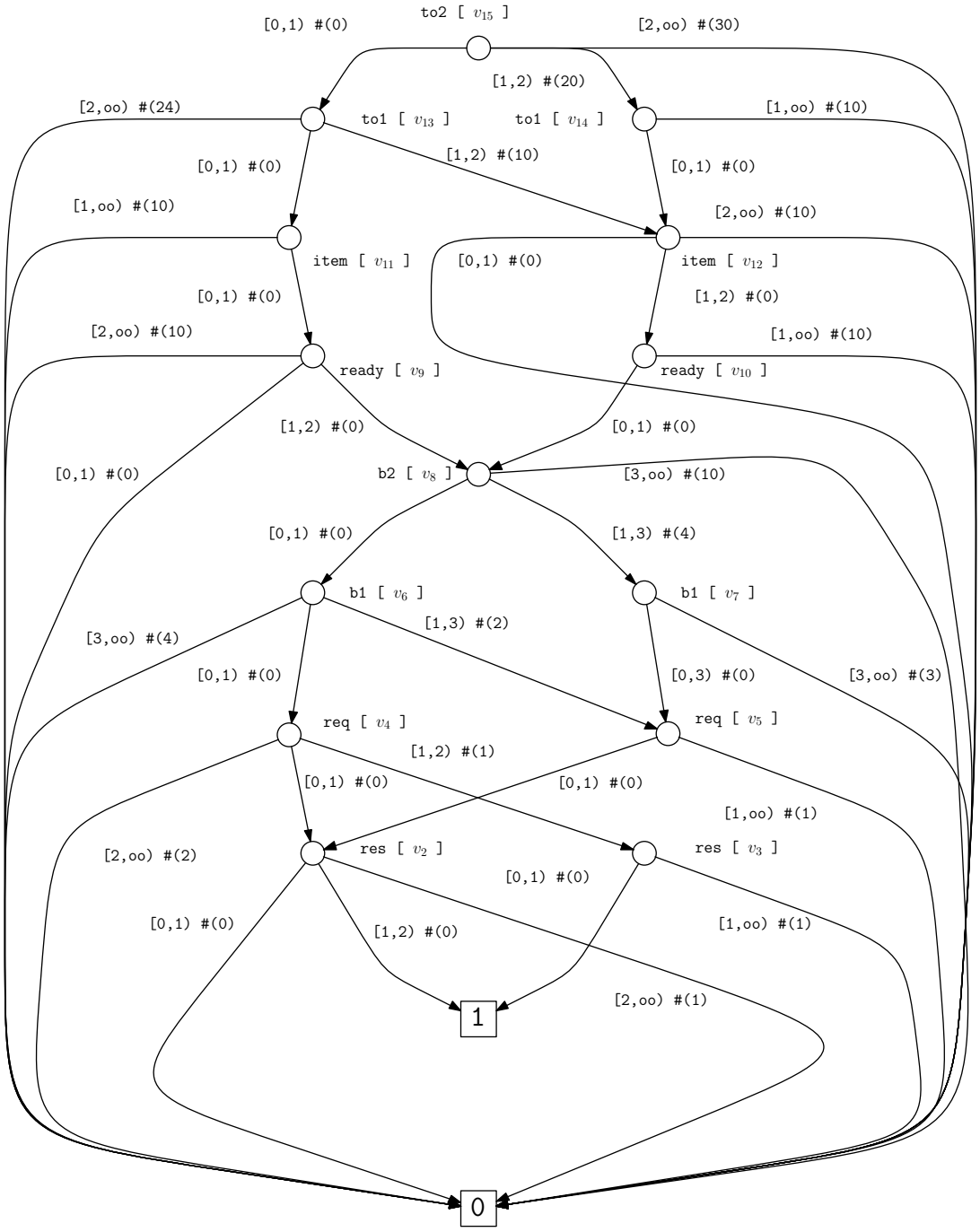


Figure 4.2: The LIDD encoding $\hat{G}_{\mathcal{S}}$ of the set of reachable states for the net in Figure 3.3 with $N = 2$ enriched with offsets. There are 30 reachable states.

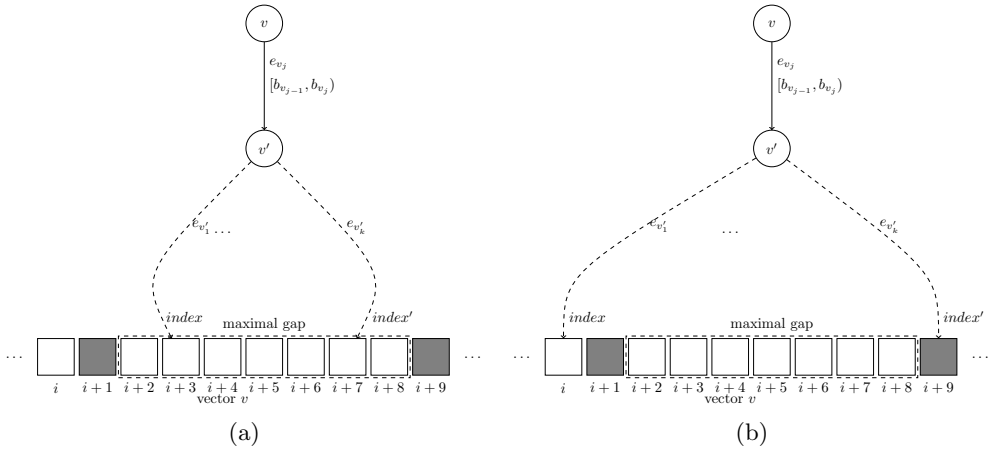


Figure 4.3: checkNone

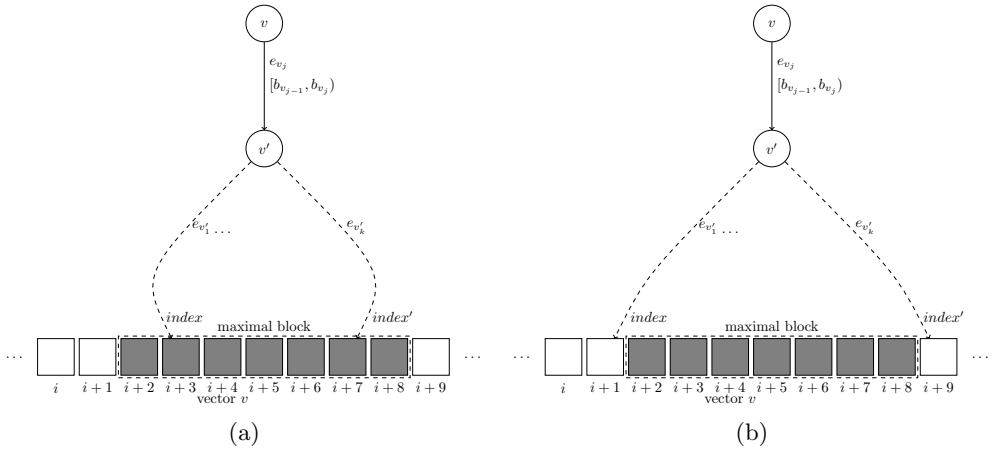


Figure 4.4: checkAll

4.3.2 Enumeration of State Transitions

Now, as we are able to enumerate the lexicographic indices related to a set of states, we can think about the enumeration of the state transitions, i.e. the matrix entries, each specified by a row and a column index and a real value. In previous approaches, the indices and the value of a matrix entry are computed either given an explicit (e.g. sparse matrices) or an implicit matrix representation (MxD, MTBDD). In any case, information concerning the row and column indices and the matrix entries are held in the same data structure.

Devours and Sanders [45] proposed an explicit matrix-free approach for several high-level formalisms, among them *GSPN*. The state transitions were computed on-the-fly, meaning that for a given state the effect of transition firing is utilized to determine the successor states and thereby the set of the related state transitions. To the best of my knowledge there is no published symbolic adaption of this approach.

The general idea. The operation *Fire* in Section 2.2.2 (and analogously *RevFire*) can be adapted to an operation which just extracts state transitions. Instead of creating for the states S and the transition t the IDD representation $G_{Fire(S,t)}$ of the successor set, it computes the indices and the assigned value of the corresponding state transitions. Algorithm 17 shows the procedure *EnumerateTransitions* which enumerates all state transitions induced by the firing of the transition t in the state set S.

The procedure extracts recursively the states encoded in the IDD G_S. Similar to Algorithm 13 and Algorithm 15, it extracts simultaneously a state s (src) in the LIDD $\hat{G}_{\mathcal{S}}$ and computes its lexicographic index $\imath_s$. Moreover it determines the successor state s' ($dest$) and its index $\imath_{s'}$. Therefor it exploits the enabling token interval ($a.enabled$) and the change of tokens ($a.shift$) on each place $p \in Env_t$ provided by the related element in the action list al. The algorithm simultaneously computes for each state transition $s \xrightarrow{t} s'$

1. the path of the state s in the IDD G_S,
2. the path of the state s in the LIDD $\hat{G}_{\mathcal{S}}$ and its lexicographic index $\imath_s$,
3. the path of the state s' in the LIDD $\hat{G}_{\mathcal{S}}$ and its lexicographic index $\imath_{s'}$.

It remains to compute the actual matrix entry $\mathbf{M}(s, s') = f_t(s)$, i.e. the value which is assigned to the state transition and defined by the possibly state-dependent function f_t of the Petri net transition. At this point I do not require

Algorithm 17 (Enumerate – State transitions)

```
1  proc AuxEnumerateTransitions(v, v̂_src, v̂_dest : unsigned, i_src , i_dest : IndexT,
2     al : ActionList , f : Function , args: FunctionArguments, op : OP)
3
4     if op.firstUse(v, v̂_src, v̂_dest, i_src, i_dest,al,f,args) then
5        return
6     fi
7     for 1 ≤ j ≤ |p(v̂)| do
8        for inf(I_j) ≤ s_src < sup(I_j) do
9           al' := al
10          if al ≠ ⊥ then
11             a := head(al)
12             if var(v) = a.var then
13                if s_src ∉ a.enabled then continue fi
14                 s_dest := s_src + a.shift
15                op.setArgument(var(v̂),args, s_src,s_dest)
16                al' := tail(al)
17             fi
18          fi
19          v' := c_j(v)
20          i'_src := i_src+r(v̂_src,s_src)
21          v̂'_src :=c(v̂_src,s_src)
22          if not isValid(v̂'_src) then continue fi
23          i'_dest := i_dest+r(v̂_dest,s_dest)
24          v̂'_dest :=c(v̂_dest,s_dest)
25          if isValid(v̂'_dest) then
26             AuxEnumerateTransitions(v',v̂'_src,v̂'_dest ,i'_src,i'_dest, al',f,args,op)
27          fi
28       od
29    od
30    op.secondUse(v, v̂_src, v̂_dest, i_src, i_dest,al, f,args)
31 end
32 proc EnumerateTransitions (G_S : IDD , t : transition, op : OP)
33    args : f_t.createArgs()
34    al := op.selectActionList(al_t,revAl_t);
35    AuxEnumerateTransitions(G_S.root,Ĝ_S.root,Ĝ_S.root,0, 0,al,F(t),args,op)
36 end
```

a special semantics of f_t (and thus $\mathbf{M}$). It may define a rate of a timed transition, the weight of an immediate transition or a secondary probability as $\frac{\mathbf{R}(s,s')}{E(s)}$ ($\mathbf{P}$) or $\frac{\mathbf{R}(s,s')}{\lambda}$ ($\mathbf{P}^{\mathcal{U}}$). In any case, the procedure must collect the function arguments specified by the individual sub-states. The interface for *Functions* and *FunctionArguments* was already specified in Algorithm 5.

The procedure *EnumerateTransitions* is parameterized with the type of operation (OP) which is finally applied to each extracted state transition. The actual work is done by the parametrized auxiliary function *AuxEnumerateTransitions*.

The special feature of *AuxEnumerateTransitions* is in particular the processing of the variable of the visited IDD node with respect to the firing of the transition t. If the variable represents a place from $t's$ environment (line 12), it checks the enabledness of t in the current sub-state (line 13). If t is enabled, the value of the variable of the resulting state s_{dest} is computed (line 14) and used to determine the child node in the LIDD $\hat{G}_{\mathcal{S}}$ (line 24). The rest should be self-explanatory. For reasons of performance I restrict the set of permitted function variables to the environment of the corresponding Peri net transition (line 15).

I assume that the state set S is a subset of $\mathcal{S}$. However, when firing the transition backwards, the algorithm may explore states which are not in $\mathcal{S}$. This situation is treated in line 25, where in the case of an invalid assignment the function $c(\hat{v}_{dest}, s_{dest})$ returns the 0-terminal node.

The given procedure can be used to enumerate the entries of the matrix $\mathbf{M}$ or its transpose $\mathbf{M}^T$. The latter can be easily achieved by firing all transitions in backward direction. The operation type OP specifies further the traversal policy and must take care of the following aspects:

- A first evaluation of the node v. Thus the procedure *firstUse* must be defined. If v is for instance the 1-terminal node, a state transition has been extracted and the actual operation is performed. Its return value tells *AuxEnumerateTransitions* whether to continue the traversal with the evaluation of the outgoing arcs. The procedure has also to define the basic operation which is applied to an extracted state transition.
- A second evaluation of the node v after traversing the outgoing arcs. The procedure *secondUse* must be defined if it is required to perform actions after the recursive traversal. The functions *firstUse* and *secondUse* are especially important for the optimization I will discuss in the remainder.
- The *fire direction* of the transition which is represented by the used action list. This is done by defining the function *selectActionList*.

- The selection of the function arguments by the procedure *setArgument*. If the transition fires backwards, the argument depends on the successor state s'.

At this point it should be mentioned that the formulation of the algorithms is meant to clearly illustrate the underlying ideas. The actual implementation is of course much more optimized.

Example 12

Let us consider again the LIDD in Figure 4.2, the selected states s_1, s_2 and s_3 in Example 11 and two Petri net transitions in Figure 3.3.

When applying Algorithm 17 with the reachable states $G_{\mathcal{S}}$ and the transition *produce_choose_b2* the fourth extracted path is ρ_{s_1} as its lexicographic index $\imath_{s_1}$ is 3. In node v_9, node v_{11} and node v_{15} the algorithm emulates the firing of the Petri net transition. It "removes" a token from the place *ready* and "puts" a token on place *item* and on place *to*2. The path of the target state s'_1 is in this case

$$v_{15} \xrightarrow{1} v_{14} \xrightarrow{0} v_{12} \xrightarrow{1} v_{10} \xrightarrow{0} v_8 \xrightarrow{0} v_6 \xrightarrow{2} v_5 \xrightarrow{0} v_2 \xrightarrow{1} 1$$

which we already know as ρ_{s_2}. It also computes the associated value as

$$0.1 \cdot 1/(1 + s_1(b1) + s_1(b2)) = 1/(1 + 0 + 2) = 0.0\overline{3}.$$

This gives the rate matrix entry $\mathbf{R}(s_1, s_2) = \mathbf{R}(3, 23) = 0.0\overline{3}$.

The application of the algorithm for transition *consume_fetch_b2_1* gives, among others, the entry $\mathbf{R}(s_1, s_3) = \mathbf{R}(3, 2) = 1$. In total, Algorithm 17 enumerates 73 state transitions for the reachable state $\mathcal{S}$ and all Petri transitions .

Observation. A transition which changes the marking of the top-place induces state transitions with a large index jump concerning the row and the column index. A transition which only changes the marking of places in lower levels causes small index jumps.

★

Algorithm 17 permits to realize the operations for the numerical analysis of SPN. I will discuss their implementation in Chapter 5. For now I illustrate its use to print the rate matrix $\mathbf{R}$ of a CTMC, given as SPN N_S and the LIDD

representation $\hat{G}_{\mathcal{S}}$ of its reachable state $\mathcal{S}$. We use the procedure *PrintRates* and the related functor given in Algorithm 18. The entries of the matrix will not been printed line-by-line. The algorithm prints the state transitions induced by the Petri net transition t_i before those of the transitions $t_j \in T : j > i$. A line-by-line enumeration is only observable when applying the algorithm for a single transition. Numerical solvers as Gauss-Seidel which require to extract the matrix entries line-wise can not be realized with such an enumeration policy.

Algorithm 18 (Print the rates)

```
1  struct PrintRates
2     func firstUse(v, v̂_src, v̂_dest : unsigned, i_src , i_dest : IndexT,
3        al : ActionList , f : Function , args: FunctionArgument)
4        if v = 1 then
5           print(i_src+ ','+i_desr+' :' +f(args))
6           return true
7        fi
8        if v = 0 then return true fi
9        //otherwise the traversal can not be stopped
10       return false
11    end
12
13    proc secondUse(v, v̂_src, v̂_dest : unsigned, i_src , i_dest : IndexT,
14       al : ActionList , f : Function , args: FunctionArgument)
15       //nothing to do
16    end
17    proc setArgument(var : unsigned, args: FunctionArguments,
18       src : unsigned, dest : unsigned)
19       args.setArgument(var, dest);
20    end
21    func selectActionList(al : ActionList, revAl : ActionList)
22       return al
23    end
24 end
25 proc PrintRates(G_S : IDD, CTMC : [Ĝ_S,N_S])
26    pr : struct PrintRates
27    for t ∈ N_S.T do
28       EnumerateTransitions(G_S,t,pr)
29    od
30 end
```

4.3.3 Performance Tuning

IDDs provide a very compact representation of a set of states S and Algorithm 17 enables the enumeration of all state transitions caused by the firing of a Petri net transition t originating in the states *enabled(S,t)*. Consequently we have a very memory efficient representation technique for the rate or generator matrix of even huge CTMCs. However, the extraction of the single states performed in the procedure *EnumerateTransitions* has the following disadvantages:

1. The plots in Figure 4.14 and Figure 4.15 reveal the same problems reported by Parker in [96]. Many paths from inner IDD nodes to the terminal nodes have to be traversed several times, and caching techniques which usually mitigate the problem of redundant computations are not applicable here.
2. So far Algorithm 17 must be applied separately for all transitions of the given Petri net.

These drawbacks cause an unacceptable runtime and prohibit special enumeration strategies. I will now present solutions for these two problems, in particular

1. an adaption of Parker's approach to reduce the depth of recursion, and
2. a generalization of Algorithm 17 which considers arbitrary sets of Petri net transitions and enables to define different policies for the enumeration of the state transitions.

Early truncation. A way to reduce the effort introduced by redundant computations is to store explicitly the information of potential path extensions in selected nodes of the decision diagram. This enables to truncate the recursion and to use the available information directly when reaching such nodes. This idea was proposed for MTBDDs by Parker [96] and later adapted for MTZDDs in [67]. I will now discuss a modification of this technique for the on-the-fly computation of state transitions.

Given a variable $x_j \in X = \{x_1, \ldots, x_n\}$ and a state transition $s \xrightarrow{f_t(s)} s'$, we refine the path ρ_s to $\rho_{x<j} \xrightarrow{s_{j-1}} \hat{v}_j \underbrace{\xrightarrow{s_j} \rho_{x>j}}_{\rho_{x\geq j}}$ and $\rho_{s'}$ to $\rho_{x'<j} \xrightarrow{s'_{j-1}} \hat{v}'_j \underbrace{\xrightarrow{s'_j} \rho_{x'>j}}_{\rho_{x'\geq j}}$ such that $var(\hat{v}_j) = var(\hat{v}'_j) = x_j$ (see Figure 4.5). For the sub-paths $\rho_{x\geq j}$ and $\rho_{x'\geq j}$ we compute the indices $\imath_{\rho_{x\geq j}}$ and $\imath_{\rho_{x'\geq j}}$ relative to the nodes $\hat{v}_j$ and $\hat{v}'_j$ as

$$\imath_{\rho_{x\geq j}} = \sum_{j\leq l\leq n} r(\hat{v}_l, s_l) \text{ and } \imath_{\rho_{x'\geq j}} = \sum_{j\leq l\leq n} r(\hat{v}'_l, s'_l).$$

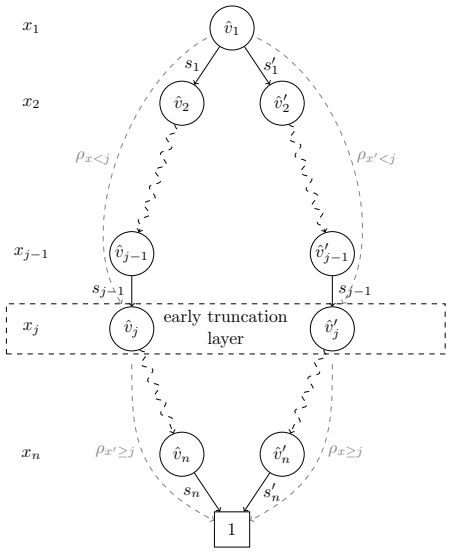


Figure 4.5: A refinement of LIDD paths.

We further bind the variables of the function $f_t(x_1, \ldots, x_n) \to \mathbb{R}_{>0}$ to the values which are given by the sub-path $\rho_{x\geq j}$.
The result is the function $f_{t,\rho_{x\geq j}}(x_1, \ldots, x_{j-1}) \to \mathbb{R}_{\geq 0}$[9]. The state transition is now specified as $(\rho_{x<j}, \rho_{x\geq j}) \xrightarrow{f_{t,\rho_{x\geq j}}(\rho_{x<j})} ((\rho_{x'<j}, \rho_{x'\geq j}))$ and identifies the matrix entry

$$\mathbf{M}(\imath(\rho_{x<j}) + \imath(\rho_{x\geq j}), \imath(\rho_{x'<j}) + \imath(\rho_{x'\geq j})) = f_{t,\rho_{x\geq j}}(\rho_{x<j}).$$

The idea is to associate the triple $(\imath(\rho_{x\geq j}), \imath(\rho_{x'\geq j}), f_{t,\rho_{x\geq j}})$ to all nodes $\hat{v}_j$ in the LIDD $\hat{G}_{\mathcal{S}}$ with $var(\hat{v}_j) = x_j$ and for each state transition characterized above. As the index of the target states depends on the Petri net transition t and the related IDD node $\hat{v}'_j$, the node $\hat{v}_j$ gets associated a data structure, which maps the triple $(\imath(\rho_{x\geq j}), \imath(\rho_{x'\geq j}), f_{t,\rho_{x\geq j}})$ as one of multiple values to the tuple $(t, v_j, \hat{v}'_j)$. The node v_j is the node in the traversed IDD G_S.

If all these information have been pre-computed, it suffices to stop the traversal in Algorithm 17 and to enumerate the entries belonging to t, v_j and $\hat{v}_{dest}$ whenever $var(v)$ equals x_j (see Algorithm 19). Yes, the alert reader will notice that we are implicitly dealing with a Multi-terminal Interval Decision Diagram whose terminal nodes store lists of multi-maps - for each "terminal" node one multi-map for each Petri net transition. Nevertheless, the proposed method still follows

[9] Such argument binding is called a partial function application.

Algorithm 19 (Enumerate – Entries)

```
1  struct EnumerateEntries
2     x_j : unsigned
3     func firstUse(v, v̂_src, v̂_dest : unsigned, i_src , i_dest : IndexT,
4        al_t : ActionList , f : Function , args: FunctionArgument)
5
6        if var(v)! = x_j then return true fi
7        forall (ι(ρ_{x≥j}), ι(ρ_{x'≥j}), f_{t,ρ_{x≥j}}) ∈ map(v̂_src)[(t, v, v̂_dest)] do
8           // do something with (i_src + ι(ρ_{x≥j}), i_dest + ι(ρ_{x'≥j}), f_{t,ρ_{x≥j}}(args))
9        od
10       return false
11    end
12 end
13 ...
14 proc Enumerate(G_S : IDD, x_j : unsigned, ctmc : [Ĝ_S,N_S])
15    ece : EnumerateEntries
16    ece.x_j := x_j
17    for t ∈ N_S.T do
18       EnumerateTransitions(G_S,t,ece)
19    od
20 end
```

an on-the-fly approach. The state s', especially its lexicographic index, and the value $f_t(s)$ are computed for each state transition $s \xrightarrow{f_t(s)} s'$ by firing the Petri net transition t in state s.

Data structures and initialization. An early truncation of the IDD traversal requires at first the pre-computation of the explicitly stored path extensions for all nodes in the LIDD $\hat{G}_{\mathcal{S}}$ labeled with the variable x_j and all Petri net transitions. This is realized with the procedure InitExplicitRep in Algorithm 20 which applies the Algorithm 17 with the related functor. When reaching a node labeled with variable x_j, the algorithms checks first whether there are available entries for the Petri net transition t and the LIDD node $\hat{v}_{dest}$. If so, the recursive descent can be stopped here. Otherwise it resets the content of what I call an *IndexDataCollector* and continues the traversal in direction of the terminal nodes. Each time the traversal reaches the 1-terminal node, the computed state transition is passed to the *IndexDataCollector*, which maps the index pair $(\imath(\rho_{x\geq j}), \imath(\rho_{x'\geq j}))$ to the related function $f_{t,\rho_{x\geq j}}$ (line 12).

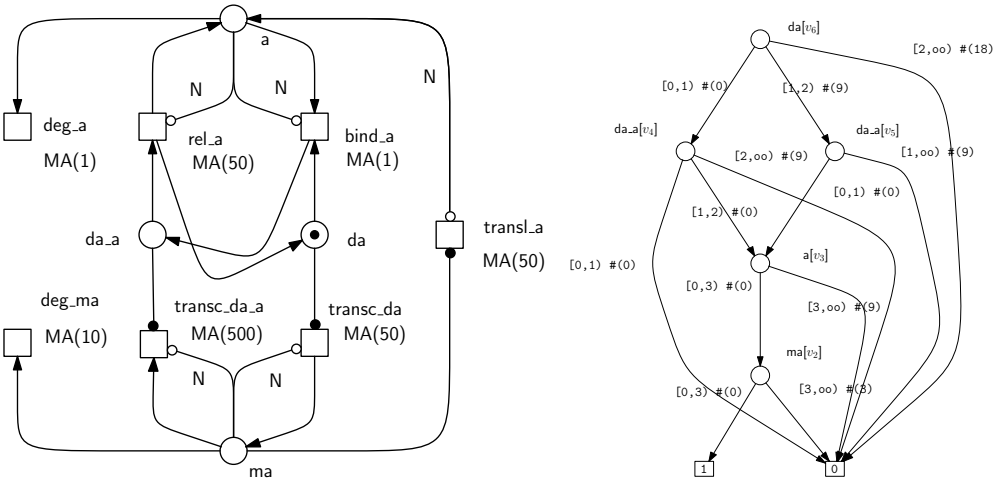


Figure 4.6: A Stochastic Petri net and the LIDD encoding of its rechable states for $N = 2$.

I want to use the sub-net of the *Circadian Clock* model (see Appendix A.2, CLOCK) given in Figure 4.6a to illustrate the following considerations. The LIDD in Figure 4.6b represents the reachable states for the constant value $N = 2$

Algorithm 20 (Initialization – Path extensions)

```
1  struct InitExplicitRep
2     op : OP //the type of Operation
3     idc : IndexDataCollector
4     x_j : unsigned
5
6     func firstUse(v, v̂_src, v̂_dest : unsigned, i_src , i_dest : IndexT,
7        al : ActionList , Function , args: FunctionArguments)
8        if v = 0 then
9           return true
10       elseif v = 1 then
11          //state transition has been extracted
12          idc.insert(f(args), i_src , i_dest)
13          return true
14       elseif var(v) = x_j then
15          if map(v̂_src)[(t, v, v̂_dest)] ≠ ∅ then return false fi
16          idc.reset()
17          return true
18       fi
19       return false
20    end
21
22    proc secondUse(v, v̂_src, v̂_dest : unsigned, i_src , i_dest : IndexT,
23       al : ActionList , f : Function , args: FunctionArguments)
24       map(v̂_src)[(t, v, v̂_dest)] := idc.content()
25    end
26
27    proc setArgument(var : unsigned, args: FunctionArguments ,
28       src : unsigned, dest : unsigned)
29       if var(v) ≥ x_j then
30          op.setArgument(var(v),args,src,dest)
31       fi
32    end
33
34    proc selectActionList(al : actionList, revAl : actionList)
35       return op.selectActionList(al , revAl);
36    end
37 end
38
39 proc InitExplicitRep(G_S : IDD, op : OP, x_j : unsigned)
40    ic : struct InitExplicitRep
41    ic.op := op
42    ic.x_j := x_j
43    EnumerateTransitions(G_S,ic)
44 end
```

and the variable order $\pi = ma < a < da\_a < da$. The abbreviation MA which is used for the specification of the rate function stands for the mass-action law and is defined for a transition t in state s with a reaction-specific constant c_t as

$$MA(c_t) = c_t \cdot \prod_{p \in {}^\bullet t} \binom{s(p)}{t^-(p)}.$$

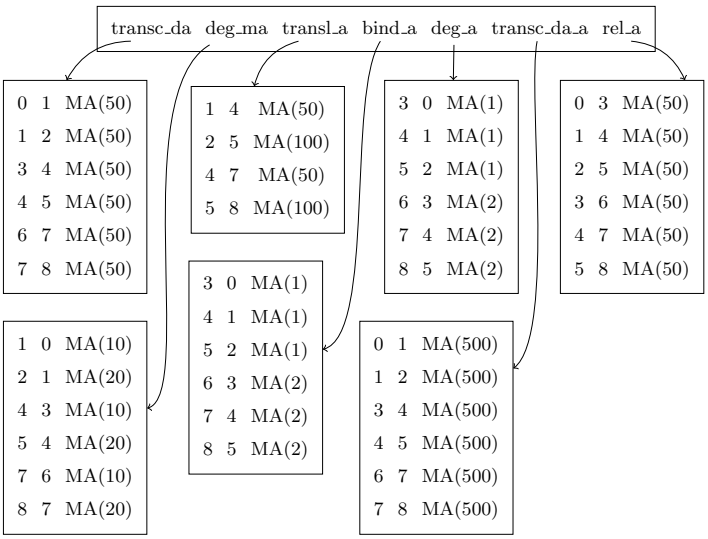


Figure 4.7: A naive scheme for the node v_3 in Figure. 4.6b

A naive realization of the described concept could simply consider lists of triples $(\imath(\rho_{x \geq j}), \imath(\rho_{x' \geq j}), f_{t,\rho_{x \geq j}})$ as it is illustrated in Figure 4.7, but a closer look reveals potential for improvements:

1. Many list entries share the same function. There are 40 list entries in total, but only seven different functions. Although the rate matrix of a CTMC may contain an huge amount of non zero entries, the number of distinct values is usually quite small[10]. At this point it makes sense to group the index entries by the associated functions, as it is illustrated in Figure 4.8.
2. After the reorganization it stands out that complete index lists may appear several times. Further, there are often references to the same function. It seems worth detecting occurrences of index lists and function instances and to store for the Petri net transitions just the related pointers as it can be seen in Figure 4.9.

[10] A small number of distinct values is a requirement for an efficient application of MTDD-based representation techniques.

3. A further observation is that a couple of the index lists exhibit some kind of regularity. This enables to store them in a very compact way.

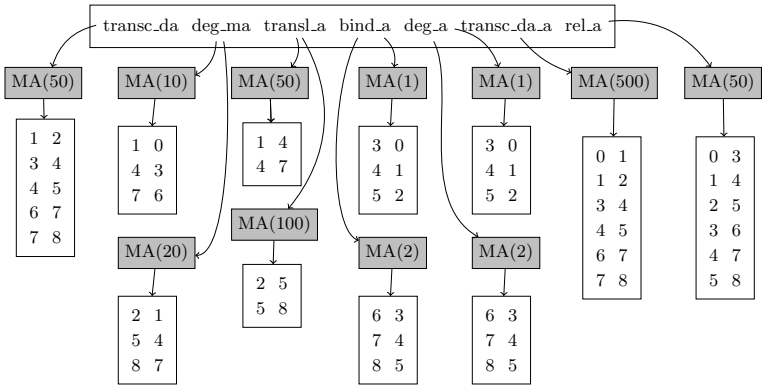


Figure 4.8: An improved scheme for the node v_3 in Figure. 4.6b

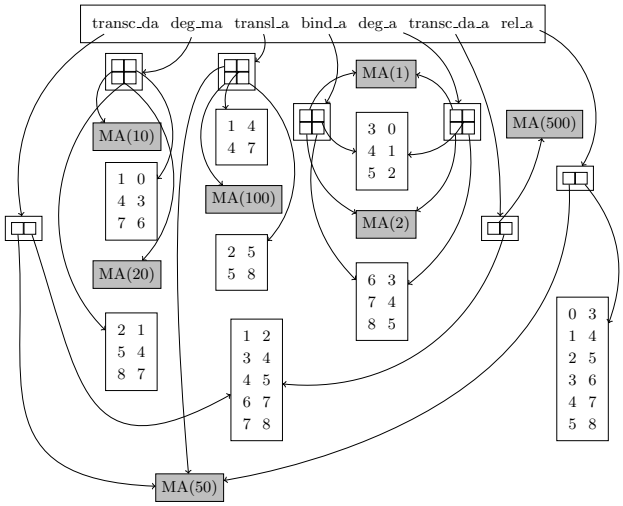


Figure 4.9: An improved scheme for the node v_3 in Figure 4.6b which takes into account the multiple occurrence of similar functions and index lists.

Exploitation of regularities. As one may expect, the success of the overall approach directly depends on the storage and the access to the functions and especially to the possibly huge amount of indices.

In the following I assume that a set of index pairs is represented by the structure *IndexPairs*:

```
1  struct IndexPairs
2      indices : array of IndexT
3      size : unsigned
4
5      proc process(offR, offC : IndexT , val : double, op : OP)
6          for 0 ≤ i < size do
7              op(row(i) + offR, col(i) + offC, val)
8          od
9      end
10 end
```

The procedure *process* iterates over all index pairs stored somehow in the array *indices* using the functions row and col, adds the offsets *offR* and *offC*, and calls the functor *op* with the resulting index pair and the value *val*. Once again, *op* encapsulates the actual operation and the related data.

The set of index pairs which is associated to a function instance often exhibits regularities in the set of row indices, the column indices or even both. My experiments revealed that at least the following patterns (types) should be considered, which I will characterize in more detail:

Type 1: No regularities. All row and column indices must be stored in the array, either absolute or relative with the distance of rows and columns to the related predecessor pair. I have chosen the relative encoding and store the first index pair at the positions zero and one. For the following pairs I store the distance to their predecessor pair in alternating order of rows and columns. This is motivated by the fact that the row and the column index of a pair will always be accessed simultaneously and should be located in the same cache line[11]. The *ith* index pair is defined as

$$row(i) = \begin{cases} indices[0] & \text{if } i = 0 \\ row(i-1) + indices[2 \cdot i] & \text{if } 1 \le i < size. \end{cases}$$

and

$$col(i) = \begin{cases} indices[1] & \text{if } i = 0 \\ col(i-1) + indices[2 \cdot i + 1] & \text{if } 1 \le i < size. \end{cases}$$

This definition directly represents how the algorithms access the entries of a set of index pairs. An index pair is always computed from its direct

[11] A cache line represents a consecutive block of Random Access Memory (RAM) which is completely transferred to the substantially faster cache memory, even if only a single datum is needed.

predecessor. In the implementation this suggests to make intensive use of an Iterator concept instead of an index-controlled **For** loop from 0 to $size$.

Type 2: All row and all column indices have a fixed distance. Of course the distance between the rows can differ from the one between the columns. The complete list contents can be mapped to only four values, the first two indices and the distance between the rows and the columns. The ith index pair is defined as

$$row(i) = \begin{cases} indices[0] & \text{if } i = 0 \\ row(i-1) + indices[2] & \text{if } 1 \leq i < size \end{cases}$$

and

$$col(i) = \begin{cases} indices[1] & \text{if } i = 0 \\ col(i-1) + indices[3] & \text{if } 1 \leq i < size. \end{cases}$$

The ith row (column) is nothing else than $indices[0(1)]+i \cdot indices[2(3)]$.

Type 3: The distance of all entries is one, which is special case of the previous type. During the numerical computations such a set of index pairs represents successive vector blocks. Such blocks enable the application of *Single Instruction Multiple Data* (SIMD) techniques.
The ith index pair is defined as

$$row(i) = \begin{cases} indices[0] & \text{if } i = 0 \\ row(i-1) + 1 & \text{if } 1 \leq i < size \end{cases}$$

and

$$col(i) = \begin{cases} indices[1] & \text{if } i = 0 \\ col(i-1) + 1 & \text{if } 1 \leq i < size. \end{cases}$$

The ith row(column) is nothing else than $indices[0(1)] + i$.

Type 4: The row indices possess fixed distance, while the column indices do not show regularity. Only the distance between the column indices of two pairs must be stored starting at array position three. Position two keeps the constant distance of the row indices.
In this case the ith index pair is defined as

$$row(i) = \begin{cases} indices[0] & \text{if } i = 0 \\ row(i-1) + indices[2] & \text{if } 1 \leq i < size \end{cases}$$

and

$$col(i) = \begin{cases} indices[1] & \text{if } i = 0 \\ col(i-1) + indices[2+i] & \text{if } 1 \leq i < size. \end{cases}$$

Type 5: Analogously to the previous type only the column indices can be characterized by a constant distance. Here the row indices require explicit storage. In this case the *ith* index pair is defined as

$$row(i) = \begin{cases} indices[0] & \text{if } i = 0 \\ row(i-1) + indices[2+i] & \text{if } 1 \leq i < size \end{cases}$$

and

$$col(i) = \begin{cases} indices[1] & \text{if } i = 0 \\ col(i-1) + indices[2] & \text{if } 1 \leq i < size. \end{cases}$$

Type 6: The last case is on hand, when the distance between row and column indices changes from pair to pair but the distance between two row indices is equal to the distance of the related column indices. The start indices are stored at the positions zero and one. The shared distances between two pairs are stored starting at position two.
The *ith* index pair is defined as

$$row(i) = \begin{cases} indices[0] & \text{if } i = 0 \\ row(i-1) + indices[1+i] & \text{if } 1 \leq i < size. \end{cases}$$

and

$$col(i) = \begin{cases} indices[1] & \text{if } i = 0 \\ col(i-1) + indices[1+i] & \text{if } 1 \leq i < size. \end{cases}$$

The investigation of these pattern saves memory and enables a fast computation of index pairs. The computation of the index pairs for the *Type1* requires access to *RAM* for each pair. This is slow and may invalidate the fast cache memory. For *Type2* and *Type3* it is possible to compute a huge number of pairs from only five values, in particular the number of pairs, the indices of the first pair and distances between neighboring pairs. The use of the remaining types saves nearly half of the required memory compared to the first one.

Figure 4.10 shows the previous scheme after the exploitation of the regularities in the index lists.

A single list of index pairs may be composed of several regular blocks. This calls for a detection mechanism which can be compared with a lexical analyzer,

breaking a stream of characters into tokens. The *IndexDataCollector* serves as such an analyzer.

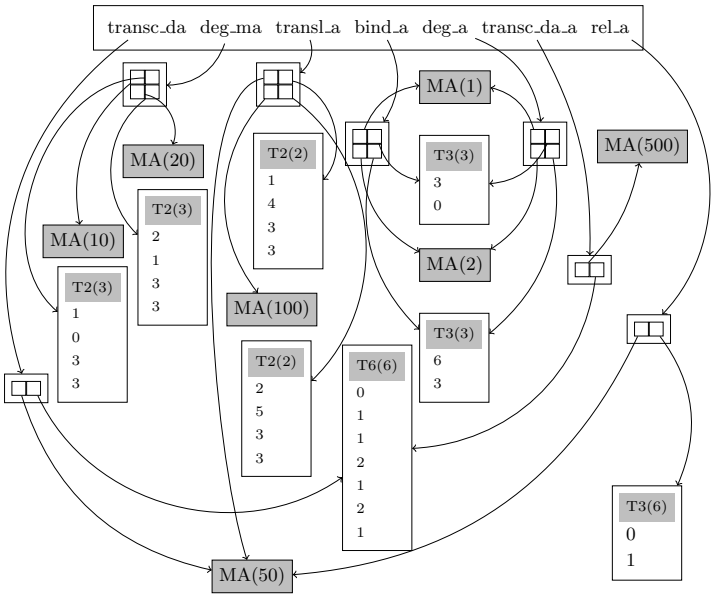


Figure 4.10: A scheme for the node v_3 in Figure 4.6b which considers the multiple occurrence of similar functions and regularities in the index lists.

When appending a new index pair, it observes the distance between neighboring indices and identifies the introduced types of regularities. In the best case, as for instance in Figure 4.10, each index list can be represented by one of the mentioned types, except *Type1*, which indeed represents the *worst case*. Each time the *IndexDataCollector* observes the end of a regular block, it decides whether it is worth creating an instance of the detected type or not as there is trade-off between a huge number of instances representing very small blocks and a small number of instances representing big blocks. Therefore the *IndexDataCollector* considers a minimal block size. It turned out for the experiments I made that a minimal block size of 100 appears to be a suitable value.

I omit a discussion of the internal implementation of the *IndexDataCollector*, but it is obvious that it can be realized as a finite automaton.

Further data structures. The identified blocks of *IndexPairs* must be associated with a function instance. An instance of the structure *MatrixEntries* serves for this purpose. It defines a set of matrix entries with the same value. The structure

defines, similar to $IndexPairs$, the parametrized procedure *process* which indeed triggers the enumeration of the index pairs by calling the related procedure.

```
1  struct MatrixEntries
2      indices : IndexPairs
3      f : Function
4
5      proc process(offR, offC : IndexT , args : FunctionArguments, op : OP)
6          val := f(args)
7          indices.process(offR, offC, val, op)
8      end
9  end
```

As the functions are potentially state-dependent, a Petri net transition t may define for the same LIDD node $\hat{v}_j$ various combinations of different functions and blocks of index pairs. Thus, each node $\hat{v}_{src}$ labeled with variable x_j refers to a map which associates a list of *MatrixEntries* instances to the key $(t, v, \hat{v}_{dest})$.

Generalization. At the beginning of this section I mentioned a second problem with the on-the-fly enumeration of state transitions. Algorithm 17 extracts for a given set of states S, encoded by the IDD G_S, the state transitions induced by the firing of a single Petri net transition t. An iterative application of the algorithm with all Petri net transitions and the set of reachable states $\mathcal{S}$ enumerates the complete transition relation of the given model, as it is done in Algorithm 18. However, doing so processes the Petri net transitions one after the other. Because the same IDD (G_S) will be traversed $|T|$ times, we can expect a long runtime. Further, the approach would not serve our needs if the numerical algorithm, as for instance Gauss-Seidel, requires a certain order in which the state transitions have to be enumerated.

Thus it is necessary to generalize the Algorithm 17. Instead of a single Petri net transition, the auxiliary function AuxEnumerateTransitions considers simultaneously a set of Petri net transitions. The consequence: all procedures and all related functors I have introduced so far have to deal with sets of *Petri net transitions*, *IDD nodes*, *indices*, *action lists*, and *function arguments*. The generalization of Algorithm 17 is straightforward and for reasons of readability I omit it here. At this point I define just the resulting interface which I refer to in the remainder.

The structure *TransitionData* encapsulates for a transition t

Algorithm 21 (Enumerate state transitions – Generalized)

```
1  proc EnumerateTransitions (G_S : IDD , T : set of transitions , op : OP)
2     ...
3  end
4  proc AuxEnumerateTransitions (v : unsigned, td : set of TransitionData, op : OP )
5     ...
6  end
7  //functions implemented by the specific policies
8  func firstUse(v : unsigned, td : set of TransitionData)
9     ...
10 end
11 func secondUse(v : unsigned, td : set of TransitionData)
12    ...
13 end
```

- the transition – t
- the current lexicographic index – $index$
- the current node in the LIDD – $\hat{v}$
- the current ActionList – al
- the associated function – f
- the list of function arguments – $args$
- a lower step bound – lsb
- an upper step bound – usb
- and a Boolean flag – $shift$.

The latter three features permit an on-the-fly generation of the approximating CTMC $C^{\mathcal{A}}$ discussed in Section 3.2.2. I will explain them later in Section 5.2.3.

The argument for the related procedures *TraverseTransitions, AuxTraverseTransitions, firstUse* and *secondUse* is a list of *TransitionData* instances, representing the transitions which are enabled in terms of the extracted sub-path.

Algorithm 19 realizes the enumeration of possible path extensions in the function firstUse of the structure *EnumerateEntries*. It must also specify the base operation which is applied to a state transition.

In the following I will distinguish between the actual base operation (e.g. print a matrix entry) and the enumeration of the path extensions. This enables the combination of different **enumeration policies** with different operations. Therefore I extract the enumeration of the path extensions to parameterizable functor types. Each type realizes a different policy for the enumeration of the path extensions and encapsulates the actual operation by a reference to a dedicated functor named *base*.

Enumeration policies. The implementation of the numerical solvers to cover the analysis of SPN,GSPN and SRN models requires basically three different enumeration policies. I assume that $\mathcal{T}$ in Algorithm 21 represents T, and G_S the set of reachable states. With this restriction the algorithm enumerates all state transitions of a row block, when it reaches a node of the truncation level j. The algorithm enumerates the blocks sequentially. Within a block there are jumps between rows and columns, depending on the processed index pair pattern and the considered Petri net transition. The size of a block (the number of rows) is defined by the maximal row index, which is contained in the set of the stored path extensions. The number and the size of the blocks depends on the number of nodes labeled with the variable x_j and thus on the selected layer j.

If the chosen layer is $|X|$, the enumeration algorithm extracts single matrix rows. Of course, in this case a row-wise enumeration has a non-acceptable runtime. An **efficient** row-wise enumeration of the matrix entries must be realized in the function *firstUse* of the according policy.

Next to the base operation it may be necessary to perform a further operation to prepare computed or collected results. For instance, after having printed all entries in a row we may want to start a new line. In most cases the operation deals with gathering of the recently computed result, so it is represented by a functor **gather**. For the realization of a numerical solver based on uniformization (see Algorithm 8), the related functor type encapsulates for instance the vector $\underline{acc}$ and takes care of the accumulation of the results (line *9*). In this case the *gather* operation is carried out after the multiplication of the matrix $\mathbf{P}^{\mathcal{U}}$ and does not affect the enumeration of the path extensions. However, in Algorithm 10 the update of a vector entry (line *14*) must be done immediately after the extraction of the related row. Thus I distinguish immediate and subsequent gathering.

Enumeration of blocks. Algorithm 22 enumerates the entries of a complete block of the truncation level. The function *firstUse* checks whether the current IDD

Algorithm 22 (Enumeration policy – BLOCK/ALL)

```
1  struct EnumerateBLOCK/ALL
2
3     base : BASE//base operation
4     gather : GATHER//gather policy
5     x_j : unsigned
6
7     func firstUse(v : unsigned, td : list of TransitionData)
8        if var(v) ≠ x_j then return true fi
9        v̂_src := td_1.v̂
10       i_src := td_1.index
11       for 2 ≤ i ≤ |td| do
12          entries := map(v̂_src)[(td_i.t, v, td_i.v̂)]
13          if entries ≠ ∅ then
14             forall e_j ∈ entries do
15                rowIndex := i_src
16                colIndex := td_i.index
17                value := e_j.f(td_i.args)
18                e_j.process(0, rowIndex, colIndex, value, base)
19             od
20          fi
21       od
22       gather(i_src, i_src + maxRow(entries))
23       return false
24    end
25
26 end
```

node v is labeled with the variable x_j. If so, it processes the existing entries. The LIDD node $\hat{v}_{src}$ and the lexicographic index of the related sub-path are provided by the first entry of the *TransitionData* list *td*. The LIDD nodes of the sub-path of target states and their related lexicographic index in case of enabled transitions start with the second entry of *td*. For existing entries it calls the procedure *process* and delegates the basic operation by passing the functor *base*. After that it finishes with a call of the functor *gather* providing the index interval given by the index $\imath_{src}$ and the block size.

If the operation does not carry out immediate gathering, the algorithm applies the base operation to **all** matrix entries before the actual gathering step; and I refer to this special case as *ALL*. The return value tells the calling procedure *AuxEnumerateTransitions* whether to continue (true) or to truncate (false) the recursive descent.

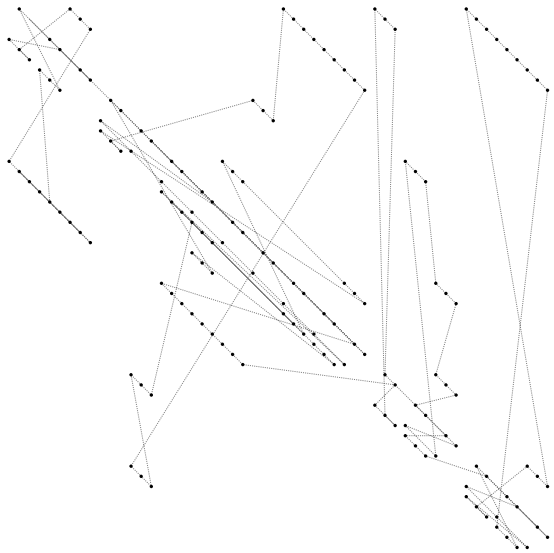


Figure 4.11: Visualization of an enumeration of the transposed rate matrix of a CTMC.

Enumeration of rows/columns. Algorithm 23 enumerates the entries of a block of the truncation layer line by line. This is achieved by using a list of *Iterator* instances. An iterator enables for a *MatrixEntries* instance the stepwise enumeration of its contents. The idea is quite simple. The list stores at position i all iterators which refer a *MatrixEntries* instance whose current row index is i. The iterator operation *next* computes the successor index pair depending on

the encapsulated pattern of index pairs. The fist part equals that of Algorithm 22, but does not process the complete content of a *MatrixEntries* instance. It creates an iterator (line *19*) and evaluates its contents. If the current row index is zero, it applies the base operation and sets the iterator to the next index pair. In any case, the iterator, provided that it is still valid, is inserted in the *list* at its row position. After this first phase (line *14* - line *27*), the iterator list has been initialized and all index pairs with a row index of zero have been processed.

In the second phase, the algorithm enumerates the single row indices starting at position one. For each position it goes through the associated iterators, processes the referred matrix entry by applying the *base* operation and sets the iterators to the next pair, including the movement of valid iterators to the correct list position. Having processed the current row position, the algorithm applies the immediate *gather*-operation restricted to the current row index (line *28* and line *37*).

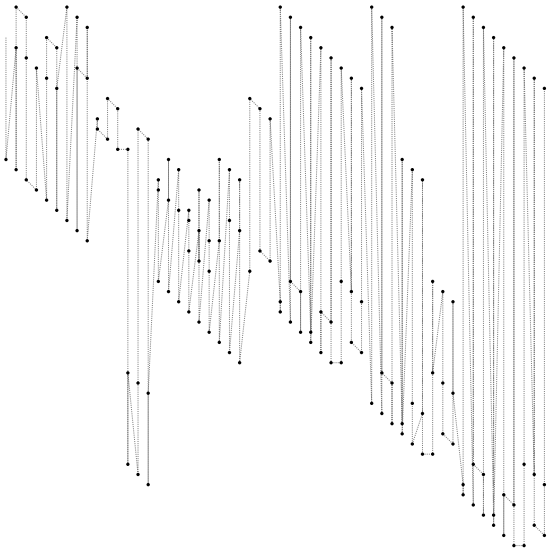


Figure 4.12: Visualization of a column-wise enumeration of the rate matrix of a CTMC.

Multiple enumeration. An enumeration policy which may not appear reasonable at the first glance is given in Algorithm 24. Again, it is very similar to the enumeration of blocks, but the algorithm considers also the *shift*, the *lsb* and the *usb* members of the *TransitionData* instances. The complete set of matrix entries is processed for each step within the bounds *lsb* and *usb* (line *17* and

line 20). The actual index pairs are further shifted by the number of reachable states of the model (line 24 and line 25). A transition for which the *shift* flag is set to *true* shifts the column index one times further and connects thereby two neighboring repeating levels (line 19).

The described enumeration of the repeating levels can be used to compute on-the-fly the rate matrix of the CTMC $C^{\mathcal{A}}$ discussed in Section 3.2.2. To omit a repeated computation of the actual values, they are computed once and stored in the array *values*.

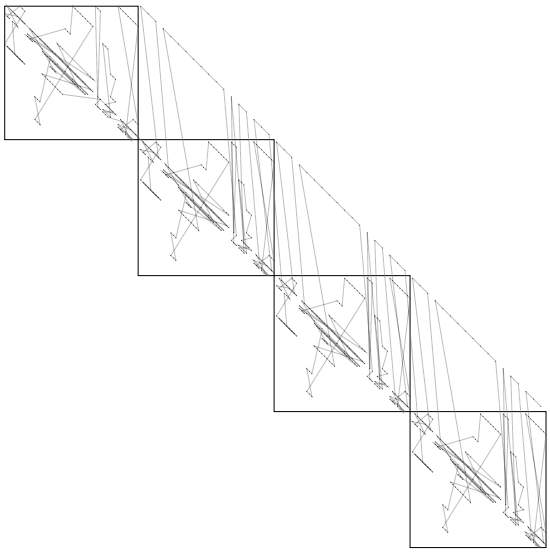


Figure 4.13: Visualization of an enumeration of the rate matrix of a CTMC which approximates an MRM. The four repeating levels are enumerated by means of the enumeration policy MULTI.

Variable order and functions. I mentioned in Section 2.2 that the variable order has crucial impact on the efficiency of DD-based approaches. A good variable order yields small-sized decision diagrams in terms of the number of nodes and arcs. Several research in this area has been done, whereby most approaches use heuristics to generate a static variable order [95, 111, 31, 83].

The IDD library [115] my work is based on implements the greedy algorithm proposed by Noack [95]. The algorithm works from the bottom ($|X|$) to the top and associates in each step the current variable to a place of the net. To decide

which of the unprocessed places, represented by the set U, will be selected next, the algorithm uses a simple heuristic which considers explicitly the Petri net structure. It is based on the observation that places (variables) whose values effect each other and thus are dependent, should be grouped together. Two places in a Petri net can be seen as dependent if they are connected to the same transition. Noack set up equation 4.2 to compute a structural criterion for each of the unprocessed places. In every step the algorithm weights the unprocessed places and selects the place with the highest value.

$$\mathrm{W}(p) := \frac{\sum\limits_{t \in {}^\bullet p} \frac{|{}^\bullet t \cap U|}{|{}^\bullet t|} + \sum\limits_{t \in p^\bullet} \frac{|t^\bullet \cap U|}{|t^\bullet|}}{|{}^\bullet p \cup p^\bullet|}. \tag{4.2}$$

This heuristic produces good variable orders for the nets I consider here. In [108] I drastically pushed the performance of the probabilistic model checker PRISM [78] by using this heuristic to compute a variable order for the MAPK model (see Table 6.1). However, for the proposed on-the-fly enumeration of state transitions with an early truncation of the IDD traversal, the heuristic can be improved.

The value of a state transition may depend on several variables. If the corresponding variables appear above the specified truncation layer, the computational effort to derive the actual function value may increase significantly. The variables should be located below this layer to reduce this effort. In this case the actual function value is computed only one time in the initialization step. Though this replacement of function instances by constant values may decrease the runtime it may increase at the same time the memory consumption if a function instance is shared among several tuples $(t, v, \hat{v})$. When computing the function values for all possible assignments the compression effect possibly gets lost.

Noack's ordering algorithm creates the order starting at the terminal node level. An adaption should select places which are used as variables in many transition functions as early as possible. This can be achieved if the computation of weights takes into account the occurrence of each place in all transition functions $\#_p := |\{t \mid p \in dom(f_t)\}|$. A high value $\#_p$ should give a high value for $\mathrm{W}(p)$.

In the implementation of Equation 4.2, I replaced some "magic" constants with the values of $\#_p$. This modification will not accelerate the enumeration of state transition in any case. If each transition function depends on its pre-places, which is often the case for biological networks, we may also observe a degradation.

If all transition functions are constants (KANBAN, Appendix A.2.2) the occurrence of places is zero and will not affect the variable order computation.

Algorithm 23 (Enumeration policy – LINE)

```
1  struct EnumerateLINE
2
3     base : BASE //base operation
4     gather : GATHER //gather policy
5     x_j : unsigned
6
7     func firstUse(v : unsigned, td : list of TransitionData)
8        if var(v) ≠ x_j then return true fi
9        it : Iterator
10       list : list of Iterator
11       row = 0
12       v̂_src := td_1.v̂
13       i_src := td_1.index
14       for 2 ≤ i ≤ |td| do
15          entries := map(v̂_src)[(td_i.t, v, td_i.v̂)]
16          if entries ≠ ∅ then
17             forall e_j ∈ entries do
18                value := e_j.f(td_i.args)
19                it := createIterator(e_j.indices, value)
20                if it.rowIndex = 0 then
21                   base(it.row, it.col, it.value)
22                   it.next()
23                fi
24                if it.hasMore() then list.insertAt(it.rowIndex, it) else release(it) fi
25             od
26          fi
27       od
28       gather(row, row)
29       rowIndex := 1
30       while rowIndex < list.size() do
31          it = list.getAndRemoveFirstElementAt(row)
32          while it do
33             base(it.row, it.col, it.value)
34             it.next()
35             if it.hasMore() then list.insertAt(it.row, it) else release(it) fi
36          od
37          gather(row, row)
38          row := row + 1
39       od
40       return false
41    end
42 end
```

Algorithm 24 (Enumeration policy – MULTI)

```
1  struct EnumerateMulti
2
3      base : BASE //the type of Operation
4      x_j : unsigned
5
6      func firstUse(v : unsigned, td : list of TransitionData)
7          if var(v) ≠ x_j then return true fi
8          v̂_src := td_1.v̂
9          i_src := td_1.index
10         for 2 ≤ i ≤ |td| do
11             entries := map(v̂_src)[(td_i.t, v, td_i.v̂)]
12             if entries ≠ ∅ then
13                 values[size(entries)] : array od double
14                 forall e_j ∈ entries do
15                     values[e_j] = e_j.f(td_i.args)
16                 od
17                 step := td_i.lsb
18                 row := i_src + lsb · |S|
19                  col := td_i.index + (lsb + td_i.shift) · |S|
20                 while step < td_i.usb do
21                     forall e_j ∈ entries do
22                         e_j.process(row, col, values[e_j], base)
23                     od
24                     row := row + |S|
25                     col := col + |S|
26                     step:=step + 1
27                 od
28             fi
29         od
30         return false
31     end
32 end
```

4.3.4 First Results

The proposed on-the-fly approach and the discussed improvements have been implemented in the model checker MARCIE [109]. I will describe the implementation of its numerical engine in the next Chapter. At this point I present some first results meant to illustrate the potential of the implementation. Therefor I took eight case studies (see Appendix A.2) and enumerated the entries of their rate matrices. The results are shown in Figure 4.14 and Figure 4.15.

I measured the overall memory consumption and the average time to enumerate all matrix entries using

- the Compressed **SPARSE** Row (Section 4.1) storage scheme,
- the on-the-fly method with enumeration policy **ALL**,
- the on-the-fly method with enumeration policy **LINE**.

I used for all experiments an *empty* base operation. The implementation requires to allocate at least on computation vector with double precision in the size of the state space. It is contained in the measured memory consumption (left inner plot). The outer plot shows the time per iteration as the average of ten sequential iterations. The right inner plot shows the total runtime and indicates the initialization effort. For all models I considered the possible truncation layers starting with two. A value of one would represent a genuine on-the-fly computation. But even for a value of two the achieved runtime is not acceptable.

The test system is MAC Pro 4×2.2 GHz with 32 GB RAM running Cent OS 5.5.

It stands out that the on-the-fly computation can compete with the sparse matrix storage with regard to runtime for higher truncation layers if the enumeration of single lines is not relevant. Concerning memory consumption the on-the-fly approach outperforms the sparse matrix storage in all cases. A line-wise enumeration must be paid with a runtime increase of a factor between five and ten.

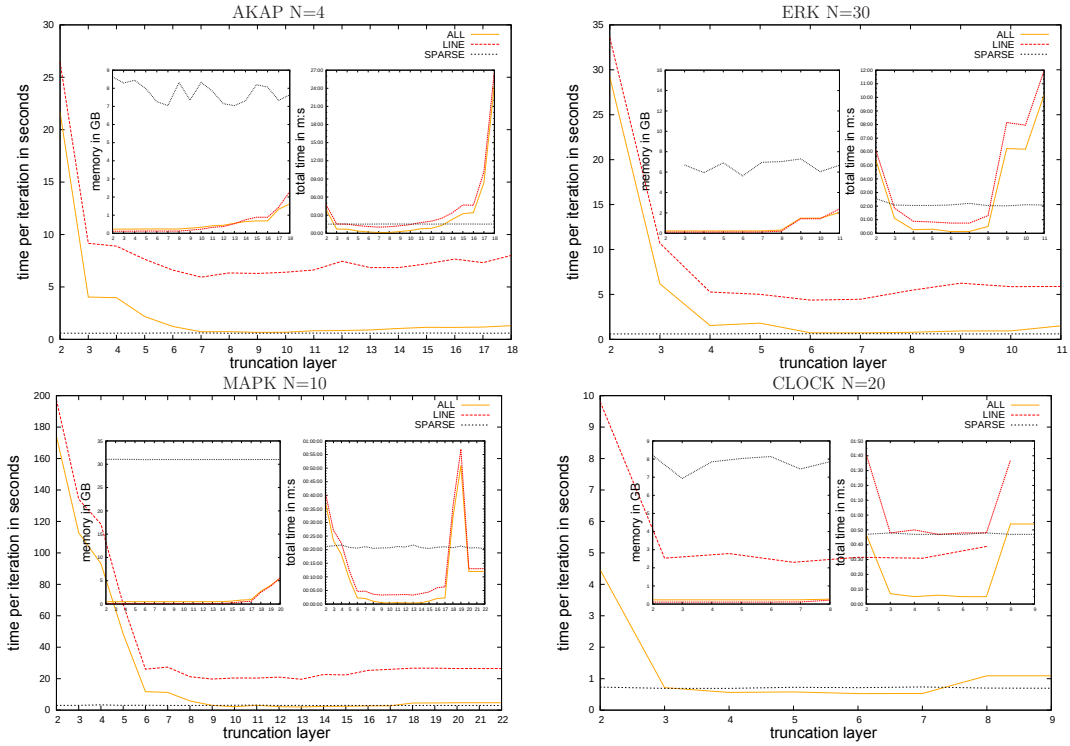


Figure 4.14: Time per iteration, memory consumption and total time for the enumeration of the rate matrix of biologic models with different truncation layers. A truncation layer of one would represent a genuine on-the-fly computation.

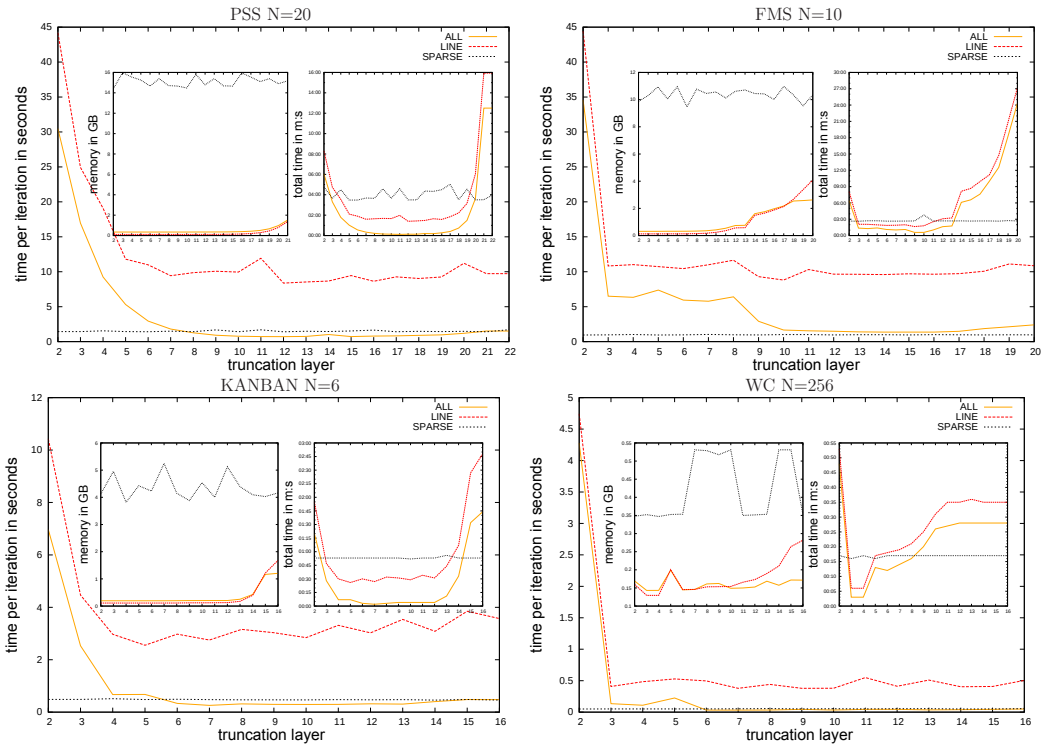


Figure 4.15: Time per iteration, memory consumption and total time for the enumeration of the rate matrix of technical models. A truncation layer of one would represent a genuine on-the-fly computation.

Figure 4.16 illustrates the positive effect of the described modification of the variable order heuristics for the FMS model (see Appendix A.2.2). In this case the better time per iteration comes with an in trend higher initialization time and also with an increased memory consumption. However, in the first half of the truncation layer range an analysis can profit of the modified variable order, provided an sufficient number of iterations.

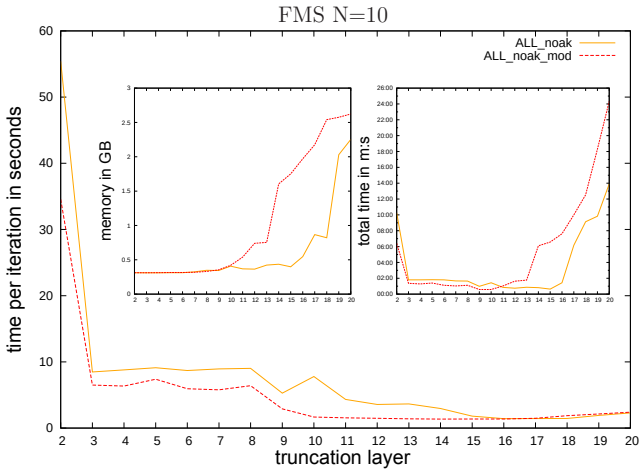


Figure 4.16: Time per iteration, memory consumption and total time for the enumeration of the rate matrix of the FMS system for different variable orders.

4.4 Summary

In this chapter I discussed advanced representation techniques for large matrices in the context of CTMCs. I sketched briefly the basic ideas of encoding techniques relying on Kronecker algebraic expressions and Multi-terminal Decision Diagrams. As the main contribution of the thesis I proposed a symbolic on-the-fly enumeration of the matrix entries given a state space representation in terms of Interval Decision Diagrams. I discussed further how to improve the approach to achieve an acceptable runtime and to realize different enumeration policies of the state transitions.

5 Implementation of Numerical Solvers

In the previous chapters I have presented generalized stochastic Petri nets and stochastic rewards nets as high level formalisms to define continuous-time Markov chains and rate-based Markov reward models as well as numerical methods to compute probability distributions and secondary measures. As a useful technique for the evaluation of such models I have further considered model checking techniques and the Continuous Stochastic Reward Logic as a powerful language to formulate quantitative model properties. To tackle the state explosion problem, I proposed an on-the-fly computation of the related matrix based on a symbolic state space representation by means of Interval Decision Diagrams.

This chapter rounds up these considerations with a brief discussion of some implementation concepts of numerical solvers. The analysis methods I outlined are available in the model checker *MARCIE* [109].

The tool is written in *C/C++* and uses the IDD library of A. Tovchigrechko [115]. *MARCIE's* multi-threaded symbolic numerical engine builds on the approach I proposed in Chapter 4. The CSRL model checker represents a further important contribution of this thesis. Figure 5.1 shows the architecture of the tool. The highlighted components have been developed by me while pursuing the Ph.D. project.

5.1 Concepts

Two features influenced the development of *MARCIE's* numerical engine, which are responsible for *MARCIE's* performance and they are supposed to ease the integration of further advanced analysis techniques. The iterative methods rely on a **generic** design of the **solver classes**. The use of *C++* with its support of parameterizable classes by templates enables to extend the implementation without using polymorphism at runtime. The numerical solvers execute tasks, realized as functor objects, in most of the cases **concurrently**. For short, **important numerical solvers are multi-threaded and enjoy a generic design**.

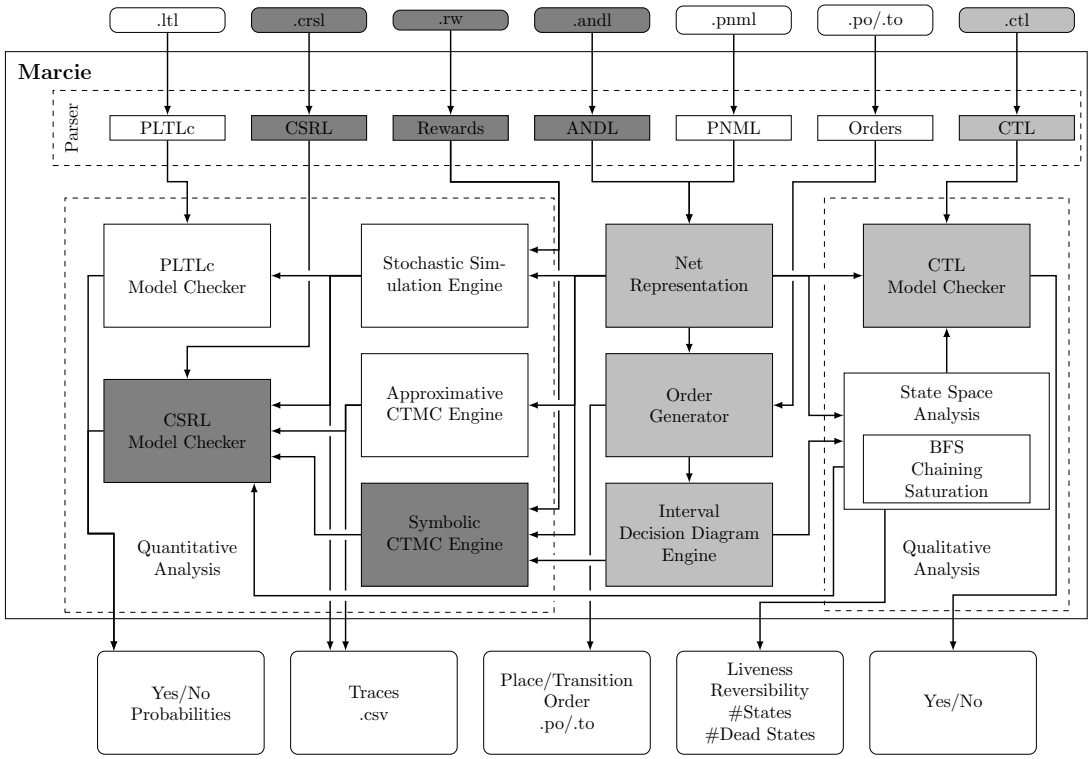


Figure 5.1: *MARCIE's* architecture [59]. The gray-colored components have been implemented, the lightgray-colored components have been upgraded by me.

When dealing with large models, it is common to use iterative methods. This holds for transient as well as steady state analysis. In Section 3.3 I sketched related methods, which can be characterized by the following algorithmic skeleton.

```
1 PrepareComputation
2 while not finished do
3     Multiplication
4     Gathering
5 od
6 PrepareResults
```

The most important, concerning runtime the most expensive, tasks are the *Multiplication* and the *Gathering*. Both tasks are intrinsically tied to each other

in the method of Gauss-Seidel. However, they can be separated and even broken into smaller sub-tasks in the case of uniformization or the method of Jacobi.

For the definition of the single tasks in MARCIE I applied a *C++*-specific design concept, namely *Policy-based Design*. In the following I will explain the underlying ideas and illustrate the benefits of this concept for *MARCIE's* implementation.

5.1.1 Policy-based Design

The term originates from [1]. I think the following quotation, given there, explains the goal of it best: "In brief, policy-based class design fosters assembling a class with complex behavior out of many little classes (called policies), each of which takes care of only one behavioral or structural aspect. ... Because you can mix and match policies, you can achieve a combinatorial set of behaviors by using a small core of elementary components". It should be clear why this concept is applied in MARCIE, as I already used the term *policy* in Chapter 4 and started to combine and nest several policies.

Policy-based design can be seen as a language-specific concept exclusively[1] offered by the programming language *C++* where classes and functions can be defined in a parameterizable fashion. So it is possible to define

- classes where the type of some members or the superclass(es)
- (member) functions where the type of some arguments or the result type

is not given explicitly. The not specified types are represented by placeholders which are replaced at compile time. Suchlike parts of the code are called templates. The *C++* compiler instantiates templates with existing types and generates thus new types. The template mechanism complements object-oriented programming and represents an alternative approach for generic programming. A recommendable textbook on *C++*-templates is [116]. While inheritance and polymorphism enable to realize context-dependent behaviour at runtime, the template mechanism provides similar effects at compile time. This concept is also known as *static polymorphism*.

I want to give an example why the difference can be – and in my opinion indeed is – crucial. For the implementation of the multiplication of a matrix and a vector one could encapsulate the actual operation using the following *C++* structs.

[1] currently

```
struct OpBase{

        //abstract function wich MUST be defined by derived classes
        virtual void multiply(int row, int col, double value) = 0;
};

struct MatrixVector : public OPBase {
        //the vectors v and r

        virtual void multiply(int row, int col, double value){
                r[row] += v[col]*value;
        }
};

struct VectorMatrix : public OPBase {
        //the vectors v and r

        virtual void multiply(int row, int col, double value){
                r[col] += v[row]*value;
        }
};

void multiply(CSRMatrix &M , OPBase op*){
        for(IndexT i = 0 ;i < M.rows; i++ ){
                IndexT i_0 = M.row[i];
                IndexT nnz = M.row[i+1];
                r[i] = 0;
                for ( IndexT i_0 = M.row[i];  j < nnz ; j++) {
                        op->multiply(i,M.col[j],M.val[j]);
                }
        }
}
```

The used technique to represent the matrix (CSR in the code snippet above) does not really matter in this context. More importantly: the algorithm may perform a matrix-vector or a vector-matrix multiplication depending on the type of the pointer *op*. The related enumeration procedure appears in the code only one time and is parameterizable. The evaluation of the pointer type happens in this case at runtime, and produces an in general negligible overhead. But, as we consider matrices with billions of entries, this overhead will be noticeable.

A more notable observation is that each implementation of the function *multiply* possesses only one line of code causing an expensive procedure call. This will definitely produce an huge overhead. *C++* compilers are able to replace function calls by the function's contents. This feature is known as *inlining*[2] and may significantly increase the performance of an application. However, in general virtual functions can not be inlined and the parameterized code comes with a price.

[2] The keyword **inline** makes a suggestion to the compiler.

Fortunately, this price has not to be paid as templates can help. The following *C++* code snippet deploys static polymorphism. The function *multiply* is defined as a template function. The classes *MatrixVector* and *VectorMatrix* have no shared superclass and declare the function *multiply* as **inline**. The genuine routines for the multiplication are created by the compiler using template instantiation, hopefully with inlining.

```
struct MatrixVector{
        //the vectors v and r
        inline void multiply(int row, int col, double value){
                r[row] += v[col]*value;
        }
};

struct VectorMatrix{
        //the vectors v and r
        inline void multiply(int row, int col, double value){
                r[col] += v[row]*value;
        }
};

template <typename OP>
void multiply(CSRMatrix &M , OP *op){
        /* code as before */
}
```

I promise to bother not the reader with *C++* code anymore. I think for the remainder the pseudo-code style will suffice to mediate the basic ideas. Types which would be declared in *C++* as template type names are highlighted in bold.

Let us strain the multiplication operation of a matrix with a vector as the example to motivate the use of policies. In *MARCIE*, the core of any kind of multiplication consists of the IDD traversal in Algorithm 21. The algorithm is the same for all numerical methods, which I consider, but lefts open three aspects, in particular:

- the *enumeration* strategy,
- the *fire* direction, and
- the *base* operation.

These different aspects (policies) permit to compose a multitude of multiplication types.

Take for instance the simple transient analysis given in Algorithm 8. The description hides several details, such as the representation of the matrix diagonal, or the fact that $\underline{\pi} := \underline{\pi} \cdot \mathbf{P}^{\mathcal{U}}$ requires two vectors (in the following $\underline{v}$ and $\underline{r}$).

Here, the exit rates are the result of the matrix-vector multiplication $\mathbf{R} \cdot \underline{1}$, and

stored in the dense vector *diag*. Algorithm 25 shows a refined version which also checks, whether the newly computed distribution has changed compared to the previous one, or whether a steady state has been reached[3]. It contains two different types of multiplication of a matrix and a vector. In line 6 it is the matrix-vector multiplication mentioned above. Translating this into the proposed on-the-fly approach means to combine the extraction policy ALL, the forward firing of Petri net transitions (FWD), and the following functor class RS ($RowSum$) specifying the basic operation.

```
1  struct RS
2      r : vector of double
3      proc operator()(rowIndex, colIndex : IndexT, value : double)
4          r[rowIndex] := r[rowIndex] + value
5      end
6  end
```

In line 17, it is the vector-matrix multiplication with the argument vector v. In this case we can combine the extraction policy ALL, the fire direction FWD and the functor class VM ($VectorMatrix$).

```
1  struct VM
2      r : vector of double
3      v : vector of double
4      proc operator()(rowIndex, colIndex : IndexT, value : double)
5          r[colIndex] := r[colIndex] + value · v[rowIndex]
6      end
7  end
```

Alternatively we can use the combination of ALL, the backward (BWD) firing and the functor class MV ($MatrixVector$).

```
1  struct MV
2      r : vector of double
3      v : vector of double
4      proc operator()(rowIndex, colIndex : IndexT, value : double)
5          r[rowIndex] := r[rowIndex] + value · v[colIndex]
6      end
7  end
```

[3] Such a transient analysis without accumulation and predefined iteration truncation is known as the Power Method.

Algorithm 25 (Uniformization – Refined)

```
1  func Uniformization(α : vector of double, ε,τ : double)
2
3      L, R : unsigned                                          }
4      w : vector of double                                     }
5      acc, v, r, diag: vector of double                        }
6      diag = R · 1                                             }
7      λ := max(diag)                                           }
8      FoxGlynn(L,R, w, λ·τ,ε )                                 }  Prepare
9      converged := false                                       }  Computation
10     v := α                                                   }
11     acc, r := 0                                              }
12     diag := (1 − diag)/λ                                     }
13     P^U := R/λ                                               }
14     k := 0                                                   }
15     while k < R and not converged do
16
17         r := v · (P^U)                                       }  Multiplication
18
19         r := r + v/diag                                      }
20         if k ∈ [L, R] then                                   }
21             acc := acc + w[k] · π                            }
22         fi                                                   }  Gathering
23         converged := diff(v, r)                              }
24         v := r                                               }
25         r := 0                                               }
26         k := k + 1                                           }
27     od
28     if k < R then                                            }
29         for j = k to j = R do                                }  Prepare
30             acc := acc + w[j] · v                            }  Results
31         od                                                   }
32     fi                                                       }
33     return acc
34 end
```

One detail still needs some further explanation. The uniformization procedure uses the matrix $\mathbf{P}^{\mathcal{U}}$ derived from $\mathbf{R}$ by dividing each element by λ. In the imple-

mentation I encode the uniformization into the rate functions of the transitions and define for each $t \in T$ the uniformized rate function $f_t^{\mathcal{U}} = \frac{f_t}{\lambda}$.

A similar discretization of the rate matrix is applied when the embedded Markov chain is considered, for instance to compute the probability to finally reach a set of states (see Section 3.3.2). In this case the discretization cannot be encoded in the rate functions, instead the following policy *MVE* (*MatrixVectorEmbedded*) is required.

```
1  struct MVE
2      r : vector of double
3      v : vector of double
4      diag : vector of double
5      proc operator()(rowIndex, colIndex : IndexT, value : double)
6          r[rowIndex] := r[rowIndex] + value/diag[rowIndex] · v[colIndex]
7      end
8  end
```

In the dialect of policy-based design, *ALL*, *FWD*, *BWD*, *MV*, *VM*, *MVE* and *RS* are called policy classes. The class specifying the traversal algorithm and which can be parameterized with the different policies, is called a host class. I name the host class *MatrixProcessor* (see Algorithm 26). Its internal type *ProcessingPolicy* is the result of combining the different policies.

The *C++* compiler instantiates a new data type for each specified policy combination. From a technical view point, there are several alternatives to realize the policy classes and their composition within a host class using aggregation or inheritance. At this point I forgo a more detailed discussion and refer to [1]. In any case, it is important to understand that policy-based design is a type generating approach. The *C++* compiler generates new data types by template instantiation, automatically and concise. In the numerical core of the PRISM model checker [78], which is also written in *C++*, the different solver classes are handcrafted by adapting copies of the same algorithm.

5.1.2 Multi-threading

In Section 3.3 I already alluded to the opportunity to improve the method of Jacobi by means of parallelization. Parallelization basically means to break a big problem into several smaller sub-problems which can be solved concurrently by different processes using multiple processing units. The type of the processes,

Algorithm 26 (Matrix processor)

```
1  struct MatrixProcessor
2     struct ProcessingPolicy
3        enumeration : Enumeration //firstUse, secondUse
4        base : BaseOperation // op
5        direction : FireDirection //selectActionList, selectArgument
6     end
7
8     G_S : IDD
9     T : set of transitions
10
11    proc operator()()
12       proc : struct ProcessingPolicy
13       // configure the processing policy
14       EnumerateTransitions(G_S, T, proc) // Algorithm 21
15    end
16 end
```

the memory access model, the architecture of the processing units and their connection delineate a fine-grained classification of parallel computing techniques. I confine myself to distinguish here only approaches based on the memory model, in particular distributed memory approaches and shared memory approaches. Both have already been considered for the analysis of large Markov models [73, 71, 79, 14].

Distributed memory. In the distributed memory approach, the participating processes reside on different machines with their own memory. The connection of these machines allows to distinguish for instance grids, where the single machines are connected via the world wide web, or clusters, where the connection consists of a fast network or bus system.

The major benefit is, next to the workload distribution of the actual computation, the increase of the available memory. The drawback is a possibly expensive communication and a challenging partitioning of the computational problem, especially if data dependencies have to be taken into account. Distributed analysis of large Markov chains is reported for instance in [73] and [14].

Shared memory. In this case the participating processes share the same main memory, and we can distinguish between heavy-weight processes having their own address space and light-weight *threads*, sharing the address space of an superordinate process. Processes are completely independent of each other, and due to their own address space, any communication, as for instance for synchronization or for data exchange, is based on message passing.

Threads also run independently, but their communication is based on shared data, with all benefits and troubles. The ability of direct access to the same data by different threads without communication overhead enables to achieve high efficiency, but requires special discipline at the developer side due to the missing protection against inconsistent memory access.

Multi-threaded uniformization-based transient analysis with Kronecker representations has been considered in [71]. In [79] the authors propose a multi-threaded solver for Gauss-Seidel based on an MTBDD encoding of the model.

Multi-threading is a core feature of *MARCIE's* numerical engine and is currently supported by several solvers. Modern programming languages as Java and recently *C++* offer skilled developers concepts to easily create robust multi-threaded applications[4]. Therefor this section will discuss the partition of the actual computational tasks rather than the implementation details or the use of concurrency patterns.

In brief: Computational tasks as the *Multiplication* and the *Gathering* are refined into several sub-tasks and processed concurrently by different threads. As the *Gathering* must happen after the *Multiplication*, and the next iteration can start after finishing the *Gathering*, the implementation requires some kind of *barrier* to synchronize the involved threads at the end of each computation phase. *MARCIE* uses the concept of a *threadpool* to achieve such behaviour. A group of so-called worker threads gets associated a synchronized queue of tasks. A user of the pool puts its tasks in the queue and waits until all tasks have been processed. Thereby all aspects of concurrency in the numerical solvers are encapsulated in an instance of the class *ThreadPool* which offers the function *SubmitTasksAndWait*. A caller of this functions, any iterative algorithm, will be suspended until the worker threads have finished their submitted tasks. For the implementation of thread pools in *C++* I refer to [118]. In the following I concentrate on the definition of the tasks based on a simple state space partition.

[4] *MARCIE's* first version was based on the *pthread* library. When writing the thesis, the implementation was prepared to make use of the multi-threading facilities offered by the *C++11* standard.

Partitioning of the state transition relation. The definition of tasks which can be processed concurrently by several threads requires to break the state transition relation $\mathbf{R}$ of a large Markov chain into smaller pieces.

As $\mathbf{R}$ is implicitly given by the reachable states $\mathcal{S}$ of a bounded SPN and its set of transitions T, a partition of $\mathbf{R}$ must be specified by means of $\mathcal{S}$ and T. A proper definition of the following issue requires operations similar to $preImg(S)$ and $Img(S)$ (see Section 2.2.2) which represent sets of state transitions. Let us define them as

$$preTr(S, \mathcal{T}) := \{(s, t, s') \mid s \in S, s' \in S_N, t \in \mathcal{T} : s' \xrightarrow{t} s\}$$

and

$$postTr(S, \mathcal{T}) := \{(s, t, s') \mid s \in S, s' \in S_N, t \in \mathcal{T} : s \xrightarrow{t} s'\}.$$

Algorithm 21 enumerates a set of state transitions defined by an arbitrary subset of the reachable states and an arbitrary subset of the Petri net transitions, and we can partition the transition relation of the corresponding model as follows.

For a Petri net $N = [P, T, V, V_R, V_I, s_0]$ with the set of reachable states $\mathcal{S}$ a state transition partition $\mathcal{P}_N \subseteq (2^{\mathcal{S}} \times 2^T)$ is a set featuring the following standard properties of any partition:

1. $\mathcal{P}_N = \{(S', \mathcal{T}) \mid (S' \subseteq \mathcal{S}) \neq \varnothing \wedge (\mathcal{T} \subseteq T) \neq \varnothing\}$
   Each element is a tuple consisting of a non-empty subset of the reachable states of N and a non-empty subset of the Petri net transitions T.
2. $\bigcup_{(S', \mathcal{T}) \in \mathcal{P}_N} postTr(S', \mathcal{T}) = postTr(\mathcal{S}, T)$
   The set of state transitions $\mathbf{R}$ is complete.
3. $\bigcap_{(S', \mathcal{T}) \in \mathcal{P}_N} postTr(S', \mathcal{T}) = \varnothing$
   There is no state transition which is defined in the context of two or more elements of $\mathcal{P}_N$.

Example 13

For a Petri net $N = [P, T, V, V_R, V_I, s_0]$ with the reachable states $\mathcal{S}$

- $\mathcal{P}_{N_1} = \{(\mathcal{S}, T)\}$
- $\mathcal{P}_{N_2} = \bigcup_{s \in \mathcal{S}} \{(s, T)\}$
- $\mathcal{P}_{N_3} = \bigcup_{t \in T} \{(\mathcal{S}, t)\}$

are trivial state transition partitions.

★

Having a transition partition $\mathcal{P}_N$, we can apply the function *EnumerateTransitions* (Algorithm 21) to each of its elements fully independently of the others and thus concurrently, as far as the specified operation does not involve shared write access to any data. If we were combining the extraction policy *ALL*, the fire direction *FWD* and the functor class *VM* (*VectorMatrix*)

```
1 struct VM
2     r : vector of double
3     v : vector of double
4     proc operator()(rowIndex, colIndex : IndexT, value : double)
5         r[colIndex] := r[colIndex] + value · v[rowIndex]
6     end
7 end
```

to realize a parallel vector-matrix multiplication, it is generally possible to have a race condition when updating vector $\underline{r}$.

However, if the partition preempts potential race conditions per se, it is not critical to perform the single tasks in parallel.

Lexicographic state transition partition. In [71] it is argued to use a column-based partition for a vector-matrix multiplication because this preempts write conflicts and does not require several computation vectors. In *MARCIE* any multiplication is a matrix-vector multiplication. If an analysis method requires a vector-matrix multiplication, it is realized by transposing the matrix. As the matrix entries are enumerated on-the-fly, transposing means to fire the Petri net transitions backwards. Thus I consider only row-based partitions.

I describe now how to derive efficiently a row-based partition to be used with the method of Jacobi and the uniformization-based transient analysis, and thus also for the computation of performability. It is based on a partition of the lexicographic indices of the reachable states. Each element (S', T) contains a set of states, representing a successive block of rows, and the complete set of Petri

net transitions T[5]. For a given size k, the partition is defined as

$$\mathcal{P}_k = \{(S_1, T), (S_2, T), \ldots, (S_k, T)\}$$

with $S_{j<k} = \{s \mid (j-1)\cdot|\mathcal{S}|/k \leq \imath_s < (j)\cdot|\mathcal{S}|/k\}$ and $S_k = \{s \mid (k-1)\cdot|\mathcal{S}|/k \leq \imath_s < |\mathcal{S}|\}$[6].

Correctness. The specified partition fulfills the criteria to be a state transition partition, because

1. all considered states are reachable states, and
2. all Petri net transitions are fired in all their enabling states exactly one time.

This state transition partition is used in all multi-threaded solvers. Algorithm 27 computes the required state sets S_j given $\mathcal{S}$ and k, reusing the Algorithm 15 with the functor *AmountOfStates*. Each call of the function *SelectStates* returns the IDD representing the $|\mathcal{S}|/k$ lexicographic smallest states contained in the unprocessed state set U[7].

5.1.3 A Generic Solver

It is time to assamble the outlined ideas to define the skeleton of a parameterizable algorithm to instantiate the different iterative numerical solvers of MARCIE's multi-threaded model checker for the Continuous Stochastic Reward Logic.

A generic solver. Algorithms 28 shows a possible implementation. The solver is parameterized with the type of analysis. The function *solve* undergoes a preparation phase, the iterative computation phase, and finally a phase of result preparation. The computation phase is the loop which first processes the matrix and then gathers the computed results. The loop's termination criterion is specified by the analysis type. The single steps are realized as callable objects, created by an instance of the analysis type. The solver possesses a *ThreadPool* instance and submits to it the tasks performing the actual numerical computation. These tasks are generated in a setup step using designated functors offered by an instance of the analysis type.

[5] The idea builds on the optimistic assumption that the number of state transitions is evenly distributed over the states.

[6] k is supposed to be the number of threads, which in turn, represent the system's concurrency.

[7] The last element has $|\mathcal{S}|/k + |\mathcal{S}| \bmod k$ states.

Algorithm 27 (Lexicographic partition)

```
1  struct LexicographicPartition
2     struct AmountOfStates
3        size : unsigned
4        func operator()(index : IndexT)
5           size := size – 1
6           return size > 0
7        end
8
9        func checkAll(index : IndexT, index' : IndexT)
10          if index' – index > size then return false fi
11          size := size + index – index'
12          return true
13       end
14
15       func checkNone(index : IndexT, index' : IndexT)
16          if size > 0 then return false fi
17          return true
18       end
19    end //AmountOfStates
20
21    func Create(G_U : IDD, T : set of transitions, k : unsigned)
22       size := |U|/k
23       slt : AmountOfStates
24       P_N := ∅
25       for 1 ≤ j < k do
26          slt.size := size
27          G_Sj := SelectStates(G_U,slt) //Algorithm 15
28          G_U := diff(G_U, G_Sj);
29          P_N := P_N ∪ {(G_Sj, T)}
30       od
31       P_N := P_N ∪ {(G_U, T)} //the remaining states represent S_k
32       return P_N
33    end
34 end
```

Algorithm 28 (Generic solver)

```
1  struct Solver
2     analysis : AnalysisType
3     threads : ThreadPool
4     gather : set of GatherTasks
5     multiply : set of Multiplications //specific type of MatrixProcessor
6
7     func setUp(k : unsigned )
8        P := analysis.createStateTransitionPartition(k));
9        for p ∈ P do
10          multiply.add(analysis.createMultiplication(p)
11          gather.add(analysis.createGathering(p))
12       od
13       init := analysis.createInitializer()
14       init()
15       TerminationPolicy terminate := analysis.createTerminationPolicy()
16       forall g ∈ gather do
17          g.check := terminate
18       od
19    end
20
21    func cleanUp()
22       deleteTasks();
23    end
24
25    func solve()
26       iteration := 0
27       while not terminate(iteration) do
28          threads.SubmitTasksAndWait(multiply)
29          bind(gather, iteration) //bind the argument to the functors
30          threads.SubmitTasksAndWait(gather)
31          iteration := iteration + 1
32       od
33       prepare := analysis.createResultPreparator()
34       prepare(iteration)
35    end
36 end
```

A transient solver. An exhaustive discussion of the implementation of all available solvers would go beyond the scope of this thesis. However, I think it is worth illustrating the composition of a transient solver for the computation of the vector $\underline{\pi}^{C}_{\tau,S'}$ based on the fast backwards-uniformization described in [68].

For this purpose the transitions fire in forward direction (FWD)[8]. The multiplication type is *MatrixVector* (MV). The results are gathered (accumulated) after a complete multiplication of the matrix $\mathbf{P}^{\mathcal{U}}$; thus the enumeration policy is ALL. I call the resulting matrix processor *CompleteMultiplication*. The specified multiplication can be performed concurrently and the solver uses the *LexicographicPartition*. It remains to define dedicated functor classes for the initialization step, the gathering and the preparation of results. The details

Algorithm 29 (Initialization functor – Uniformization)

```
1  struct InitUniformization
2      ctmc : [N_S,S]
3      vecs : Vectors
4      G_S' : IDD
5      fgw : FoxGlynnWeights
6      τ, ε : double
7      proc operator()()
8          vecs.diag := R · 1
9          λ = max(vecs.diag)
10         fgw.generate(λ · τ, ε)
11         forall t ∈ N_S.T do
12             f_t^U := f_t/λ //uniformize the transition relation
13         od
14         vecs.diag := 1 − vecs.diag/λ
15         InitStates(G_S', vecs.v, 1.0) // Algorithm 14
16     end
```

of the implementation of the algorithm of Fox and Glynn [50] are encapsulated in the structure *FoxGlynnWeights* which yields access to the members L, R and pw, the actual Poisson weights. The function *generate* initializes the internal data depending on the uniformization constant λ. At first I de-

[8] This may seem confusing: the probabilities are propagated backwards by a matrix-vector multiplication. An on-the-fly matrix-vector multiplication requires a forward firing of the Petri net transitions.

fine the functor class *InitUniformization* given in Algorithm 29. An instance encapsulates the related data and takes care of their initialization when the GenericSolver calls its function call operator. Further I define the functor class *GatherTransientWithConvergenceCheck* in Algorithm 30. Its purpose is the accumulation of the transient probabilities in the vector *acc* and the preparation of the vectors *v* and *r* for the next iteration. These vectors are shared among all functor instances. It checks also whether the computed probabilities differ significantly from the previous computation step. An instance considers the contents of the vectors within the positions *lb* and *ub*. If the individual instances define a partition of the vector entries, what I assume here, the gathering can be performed without interference by different threads. A preparation of the computed

Algorithm 30 (Gather functor – Uniformization)

```
1  struct GatherTransientWithConvergenceCheck
2     struct Vectors
3        r : vector of double
4        v : vector of double
5        diag : vector of double
6        acc : vector of double
7     end vecs
8     ub : IndexT
9     lb : IndexT
10    fgw : FoxGlynnWeights
11    check : TerminationPolicy
12    func operator()(iteration : unsigned)
13       for lb ≤ i < ub do
14          old := vecs.v[i]
15          vecs.r[i] := vecs.r[i] + vecs.v[i]/vecs.diag[i]
16          check(old, vecs.v[i])
17          vecs.r[i] = 0
18          if iteration ∈ [fgw.L, fgw.R] then
19             vecs.acc[i] := vecs.acc[i] + fgw.pw[iteration − pw.left] · vecs.v[i]
20          fi
21       od
22    end
23 end
```

results is required if the iteration procedure is truncated due to the detection of

a steady state. In this case the iteration number is below truncation point R, and the computed stable probability distribution must be weighted and accumulated for the remaining iterations. This is done by one instance[9] of the functor class *ProcessRemainingIterations* in Algorithm 31. To evaluate for example the CSL

Algorithm 31 (Result preparation functor – Uniformization)

```
1  struct ProcessRemainingIterations
2     vecs : Vectors
3     ub : IndexT
4     lb : IndexT
5     fgw : FoxGlynnWeights
6     proc operator()(iteration : unsigned)
7        for iteration ≤ j < fgw.R do
8           if i ≥ fgw.L then
9              for lb ≤ i < ub do
10                vecs.acc[i] := vecs.acc[i] + fwg.pw[j − fgw.L] · vecs.v[i]
11             od
12          fi
13       od
14    end
15 end
```

(see Section 3.4) formula

$$\mathcal{P}_{<p}[\Phi \mathbf{U}^{[\tau,\tau]} \Psi],$$

we can finally use the procedure in Algorithm 32.
The actual class*BWDTransientAnalysis* (not given), which is used by the GenericSolver, acts as a factory for the single tasks and specifies for this purpose the procedures *CreateInitializer*, *CreateMultiplication*, *CreateGathering* and *CreateResultPreparator*.

5.2 Miscellaneous

In this section I address some aspects which may need some further explanation, as for instance the extensions which are necessary to deal with generalized stochastic Petri nets or stochastic reward nets.

[9] This step can also be parallelized.

Algorithm 32 (Evaluation of a CSL formula)

```
1  func EvalUntilAtTime(ctmc : [N_S, S], τ, p: double, Sat(Φ), Sat(Ψ): IDD)
2      analysis : BWDTransientAnalysis
3      analysis.init := Sat(Ψ)
4      analysis.abs := diff(S, union(Sat(Φ), Sat(Ψ)) //¬Φ ∧ ¬Ψ
5      analysis.ctmc := copy(ctmc) //the solver may change the CTMC
6      analysis.τ := τ
7      solver : GenericSolver
8      solver.analysis := analysis
9      solver.solve(System.concurrency())
10     return StatesLessThen(S, analysis.vecs.acc, p) // Algorithm 16
11 end
```

5.2.1 Parker's Pseudo-Gauss-Seidel

The experimental results given in Section 4.3.4 indicate that an efficient implementation of a Gauss-Seidel solver using the on-the-fly approach is a severe problem. The line-wise enumeration of the state transitions is bought with high runtime overhead. A similar problem with this method was identified by Parker [96] in the context of MTBDD based analysis of Markov models[10]. He came up with an hybrid of the methods of Gauss-Seidel and Jacobi, called Pseudo-Gauss-Seidel (PGS). The idea is quite simple. In the method of Jacobi, the computation vector is updated after the enumeration of the complete matrix. In the method of Gauss-Seidel, the update is done after enumerating the state transitions of a single line. In PGS, the update is done after a block of rows specified by the available entries in a node of the truncation layer. The hybrid algorithm requires less iterations than Jacobi. A second advantage is a decreased memory consumption as the second computation vector must only have the size of the largest block in terms of rows (Figure 5.2). A parallelization is indeed not obvious.

To realize a PGS solver, the *MatrixProcessor* is instantiated with *BLOCK* enumeration, combined with a dedicated immediate *Gather*-policy, and with *BWD* or *FWD* firing depending on the multiplication type.

[10] PRISM's efficient Gauss-Seidel is based on a different data structure which is closely related to matrix diagrams [88].

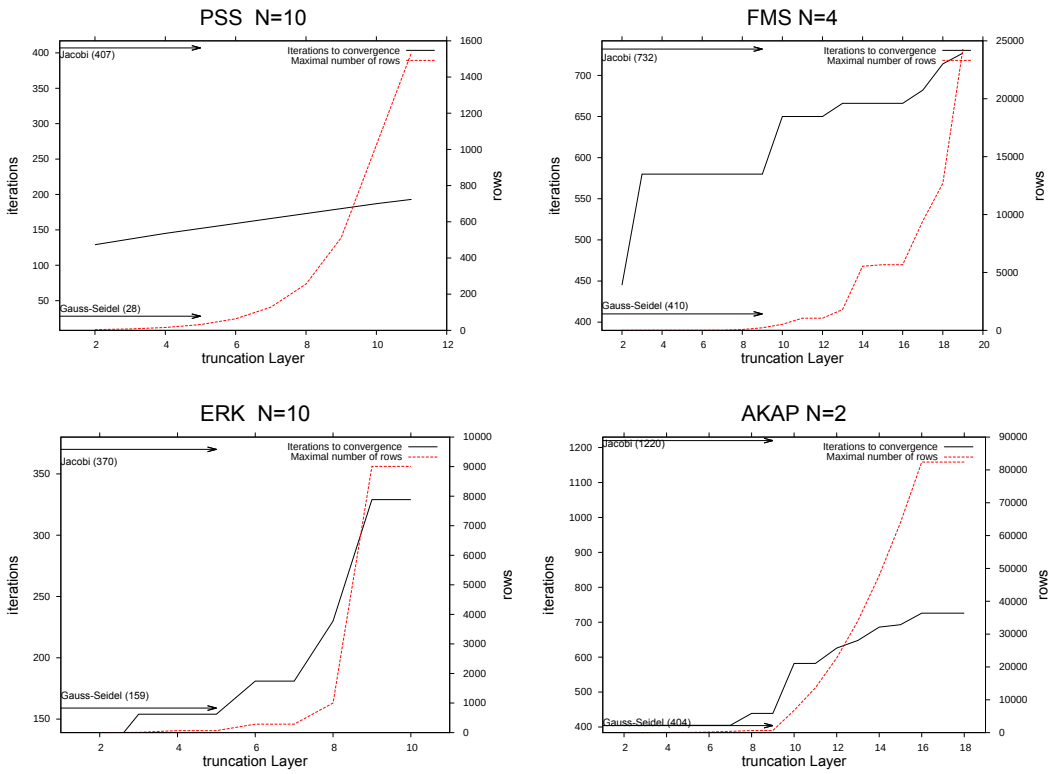


Figure 5.2: Pseudo-Gauss-Seidel: The number of iterations and the size of the temporary computation vector depending on the truncation layer for selected models. The required number of iterations for Gauss-Seidel and Jacobi are given for comparison.

5.2.2 Generalized Stochastic Petri Nets

The analysis of GSPN models requires an extended version of the multiplication. So far the multiplication treats all state transitions explicitly as induced by timed transitions enabled in tangible states. In the case of GSPN we have to consider also the immediate Petri net transitions firing in the vanishing states. Then the function of a transition does not specify a firing rate, but a weight which quantifies the transferred portion of probability. If an immediate transition is enabled and not in conflict with another one, the weight is 1. Otherwise the conflict resolution can be realized by deploying the concept of random switches (see Section 3.2.1). Given the symbolic state space representation of the reachable states $\mathcal{S}$ consisting of the tangible states S_T and the vanishing states S_V, Algorithm 33 can be used to compute the set $RS := \{(S_{V_1}, T_{I_1}), \ldots, (S_{V_m}, T_{I_m})\}$. Each tuple

$(S_{V_i} \subseteq S_V, T_{I_i})$ represents a set of vanishing states where the transitions in T_{I_i}, which are additional transitions in the GSPN, are enabled and fire with the probability defined by the Equation 3.1. RS is a compact encoding of the set of all random switches. To explain the different treatment of timed and immediate

Algorithm 33 (Computation of random switches)

```
1  proc ComputeRandomSwitches(N = [P, T = T_S ∪ T_I, V, V_R, V_I, F, s_0] : GSPN)
2
3     RS : ∅
4     proc AuxComputeRandomSwitches(S_{V_i} : set of states, T_r, T_c : set of transitions)
5        if S = ∅ then
6           return
7        end
8        if T_r ≠ ∅ then
9           t := SelectOneOf(T_r)
10          T_r := T_r ∖ {t}
11          S_e := enabled_t(S_{V_i})
12          S_d := S' ∖ S_e
13          AuxComputeRandomSwitches(S_e, T_r, T_c ∪ {t} )
14          AuxComputeRandomSwitches(S_d, T_r, T_c)
15       else
16          T_{I_i} := ∅
17          forall t ∈ T_c do
18             t' := createCopyOfTransition(t, N)
19             F := F ∪ {(t', h_t / Σ_{t''∈T_c} h''_t)}
20             T_{I_i} := T_{I_i} ∪ {t'}
21          od
22          RS := RS ∪ {(S_{V_i}, T_{I_i})}
23       end
24    end
25
26    S_V := enabled_{T_I}(R_N(s_0))
27    AuxComputeRandomSwitches(S_V, T_I, ∅)
28 end
```

state transitions, let us consider the $|\mathcal{S}| \times |\mathcal{S}|$ matrices

$$\mathbf{R}_T = \left(\begin{array}{c|c} \mathbf{R}_{TT} & \mathbf{R}_{TV} \\ \hline 0 & 0 \end{array}\right) \text{ and } \mathbf{P}_V = \left(\begin{array}{c|c} 0 & 0 \\ \hline \mathbf{P}_{TV} & \mathbf{P}_{VV} \end{array}\right),$$

based on the order of states (rows) $\underbrace{s_1 < \ldots < s_i}_{S_T} < \underbrace{s_{i+1} < \ldots < s_n}_{S_V}$, with

1. $\mathbf{R}_{TT}$ defines the rates of state transitions between tangible states
2. $\mathbf{R}_{TV}$ defines the rates of state transitions from tangible to vanishing states
3. $\mathbf{P}_{VT}$ defines the probabilities of state transitions from vanishing to tangible states
4. $\mathbf{P}_{VV}$ defines the probabilities of state transitions between vanishing states.

The complete set of state transitions is characterized by the matrix

$$\mathbf{U} = \mathbf{R}_T + \mathbf{P}_V = \left(\begin{array}{c|c} \mathbf{R}_{TT} & \mathbf{R}_{TV} \\ \hline \mathbf{P}_{TV} & \mathbf{P}_{VV} \end{array}\right).$$

We specify the actual rate matrix $\mathbf{R} = \mathbf{R}_T + \mathbf{P}_{VT} \cdot \mathbf{P}_{VV}^{\infty}$ [85] by replacing each possible path through the vanishing states by a timed state transition. Given that $\mathbf{P}_V$ defines an acyclic transition relation, there exists a k such that $\mathbf{P}_{VV}^{\infty} = \mathbf{P}_{VV}^{k}$, and we can rewrite $\mathbf{R}$ as $\mathbf{R}_T + \mathbf{P}_{VT} \cdot \mathbf{P}_{VV}^{k}$. It is not possible to compute $\mathbf{R}$ this way with the proposed on-the-fly approach. The enumeration of its entries requires the related SPN description, which can be generated using the reduction I already sketched in Section 3.2.1.

However, MARCIE supports the explicit analysis of GSPN by iteratively propagating the probability mass through the set of vanishing states. The entries of the matrix $\mathbf{P}_V$ can be enumerated by applying Algorithm 21 for the elements of RS. The state transitions have to be treated differently depending on the type of multiplication.

Vector-matrix multiplication. Let us first consider a vector-matrix multiplication. In this case, the vanishing states, reachable from tangible states in one step, receive a non-zero value representing the probability to reside there. This value moves immediately to their successor states. If the successors are vanishing states as well, this probability propagation repeats until the complete probability mass has been absorbed by tangible states. We can characterize this procedure

recursively as

$$\underline{v}^0 + (\sum_{i=1}^{k} \underline{v}^{i-1} \cdot \mathbf{P}_{VV}) \cdot \mathbf{P}_{VT} \text{ with } \underline{v}^0 = (\underline{v} \cdot \mathbf{R}_T)$$

and achieve this behavior with Algorithm 34.

Algorithm 34 (Vector-Matrix multiplication – GSPN)

```
1  o := 0
2  r := v · R_T
3  tmp := r
4  while o ≠ v do
5      o := v
6      r := v · P_VV
7      tmp := tmp + r
8      v := r
9  od
10 v := tmp · P_VT
11 r := v + tmp
12 InitStates(S_V, r, 0)
```

The first step is the multiplication with the matrix $\mathbf{R}_T$. The probability mass is propagated in a second step by the transitions connecting only vanishing states. The propagated probability mass is accumulated in the additional vector $\underline{tmp}$. When there is no further change, the accumulated mass is propagated in a last step from the vanishing states with a transition to tangible states.

Matrix-Vector multiplication. In case of a matrix-vector multiplication, the propagation through vanishing states is directed backwards and applied before multiplying with the rate matrix. We can rewrite $\mathbf{R} \cdot \underline{v}$ as $(\mathbf{R}_T + \mathbf{P}_{VT} \cdot \mathbf{P}_{VV}^k) \cdot \underline{v}$ and formulate the following recursive definition

$$\mathbf{R}_T \cdot (\underline{v} + \sum_{i=1}^{k} \mathbf{P}_{VV} \cdot \underline{v}^{i-1}) \text{ with } \underline{v}^0 = \mathbf{P}_{VT} \cdot \underline{v}$$

yielding in Algorithm 35.

The disadvantages of an iterative propagation are obvious.

Algorithm 35 (Matrix-Vector multiplication – GSPN)

```
1  o := 0
2  r := P_VT · v
3  tmp := v
4  v := r
5  while o ≠ v do
6      o:=v
7      r := P_VV · v
8      tmp := tmp + r
9      v := r
10 od
11 r := R^T · tmp
```

- The computation vectors have to be in the size of $\mathcal{S}$ and in general it holds that $|\mathcal{S}| >> |S_T|$.
- The accumulation of the propagated probabilities requires an additional $|\mathcal{S}|$-vector.
- The propagation is time-consuming.

In general the explicit treatment of immediate transitions has to be paid with a significant increase of memory consumption and runtime (see Figure 6.28).

Nevertheless, in [24] the authors discuss CSL model checking of GSPNs where atomic propositions refer to the marking of so-called *vanishing* places which would be removed when reducing the GSPN. The discussed iterative probability propagation permits to consider also such formulas[11].

5.2.3 Stochastic Reward Nets

For a stochastic reward net, specified by the SPN $N_S = [P, T, V, V_R, V_I, F, s_0]$ and the Petri net $N^{\varrho} = [P^{\varrho} \subseteq P, T^{\varrho}, V^{\varrho} = \varnothing, V_R^{\varrho}, V_I^{\varrho}, F^{\varrho}, \varnothing]$, *MARCIE's* analysis capabilities range from the computation of reward vectors, for instance to derive instantaneous and cumulative reward measures, to model checking of CSRL formulas. In the latter case *MARCIE* computes the distribution of the accumulated reward based on Markovian approximation (Section 3.2.2). For the sake

[11] When writing the thesis vanishing places are not considered by MARCIE's CSL model checker.

of completeness some important aspects shall be mentioned on this topic, too. However, the reader should not expect an in-depth discussion.

Computation of the reward vector. The computation of the reward vector is realized in the same way as the computation of the exit rate vector. The set of reachable states $\mathcal{S}$ and the reward transitions T^{ϱ} specify formally the matrix

$$\mathbf{M}_{\varrho}(s,s') = \begin{cases} \varrho_s & \text{if } s = s' \\ 0 & \text{otherwise .} \end{cases}$$

Its diagonal elements represent the reward vector $\underline{\varrho}$. To compute $\underline{\varrho} = \mathbf{M}_{\varrho} \cdot \underline{1}$ we can deploy the composition of ALL, FWD and RS with respect to the state transition partition $\mathcal{P}_{\varrho} := \{(\mathcal{S}, T^{\varrho})\}$.

Computation of performability. The computation of $v_{\alpha,\tau,y}^{[C,\varrho]}$ is basically a transient analysis in the context of the approximating SPN $N^{\mathcal{A}}$ the generation of which is described in Section 3.2.2. The solver is derived from the transient solver I discussed so far. Its distinguishing feature is that it considers the place p_y just implicitly. For this purpose the *MatrixProcessor* is instantiated with the enumeration policy $MULTI$. Given that the number of reward levels is l[12], all transitions ($T^{\mathcal{A}}$) fire within the step bounds $[0, l)$. When preparing the transition set $T^{\mathcal{A}}$, the solver does not create arcs to a place p_y, but sets the *shift* flag in the related *TransitionData* instances to true. This constitutes implicitly the transition relation $\mathbf{R}_{i,i+1}$ for all $0 \leq i < l$.

The computation vectors $\underline{v}, \underline{r}$ and $\underline{acc}$ must have the size $|\mathcal{S}| \cdot l$. To store the exit rates, the vector $\underline{diag}$ can have size $|\mathcal{S}|$. With exception of the absorbing level l, all reward levels possess the same exit rates. However, the last level absorbs the probability mass and is not relevant for computation of the transient probabilities of interest. Thus an access to an exit rate, independently of the reward level, can be redirected to the related exit rate of reward level $\mathbf{R}_0$. This requires to adapt the *Gathering* policy in Algorithm 30 at line *15* by a translation of the actual index.

CSRL specific adaptions. In the context of CSRL model checking, the computation of the distribution of the accumulated reward may require some adaptions which depend on the given time and reward interval. According to Section 3.4 I

[12] $l = \lfloor \frac{y}{\Delta} \rfloor + 2$

implicitly encode the level-dependent CSL formulas (Table 3.4) into the solvers. This affects the step bounds ul and lb, the initialization of the argument vector and also the size of the exit rate vector $\underline{diag}$ and the access to it.

- **Vector initialization** With the fast backward transient analysis [68] we are able to compute $\underline{Prob}^{C^{\mathcal{A}}}(\Phi \mathbf{U} \Psi)$ in one pass. The argument vector $\underline{v}$ is initialized in each position with the probability of the related state to reach a Ψ-state, which is 1 for all Ψ-states and 0 otherwise. This initialization must be done for the levels specified by the reward interval J. The different cases are given below.

J	$\underline{v}[i] = 1 \Leftrightarrow s_{(i \mod \lvert\mathcal{S}\rvert)} \in Sat(\Psi)$ and
$[y, y]$	$\lvert\mathcal{S}\rvert \cdot (l-2) \leq i < \lvert\mathcal{S}\rvert \cdot (l-1)$
$[0, y]$	$0 \leq i < \lvert\mathcal{S}\rvert \cdot (l-1)$
$(y, y']$	$\lvert\mathcal{S}\rvert \cdot \lfloor \frac{y}{\Delta} \rfloor < i \leq \lvert\mathcal{S}\rvert \cdot (l-1)$
(y, ∞)	$i = l$

- **Restricting the state transition relation** The reward interval J must be further encoded into the state transition relation by means of the enabledness of the transitions. The enabledness of a transition depends now on the actual system state **and** on the reward level represented by the level bounds lb and ub, and may affect also the exit rates of states. We must distinguish the following situations for the specification of the restricted transition relation:

  1. In the simplest case we are dealing with homogeneity in the reward dimension ($J = [0, y)$) or consider a single time point ($I = [t, t]$). Homogeneity in the reward dimension allows to fire all transitions independently of the reward bound, and the absorbing states are derived solely from the time interval I. If the latter specifies a time point, the situation is similar, as only $(\neg\Phi)$-states are made absorbing. In this case the vector storing the exit rates has size $\lvert\mathcal{S}\rvert$.

  2. If $J \neq [0, y]$ and $I \neq [t, t]$ we have an inhomogeneous model, possibly in both dimensions. We have now to consider the lower reward bound when specifying the absorbing states. In this case the exit rate vector has the size of the implicit state space. Further, the inhomogeneity must be encoded into the transition relation as follows:

     a) all transition fire in $(\Phi \wedge \neg\Psi)$-states for all reward levels,

     b) reward induced transitions fire in $(\neg\Phi \wedge \Psi)$-states only until reaching the lower reward level bound, which is $\lfloor \frac{y}{\Delta} \rfloor$

c) all transitions fire in $(\Phi \wedge \Psi)$-states only until reaching the lower reward level bound, which is $\lfloor \frac{y}{\Delta} \rfloor$

d) $(\neg\Phi \wedge \neg\Psi)$-states are absorbing.

5.3 Summary

The analysis techniques for SPN, GSPN and SRN, which I consider in this thesis, are available in the CRSL model checker *MARCIE* . In this chapter I briefly described the most important implementation features of the numerical engine, which is based on the proposed on-the-fly generation of the involved matrices. The various solvers are realized on a generic design principle and are instantiated from a manageable number of so-called policies classes (see Table 5.1). Multi-threading is a core feature of *MARCIE's* numerical engine.

Table 5.1: An excerpt of *MARCIE's* numerical solvers as discussed in this thesis and their composition by instantiation of policy classes. There are more CSRL-specific solvers. [1)] indicates multi-threading support, [2)] indicates GSPN support

Analysis	MatrixProcessor	Im. Gathering	Gathering	Init	Application
$MatrixVector^{1),2)}$	FWD,ALL,RS	-	-	-	$\underline{diag}$, ϱ,$\mathbf{X}_J^I$
$FTransient^{1),2)}$	BWD,ALL,MV	-	GT	Unif	$\pi^C_{\alpha,\tau}$,$\iota^C_{\alpha,\tau}$
$BTransient^{1),2)}$	FWD,ALL,MV	-	GT	Unif	$\underline{\pi}_{\tau,S'}$
$FMTransient^{1)}$	BWD,MULTI,MV	-	GTM	UnifM	$v^{[C,\varrho]}_{\alpha,\tau,y}$
$BMTransient^{1)}$	FWD,MULTI,MV	-	GTM	UnifM	$\underline{v}_{\tau,t,S'}$
$JacobiHOM^{1),2)}$	FWD,ALL,MV	-	GJH	HLES	π^C_α
GSHOM	FWD,LINE,MV	GGSH	-	HLES	π^C_α
PGSHOM	FWD,BLOCK,MV	GGSH	-	HLES	π^C_α
$JacobiLIN^{1),2)}$	FWD,ALL,MVE	-	GJL	LES	$\underline{\pi}_{S'}$
GSLIN	FWD,LINE,MVE	GGSL	-	LES	$\underline{\pi}_{S'}$
PGSLIN	FWD,BLOCK,MVE	GGSL	-	LES	$\underline{\pi}_{S'}$

6 Evaluation

6.1 Methodology

In this chapter I present an experimental evaluation of the capabilities of the CSRL model checker *MARCIE* in terms of runtime and memory consumption. I consider the eight case studies given in Appendix A.2. To enable a judgement of the results, I compare them with figures obtained for the probabilistic model checker PRISM [78] in version 4.0, in particular with its hybrid engine.

One could compare with a couple of tools and technologies as it has been done in [66] or [67]. However, therein the comparison considers tools which are actually hard to compare, or the tools were employed with their default settings which often do not reflect their actual potential. I am interested in a meaningful comparison and restrict myself to PRISM, which I regard to be - in accordance with the literature - the state-of-the art model checker for Markov models. Further I require,

1. that the considered tools are built with same the compiler on the same hardware. The source code of PRISM is available under the GPL licence. *MARCIE* and PRISM 4.0 have been compiled with the GNU C++ compiler in version 4.1.2 with identical optimization settings (-O3).
2. that the used engines and analysis capabilities are comparable. Both tools are model checkers and support the Continuous Stochastic Logic. This eases to consider in the experiments the complete state space or to make a considerable set of states absorbing. Both tools compute complete probability distributions. A comparison with simulative/statistical or approximative engines is not meaningful. Both *MARCIE* and PRISM's hybrid engine realize the multiplication of a matrix and a vector as an early truncated DD traversal. The traversal depth can be specified by the user and represents the most important parameter affecting the tool's performance. The consideration of several truncation layers permits to relate the obtained figures to each other. Contrary, only fixed values were used in [67].
3. that the comparison is, as far as possible, independent of the model spec-

ification style. For all experiments I consider the same variable order for PRISM and *MARCIE*.

Nevertheless, the consideration of tools as SMART or Möbius, in particular their Matrix Diagram-based engines, would be very interesting. Because of the lac of model checking capabilities and especially the requirement to define structured models I do not consider them here. For the time being the figures presented in [67] could be used to appraise their capabilities in relation to PRISM and *MARCIE*.

Model specification. For the experiments I ignore the modularization feature of the PRISM language. In particular, all used PRISM models are generated by MARCIE[1] using the variable order heuristics mention in Section 4.3.3 applied to the model descriptions given in Appendix A.2. This approach is also motivated by our observations in [61], were we compared such generated PRISM model with the modularized version from [77]. Table 6.1 is taken from [61] and shows, how the used variable order affects the state space construction with PRISM.

Further, variables in PRISM need to be specified with an upper bound of the value range. The export permits to specify the actual value computed by *MARCIE*.

The FMS and the WC system as given in Appendix A.2.2 are GSPN models. For the experiments I considered in both cases the stochastically equivalent SPN.

The settings. All experiments were done on a MAC Pro 8×2.2 GHz with 32 GB RAM running Cent OS 5.5. For each experiment I measured the average runtime per iteration, the memory consumption peak, and either the total runtime, including model construction and initialization, or the total iteration runtime including the initialization time[2]. The time information where extracted from the tools‘ output. The memory consumption was obtained by a background shell-script which runs periodically the *ps*-command. Each experiment was automatically repeated for *MARCIE* and PRISM by a script with different truncation layers. The obtained figures are given in plots as shown in Figure 6.1. The x_1-axis is labeled with the truncation layers for the experiments made with *MARCIE*. The x_2-axis shows the truncation layers for the PRISM experiments. In PRISM, the number of truncation layers corresponds to the number of variables required to represent the binary encoding of the reachable states. A comparison of the

[1] *MARCIE's* PRISM export is an undocumented feature, available on request.
[2] When explicitly limiting the number of iterations, the tools do not not output the total runtime.

Table 6.1: Comparison of two variable orders. The table shows the time and the number of MTBDD nodes, which PRISM needs to construct the rate matrix of the CTMC for a good variable order,computed using Equation 4.2, and for the plain order of the original PRISM model, specified according to [77].

levels	terminal	good order		original order	
	nodes[a)]	time	nodes	time	nodes
4	30	0.12	8,672	2.47	123,730
8	76	1.56	60,452	401.68	3,881,914
12	140	22.99	199,496	-	-
16	219	71.25	542,339	-	-
20	320	296.87	953,146	-	-
24	453	635.92	2,029,598	-	-
28	697	928.45	3,771,617	-	-
32	770	1847.60	6,015,521	-	-

[a)] *i.e., number of different entries in rate matrix; '– exceeds the available memory;*

range of the x-axes shows how PRISM expands the set of model variables (and the set of variables has implicitly to be doubled due to encoding of state transitions). The numerical computation were triggered by model-specific CS(R)L formulas which may not necessarily be meaningful from a modeler‘s perspective. I provide all scripts, models, property and result files as supplementary material[3]. *MARCIE* is documented [106] and available in terms of statically linked binaries for Linux and MAC/OS. All experiments can be reproduced.

Please note that the presented results are only samples. The same experimental setting with a slight modification of the considered variable order in the model specification may result in different runtime and memory consumption. But a comparison of the tools, which considers the models with different initial states and different variable orders exceeds by far any reasonable size of a thesis.

6.2 Transient Analysis

In this section I evaluate the capabilities of PRISM and *MARCIE* for transient analysis and utilize the backward transient analysis reported in [68]. The exper-

[3] available on request

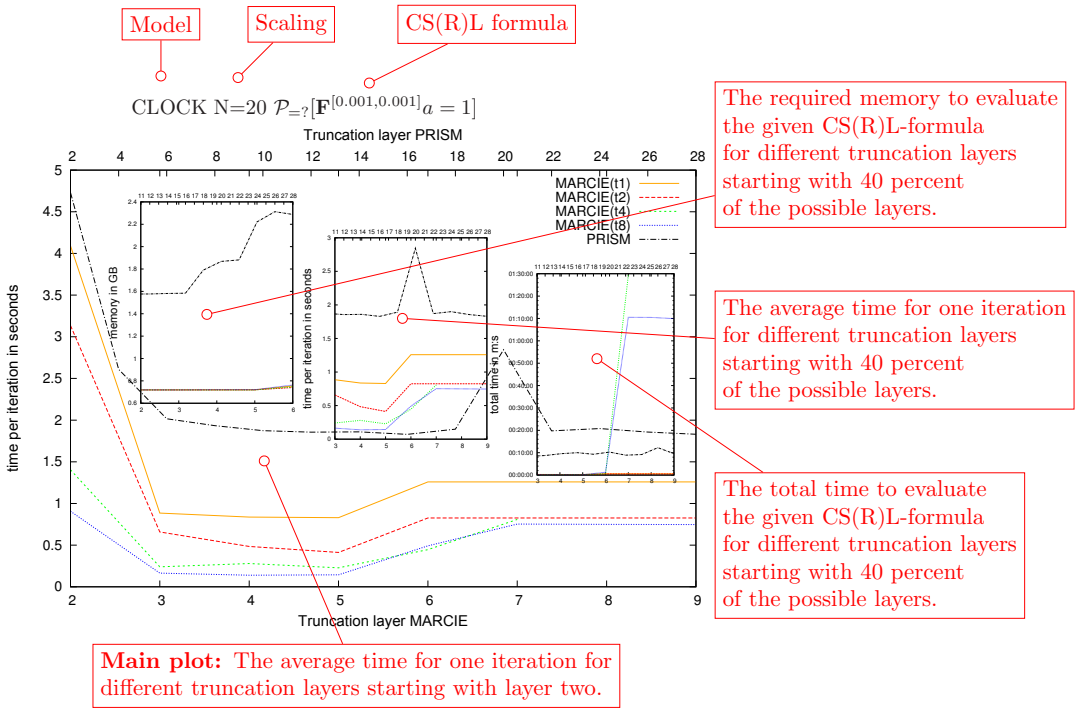


Figure 6.1: An example plot.

iments are triggered with the CSL template

$$\mathcal{P}_{=?}[\mathbf{F}^{I}\phi],$$

where I and ϕ are defined in a model-specific way. The time interval I is generally chosen to achieve a small number of iterations. Its shape enables further to make the ϕ-states absorbing. As *MARCIE* offers multi-threaded transient analysis, I run the experiments also with two, four and eight threads.

No absorbing states. To consider the complete Markov chain, experiments are triggered with the CSL-template

$$\mathcal{P}_{=?}[\mathbf{F}^{[\tau,\tau]}\phi].$$

The settings are summarized in Table 6.2.

Table 6.2: Overview of experiments $\mathcal{P}_{=?}[\mathbf{F}^{[\tau,\tau]}\phi]$.

Model	N	#	ϕ	τ	it	plot
AKAP	4	1993	$cAMP < 2$	0.001	11	6.2
CLOCK	20	209	$a = 1$	0.001	15	6.3
ERK	30	1522	$ERK = 0$	0.000625	10	6.3
MAPK	10	53	$kpp + kkpp = 12$	0.0005	13	6.4
LEV	10	127	$kpp + kkpp = 12$	0.01	14	6.4
FMS	10	74	$P1 = 0$	0.1	16	6.5
KANBAN	8	14	$x1 = 1$	0.1	9	6.5
PSS	20	4	$s1 = 1 \wedge \neg(s = 1 \wedge a = 1)$	0.01	13	6.6
WC	256	267	$Down\_4 = 1$	1.0	16	6.6

'#' is the number of terminal nodes in the MTBDD representation. 'it' is the number of iterations.

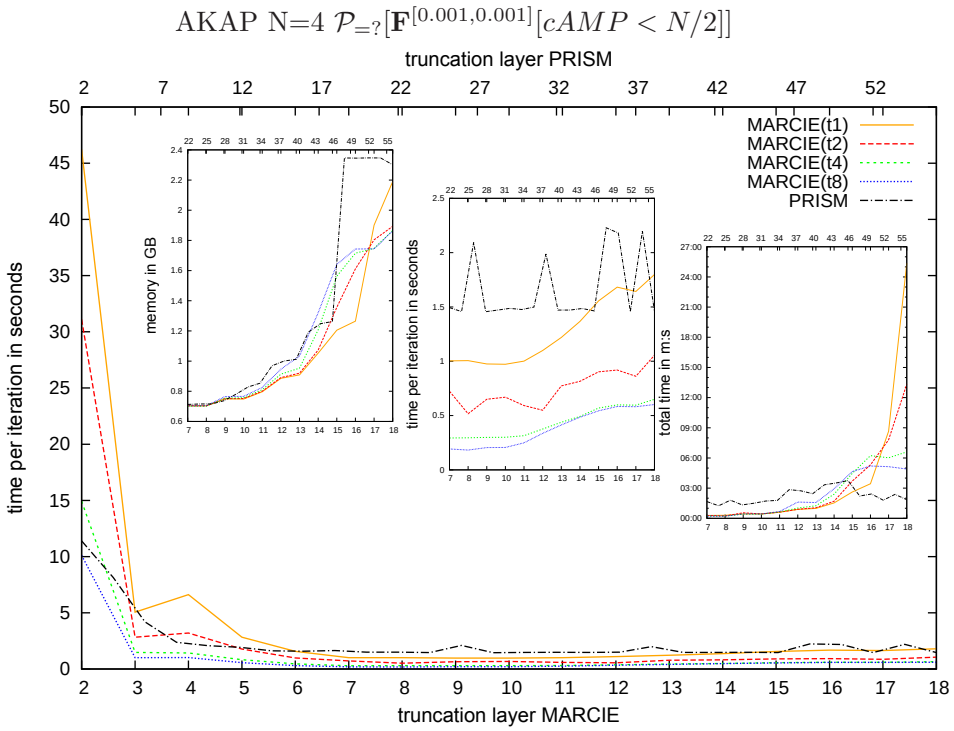


Figure 6.2: $\mathcal{P}_{=?}[\mathbf{F}^{[\tau,\tau]}\phi]$ – AKAP

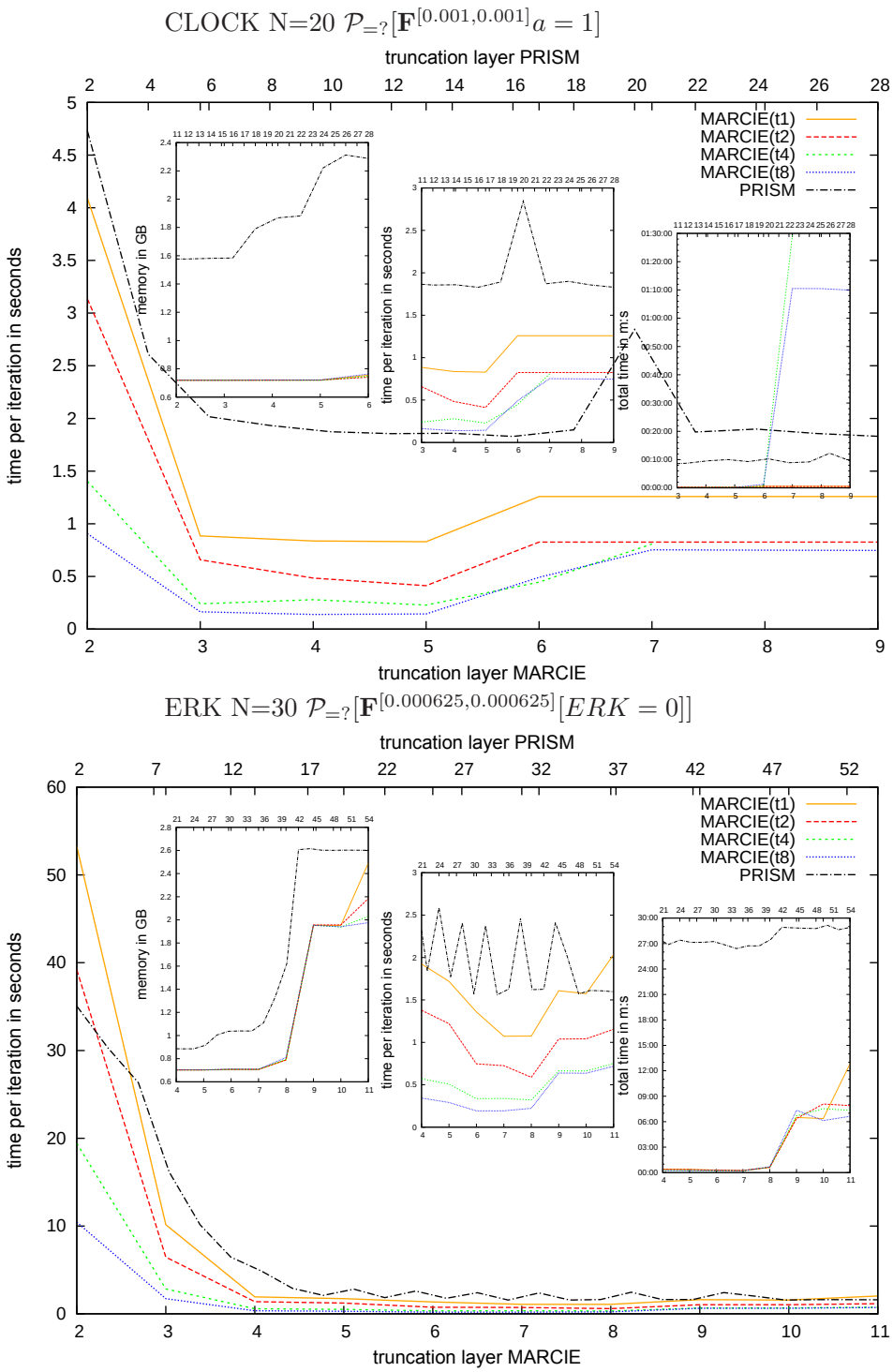
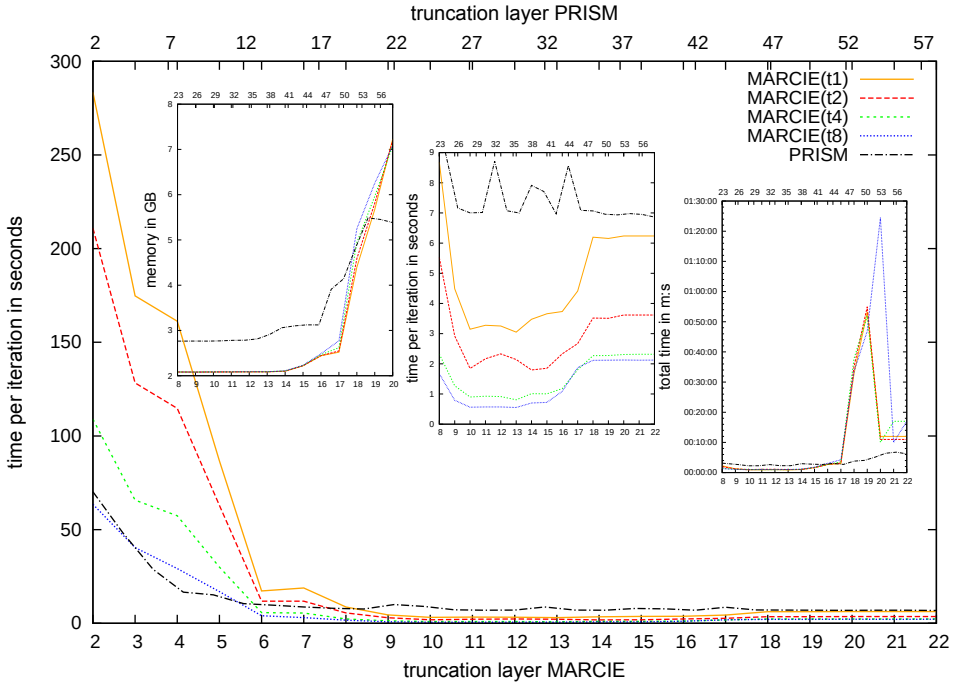


Figure 6.3: $\mathcal{P}_{=?}[\mathbf{F}^{[\tau,\tau]}\phi]$ – CLOCK and ERK

MAPK N=10 $\mathcal{P}_{=?}[\mathbf{F}^{[0.0005,0.0005]}[kpp + kkpp < N/2]]$



LEV N=10 $\mathcal{P}_{=?}[\mathbf{F}^{[0.01,0.01]}kpp + kkpp = 12]$

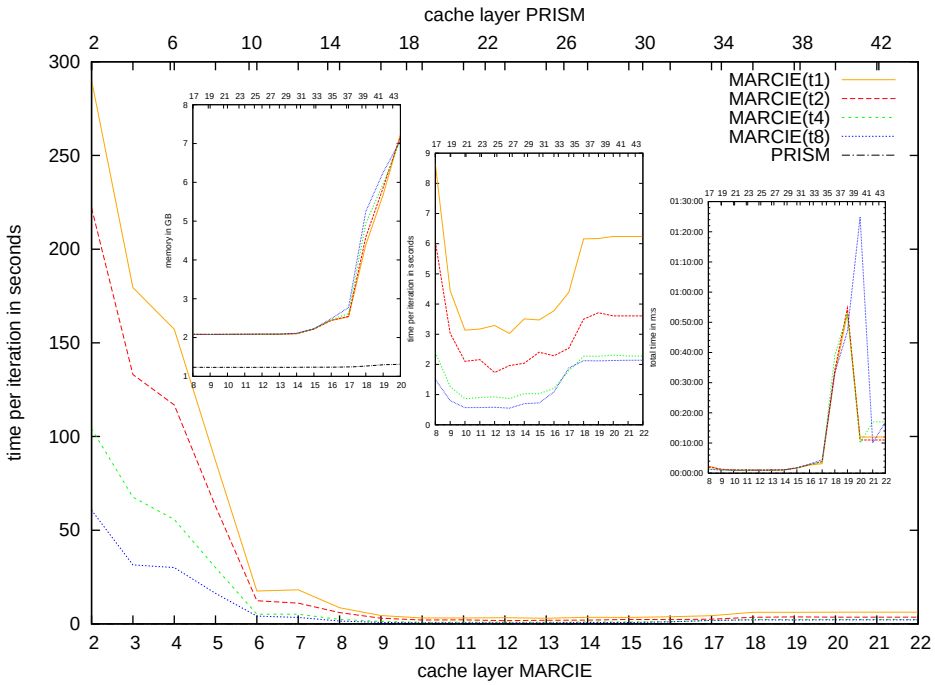


Figure 6.4: $\mathcal{P}_{=?}[\mathbf{F}^{[\tau,\tau]}\phi]$ – MAPK and LEV

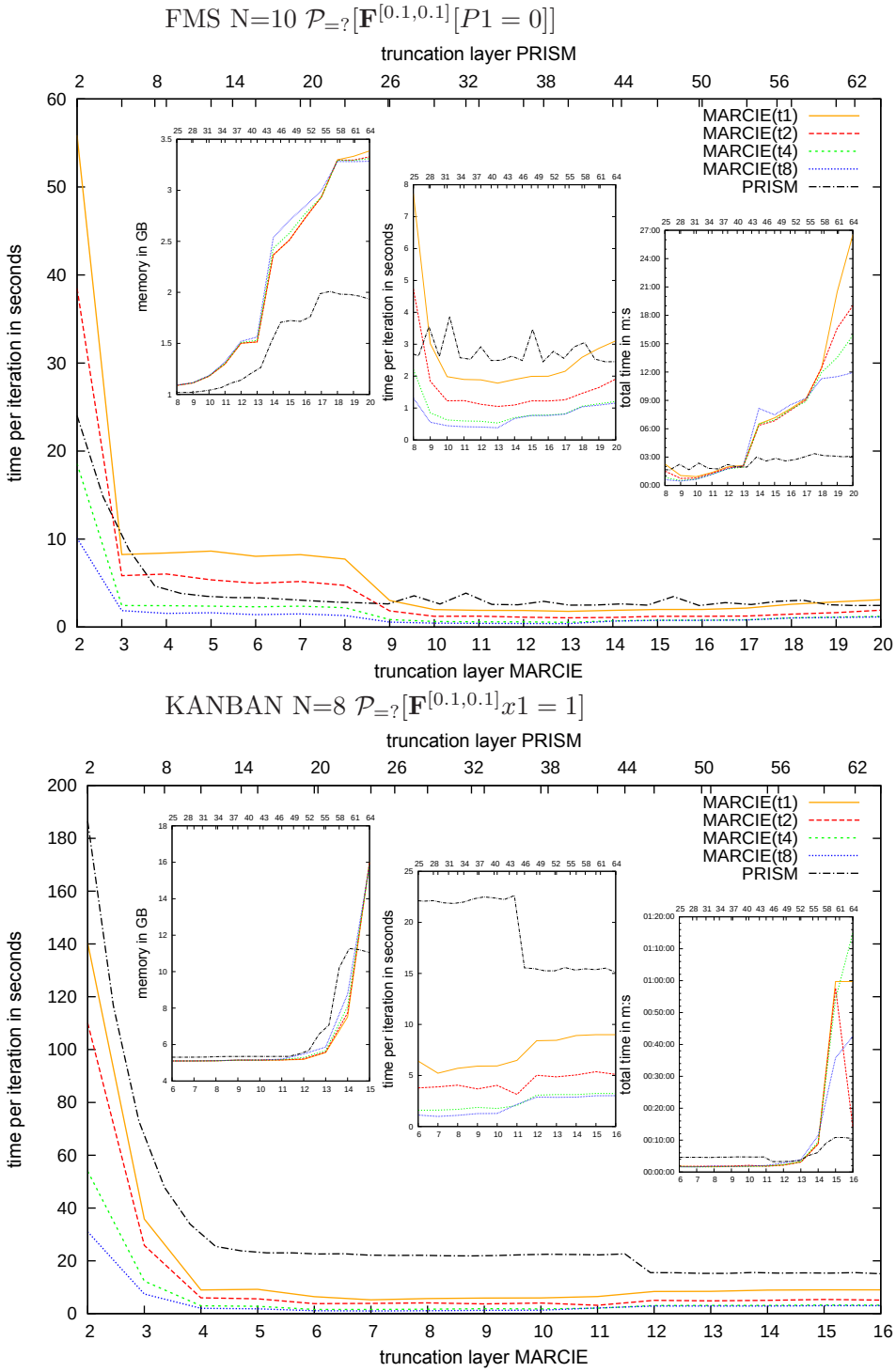
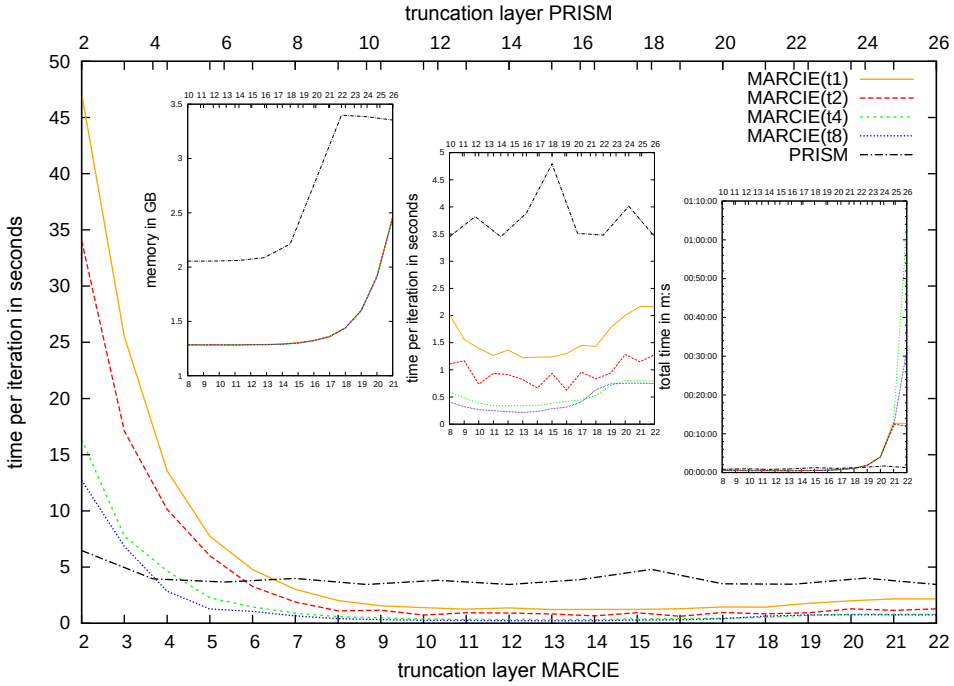


Figure 6.5: $\mathcal{P}_{=?}[\mathbf{F}^{[\tau,\tau]}\phi]$ – FMS and KANBAN

PSS N=20 $\mathcal{P}_{=?}[\mathbf{F}^{[0.01,0.01]}s1 = 1 \wedge \neg[s = 1 \wedge a = 1]]$



WC N=256 $\mathcal{P}_{=?}[\mathbf{F}^{[1,1]}[Down\_4 = 1]]$

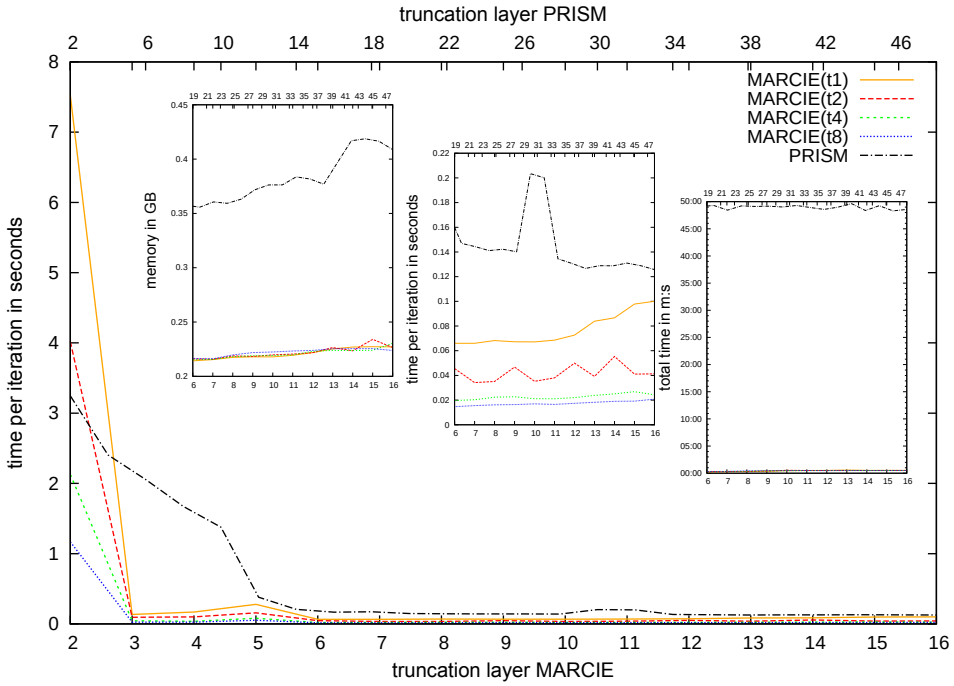


Figure 6.6: $\mathcal{P}_{=?}[\mathbf{F}^{[\tau,\tau]}\phi]$ – PSS and WC

With absorbing states. To make certain states absorbing, I use the CSL template

$$\mathcal{P}_{=?}[\mathbf{F}^{[0,\tau]}\phi].$$

The settings are summarized in Table 6.3. As all ϕ-states become absorbing, the choice of the sub-formula ϕ affects the size of the resulting Markov chain. The column % in Table 6.3 displays the percentage of the original number of states.

Table 6.3: Overview of experiments $\mathcal{P}_{=?}[\mathbf{F}^{[0,\tau]}\phi]$.

Model	N	#	ϕ	%	τ	it	plot
AKAP	4	1993	$cAMP < N/2$	95.0	0.001	11	6.7
CLOCK	20	209	$a < N/2$	52.5	0.001	15	6.8
ERK	30	1522	$ERK < N/2$	20.2	0.000625	10	6.8
MAPK	10	53	$kpp + kkpp = N/2$	40.2	0.0005	13	6.9
LEV	10	127	$kpp + kkpp < N/2$	40.2	0.01	14	6.9
FMS	10	74	$P1 < N/2$	12.5	0.1	16	6.10
KANBAN	8	14	$x1 < N/2$	21.2	0.1	9	6.10
PSS	20	4	$s1 = 1 \wedge \neg(s = 1 \wedge a = 1)$	50.8	0.01	13	6.11
WC	256	267	$Down\_4 = 0$	42.8	1	16	6.11

'%' is the fraction of non-absorbing states with regard to $\mathcal{S}$. '#' is the number of terminal nodes in the MTBDD representation. 'it' is the number of iterations.

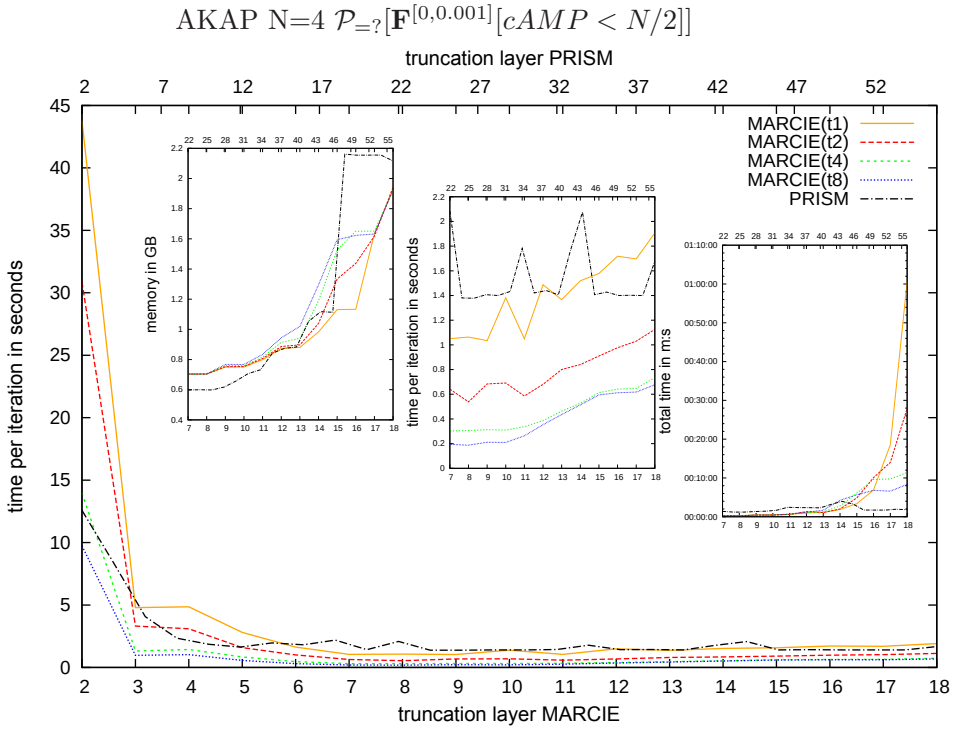


Figure 6.7: $\mathcal{P}_{=?}[\mathbf{F}^{[0,\tau]}\phi]$ – AKAP

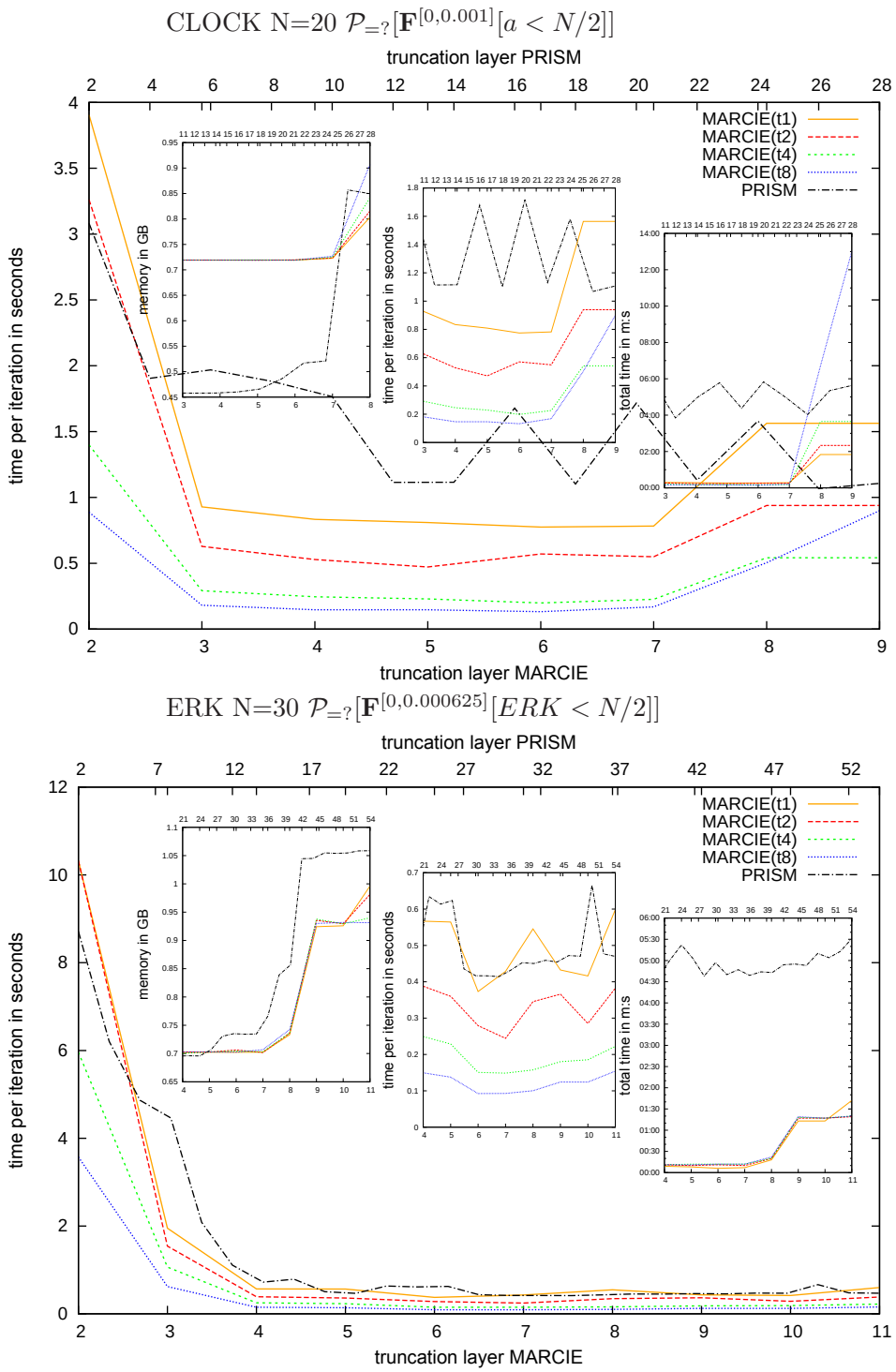
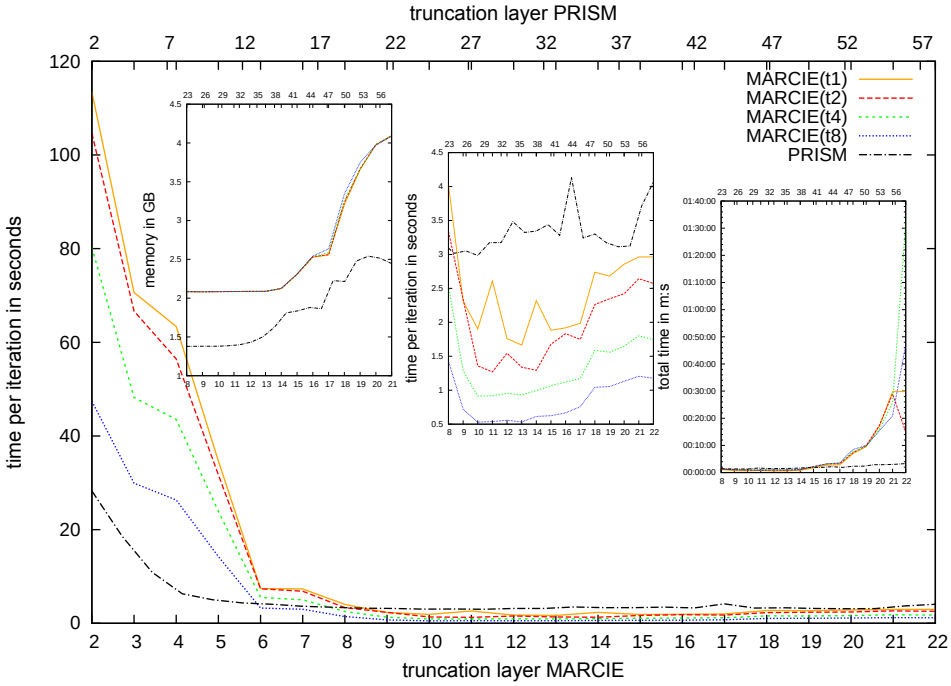


Figure 6.8: $\mathcal{P}_{=?}[\mathbf{F}^{[0,\tau]}\phi]$ – CLOCK and ERK

MAPK N=10 $\mathcal{P}_{=?}[\mathbf{F}^{[0,0.0005]}[kpp + kkpp < N/2]]$



LEV N=10 $\mathcal{P}_{=?}[\mathbf{F}^{[0,0.01]}[kpp + kkpp < N/2]]$

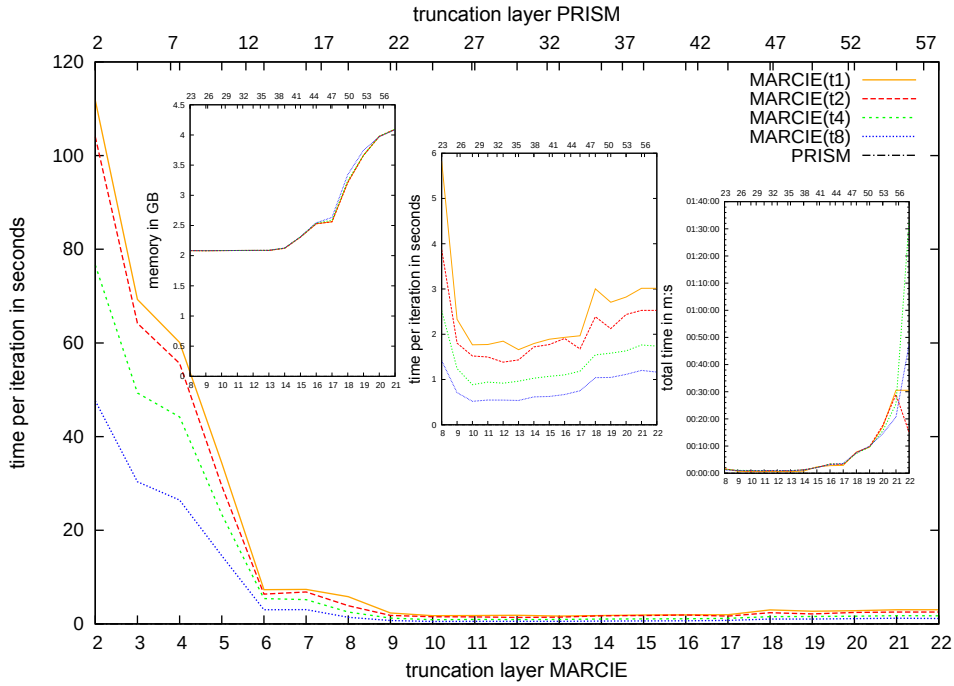


Figure 6.9: $\mathcal{P}_{=?}[\mathbf{F}^{[0,\tau]}\phi]$ – MAPK and LEV

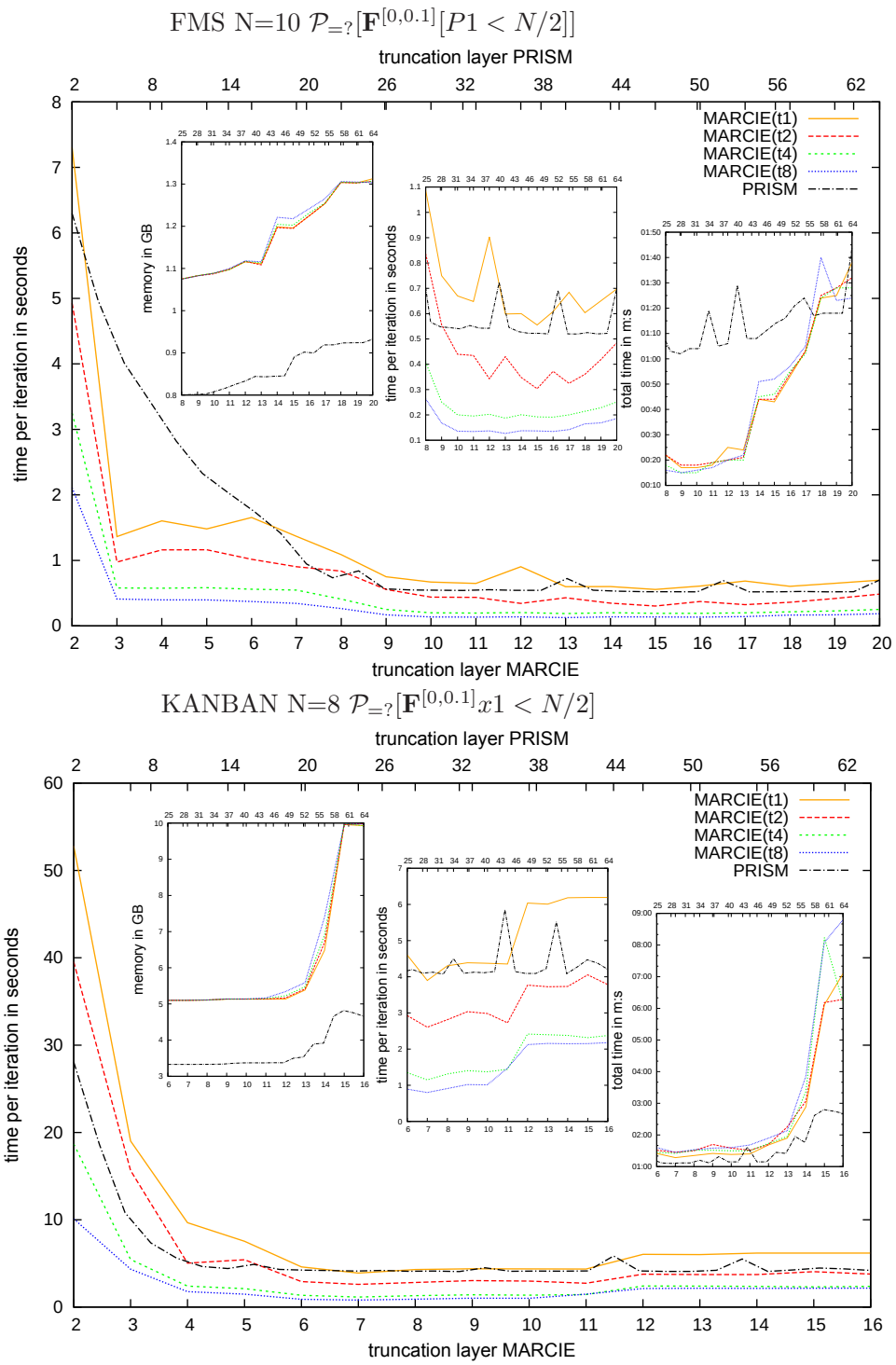
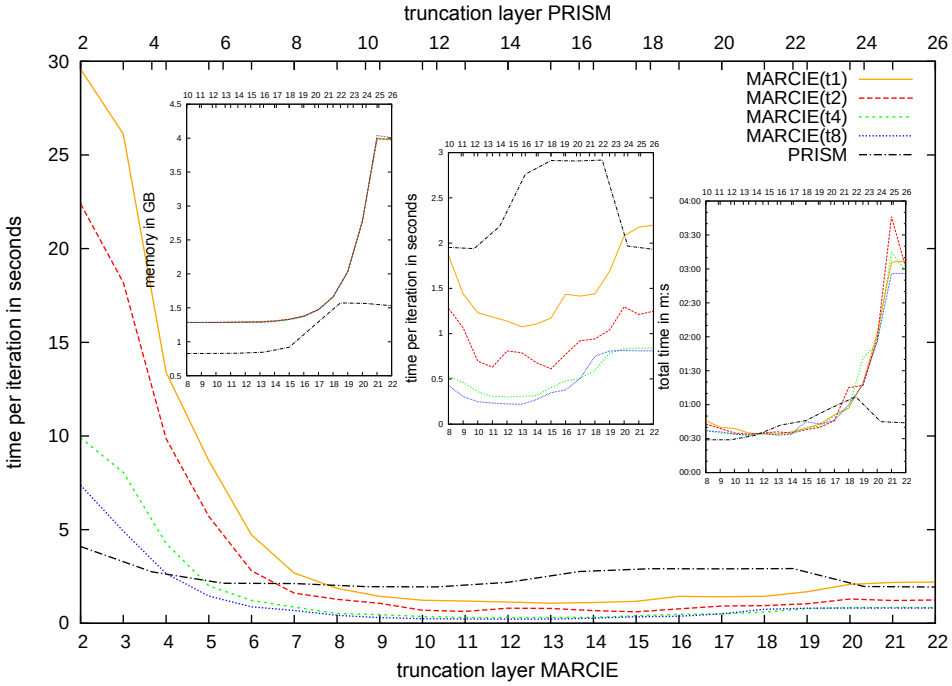


Figure 6.10: $\mathcal{P}_{=?}[\mathbf{F}^{[0,\tau]}\phi]$ – FMS and KANBAN

PSS N=20 $\mathcal{P}_{=?}[\mathbf{F}^{[0,0.01]} s1 = 1 \wedge \neg[s = 1 \wedge a = 1]]$



WC N=256 $\mathcal{P}_{=?}[\mathbf{F}^{[0,1]}[Down\_4 = 0]]$

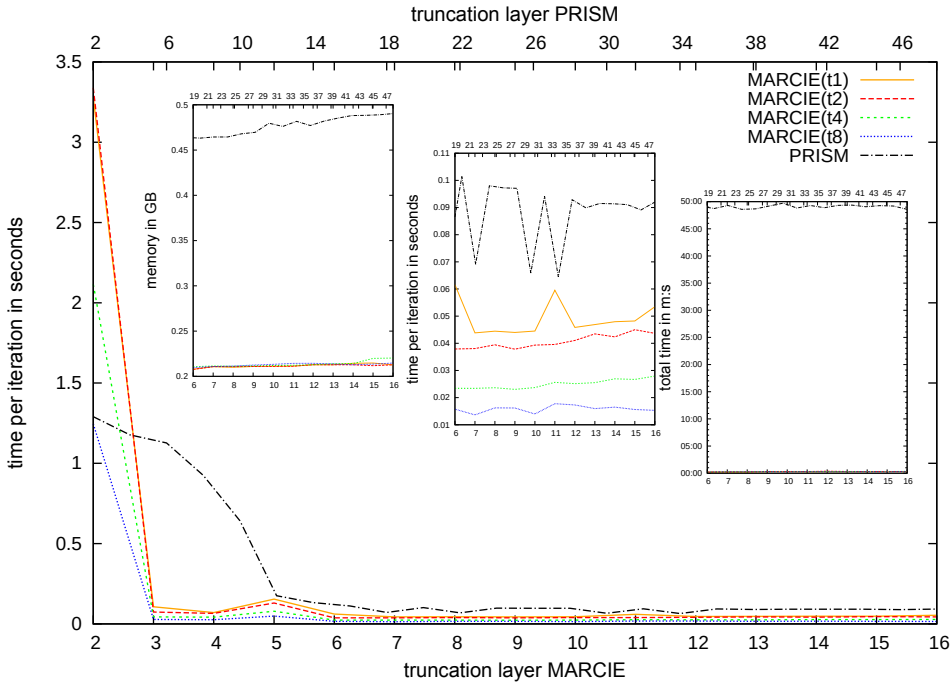


Figure 6.11: $\mathcal{P}_{=?}[\mathbf{F}^{[0,\tau]}\phi]$ – PSS and WC

Discussion. The evaluation of the obtained figures for transient analysis bares the following observations. With regard to the average runtime per iteration we observe that both approaches have an unacceptable runtime if the truncation layer is near the terminal nodes. There is a considerable improvement starting with layer three. For *MARCIE*, the initialization effort explodes if the truncation layer is set near to the root. In general it seems to be advisable to set the layer between 40 and 70 percent of the maximal value. Within this interesting range, *MARCIE* is in almost all cases much faster than PRISM if the complete Markov chain has to be considered. The speedup factor lies often between two and four (KANBAN, PSS, WC). If only a subset of the states is considered, the differences become moderate. In any case, the *MARCIE* is at least on a par. Multi-threading decreases significantly the runtime in all cases without an influence on memory consumption. But the achieved speedup is not linear to the number of threads. It stands out that for truncation layers near the root layer, *MARCIE's* initialization effort is not acceptable. However, for the interesting range, the initialization can be neglected for all considered models. In contrast, PRISM's generation of the sparse matrices may represent a bottleneck independently of the chosen layer. This effect is model-dependent and was observed for ERK, LEV and WC. For LEV I did not obtain results within the specified time limit of three hours.

There is no clear picture for the memory consumption. For some models, *MARCIE* has a significant higher consumption than PRISM and vice versa. Furthermore, PRISM implements some ideas as indexing[4]. It could also be introduced in *MARCIE* .

PRISM's memory consumption is highly affected by the fraction of absorbing states, compare Figure 6.3 and Figure 6.8.

[4]The exit rate vector stores unsigned indices referring to entries in a separate array which contains the actual double precision values. This optimization can by applied if there are only few different exit rates.

6.3 Steady State Analysis

In this section I investigate the potential of PRISM and *MARCIE* for steady state analysis with the Jacobi, Gauss-Seidel and the Pseudo-Gauss-Seidel methods. The experiments are triggered with the CSL template

$$\mathcal{S}_{=?}[\phi],$$

where ϕ is some model-specific proposition. The choice of ϕ does not affect the size of the considered state space. We are dealing with the complete Markov chain. In most cases the solution of the problem would take hundreds of iterations. In some cases the method of Jacobi does not converge within 10000 iterations[5]. Thus I limited all experiments to 15 iterations. PRISM as well as *MARCIE* offer a related option. In this case the tools do not output the total time. Thus the right inner plot in the diagrams shows now the total iteration time, which represents the actual solution time and the time for initialization. It does not contain the time for state space construction and the time to determine the BSCCs. For reasons of completeness, the settings are summarized in Table 6.4. At this point I do not consider multi-threading, although *MARCIE's* Jacobi (JAC) solver supports this feature. I present related figures in Section 6.4. The most important point is the comparison of the Jacobi and the Pseudo-Gauss-Seidel (PGS) method. *MARCIE's* current implementation of the Gauss-Seidel (GS) method can not compete with that of PRISM, which is indeed based on techniques which are no genuine BDDs anymore [88]. The related figures just highlight that there is demand for improvements.

[5] The default limit in PRISM and *MARCIE*.

Table 6.4: Overview of experiments $\mathcal{S}_{=?}[\phi]$.

Model	N	ϕ	plot
AKAP	4	$cAMP < N/2$	6.12
CLOCK	20	$a < N/2$	6.13
ERK	30	$ERK < N/2$	6.13
MAPK	10	$kpp + kkpp < N/2$	6.14
LEV	10	$kpp + kkpp < N/2$	6.14
FMS	10	$P1 < N/2$	6.15
KANBAN	8	$x1 < N/2$	6.15
PSS	20	$s1 = 1 \wedge \neg(s = 1 \wedge a = 1)$	6.16
WC	256	$Down\_4 = 0$	6.16

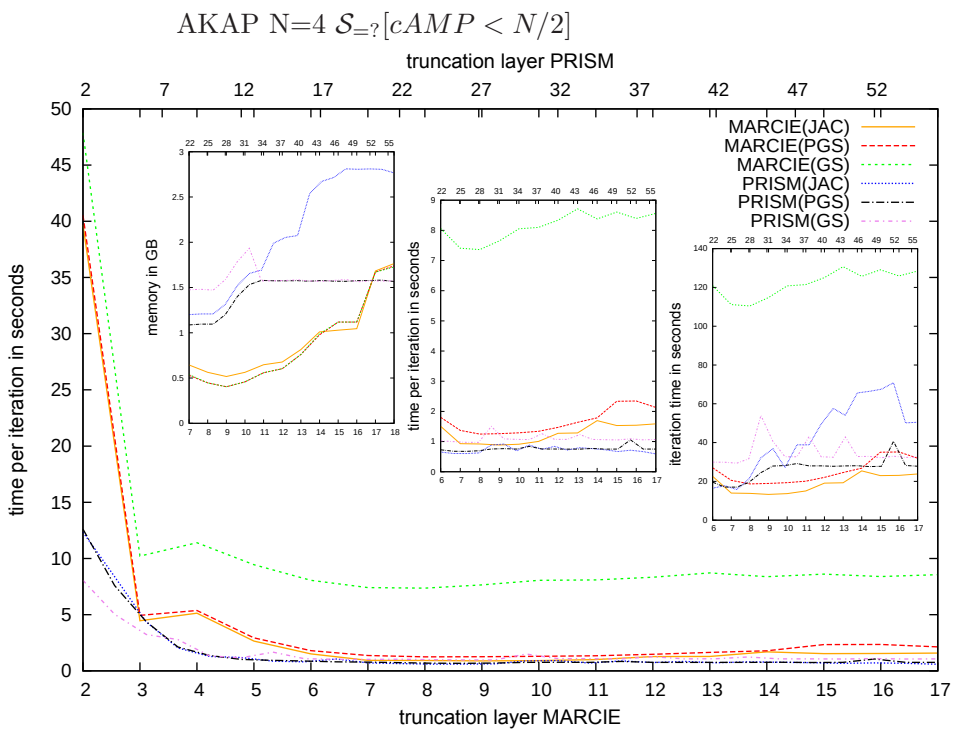


Figure 6.12: $\mathcal{S}_{=?}[\phi]$ – AKAP

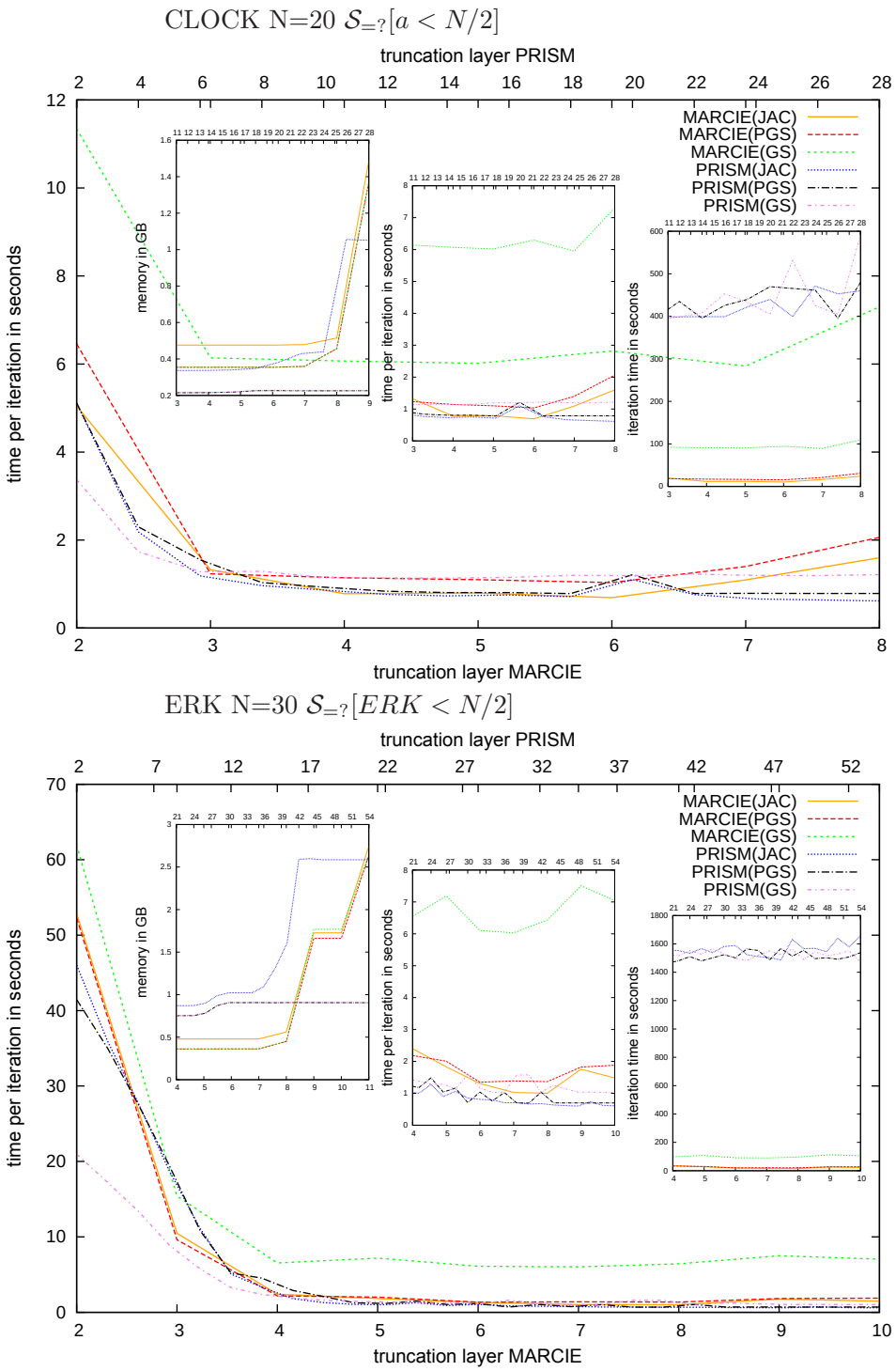


Figure 6.13: $\mathcal{S}_{=?}[\phi]$ – CLOCK and ERK

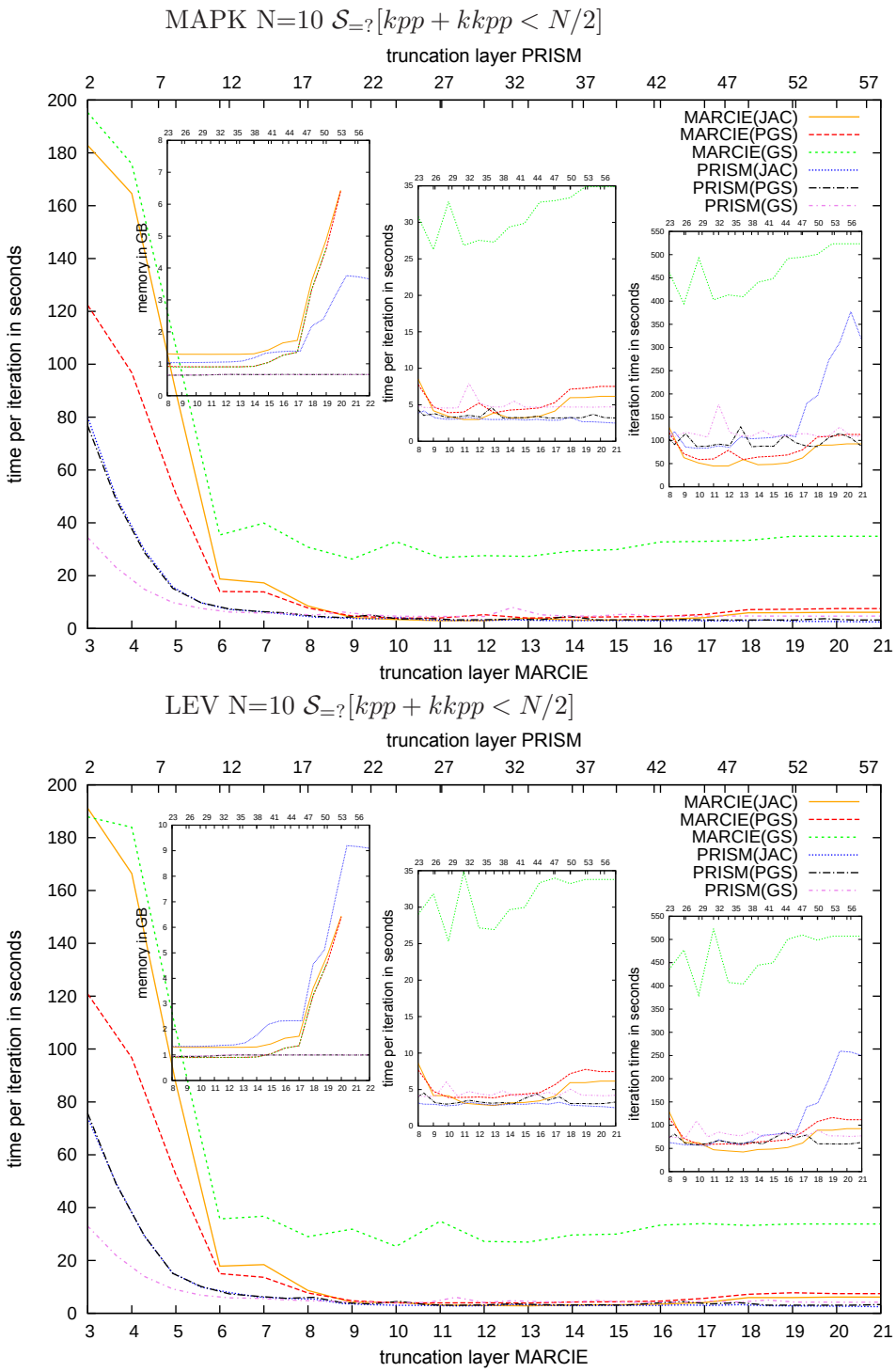


Figure 6.14: $\mathcal{S}_{=?}[\phi]$ – MAPK and LEV

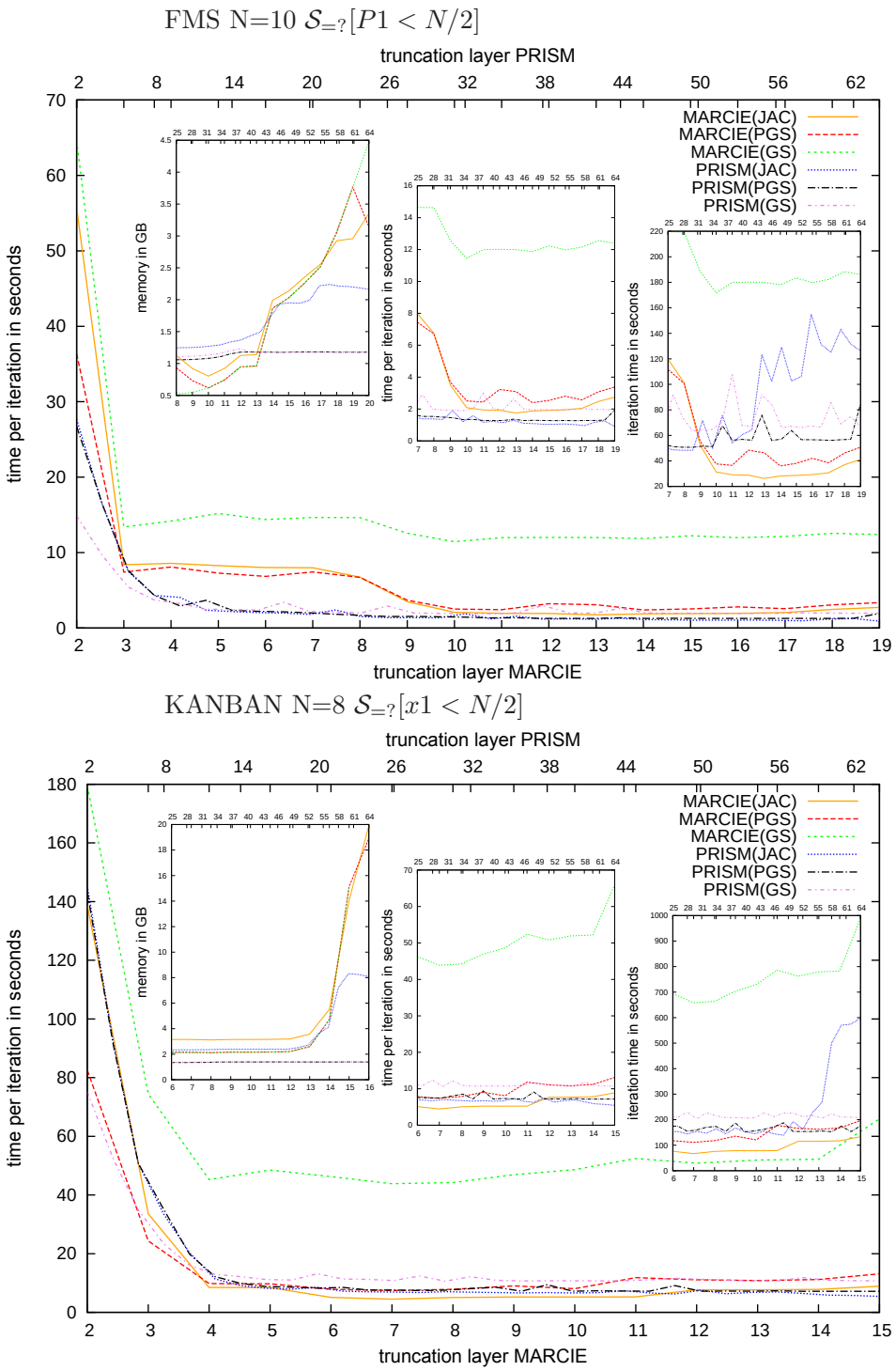


Figure 6.15: $\mathcal{S}_{=?}[\phi]$ – FMS and KANBAN

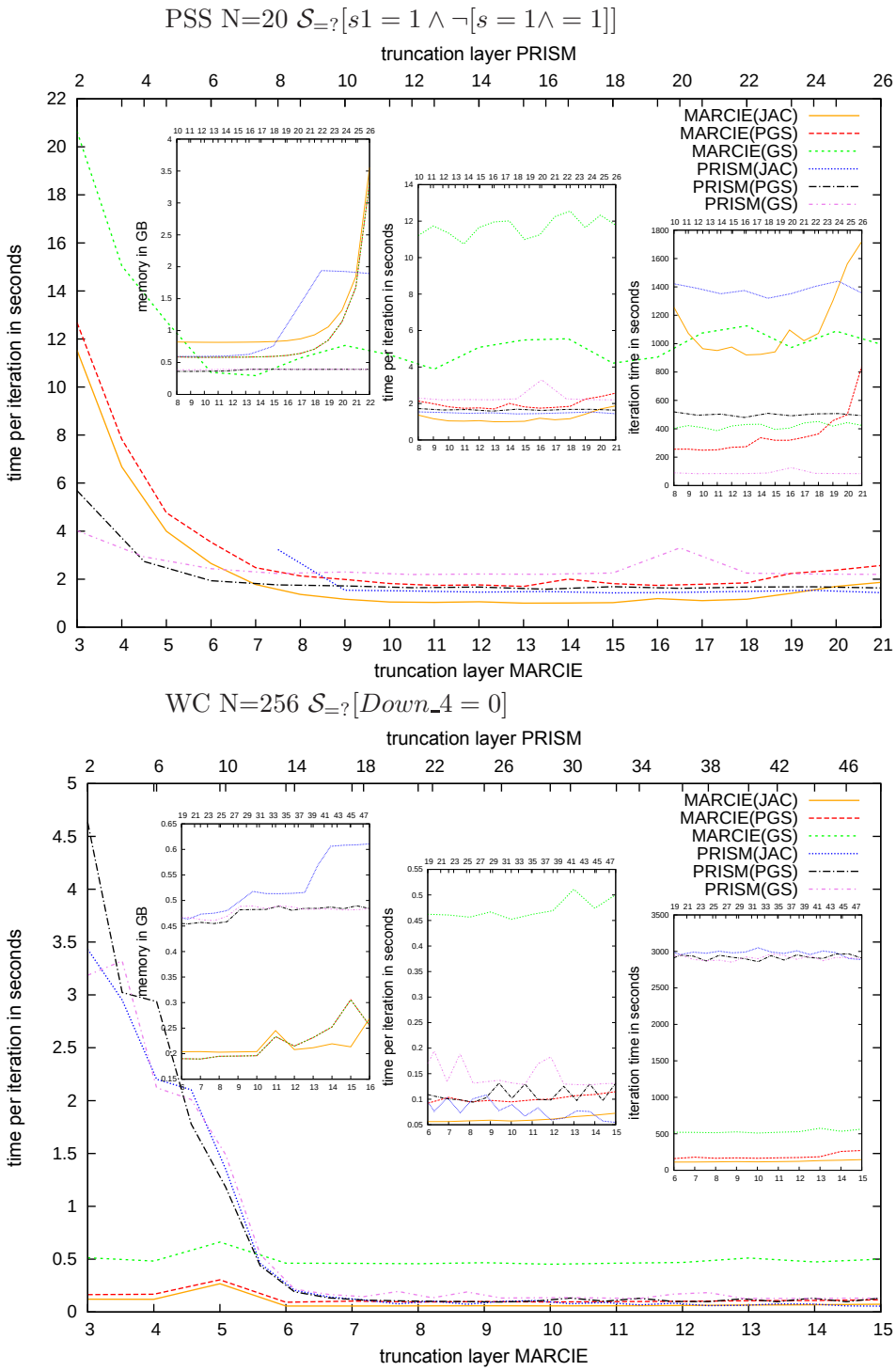


Figure 6.16: $\mathcal{S}_{=?}[\phi]$ – PSS and WC

Discussion. For the considered models (for the chosen state space sizes) PRISM's PGS solver is generally faster. Concerning the method of Jacobi, the solvers deliver comparable runtime and memory consumption. Significant differences as for transient analysis are not observable (with exception of the GS solvers). On the whole I would say that both tools play at the same league. Steady state analysis is not *MARCIE's* speciality. However, it should be noted that the employment of multi-threading decreases significantly the runtimes as it is illustrated with the KANBAN model in Figure 6.17.

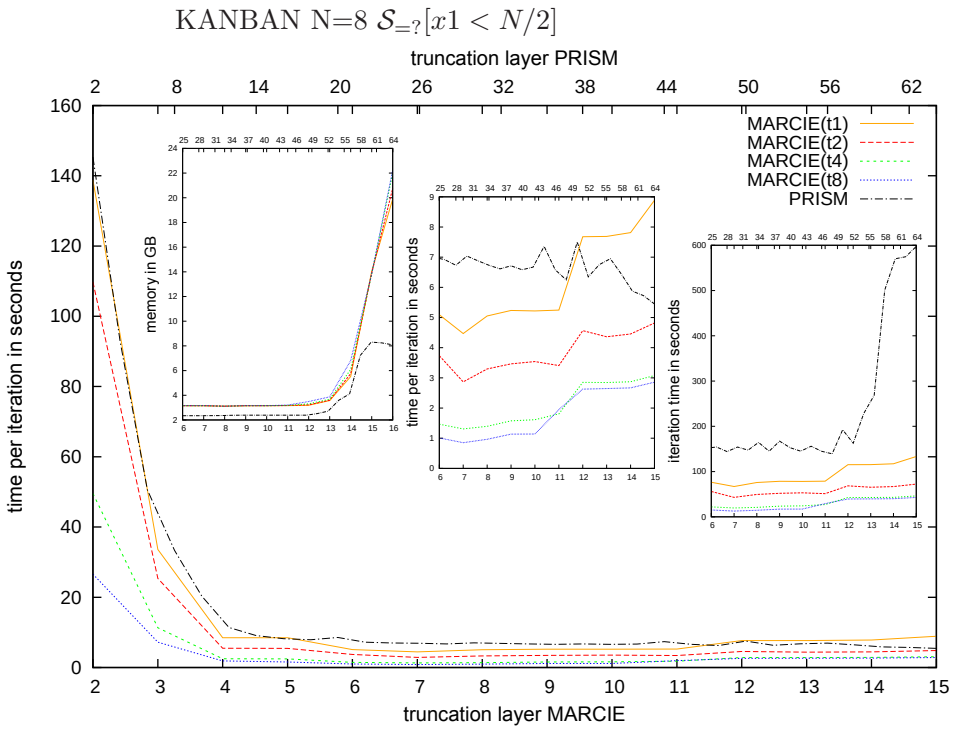


Figure 6.17: $\mathcal{S}_{=?}[\phi]$ – KANBAN multi-threaded Jacobi

6.4 Embedded Markov Chain

The third experiment type is used to judge the analysis capabilities with regard to the embedded Markov chain. The main application of such an analysis is to determine the probability to reach a set of states. The PRISM model checker supports several reward-related operators extending CSL [76] which enable for

instance to specify the following formula template

$$\mathcal{R}\{\varrho\}_{=?}[\mathbf{F}\phi].$$

It asks for the expected accumulated reward until reaching one of the ϕ-states. If the reward structure ϱ represents the sojourn time, we can determine the expected time until reaching such a state[6]. The numerical computation behind the scenes considers the embedded Markov chain. I deploy this formula pattern to trigger the following multi-threaded experiments using the method of Jacobi. Table 6.5 gives the model-specific details, as for instance the number of iterations (in this case not limited) or the number of the MTBDD terminal nodes, which often inhibit the generation of PRISM's internal data structures.

Table 6.5: Overview of experiments $\mathcal{R}\{\varrho\}_{=?}[\mathbf{F}\,\phi\,]$.

Model	N	#	$\#_E$	ϕ	it	plot
AKAP	4	1,978	1,572,694	$cAMP < N/2$	127	6.18
CLOCK	20	-	-	$a < N/2$	15	6.19
ERK	30	-	-	$ERK < N/2$	15	6.19
MAPK	10	-	-	$kpp + kkpp < N/2$	15	6.20
LEV	10	-	-	$kpp + kkpp < N/2$	15	6.20
FMS	10	74	16,507	$P1 < N/2$	32	6.21
KANBAN	8	14	949	$x1 < N/2$	180	6.21
PSS	20	44	81	$s1 = 1 \wedge \neg(s = 1 \wedge a = 1)$	15	6.22
WC	128	141	222,4954	$Down\_4 = 0$	43	6.22

'#' is the number of terminal nodes in the MTBDD representation. '$\#_E$' is the number of terminal nodes in the MTBDD representation (embedded Markov chain). '–' PRISM did not finished the initialization because it exceeded the time limit or terminated with a memory error. 'it' is the number of iterations.

[6] *MARCIE* specifies implicitly a reward structure *time* for this purpose.

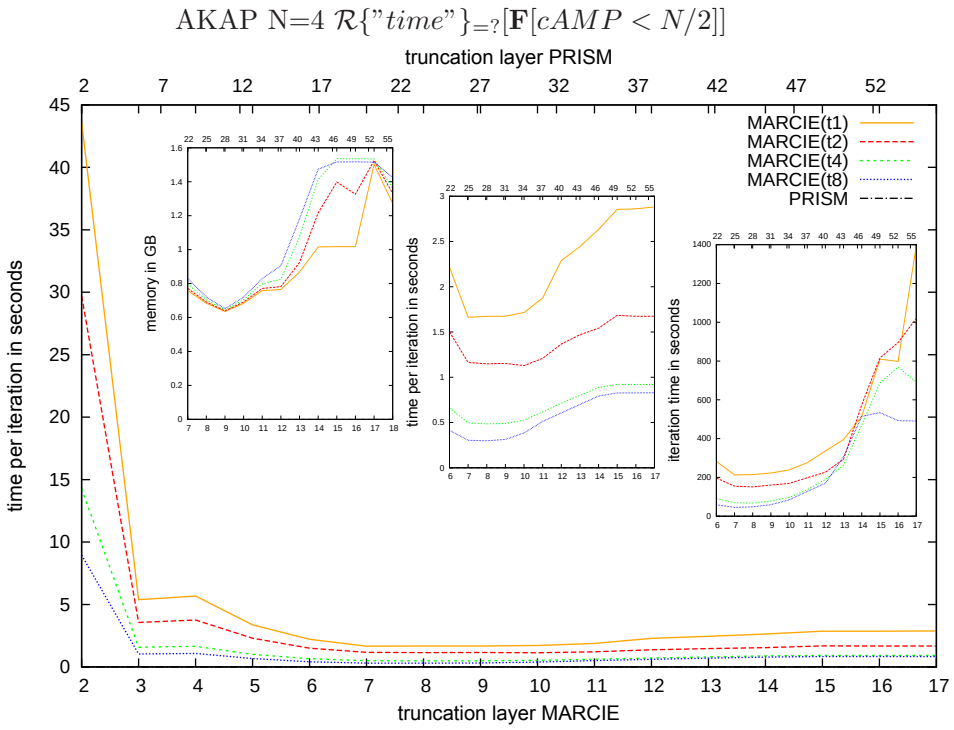


Figure 6.18: $\mathcal{R}\{\varrho\}_{=?}[\mathbf{F}\,\phi\,]$ – AKAP

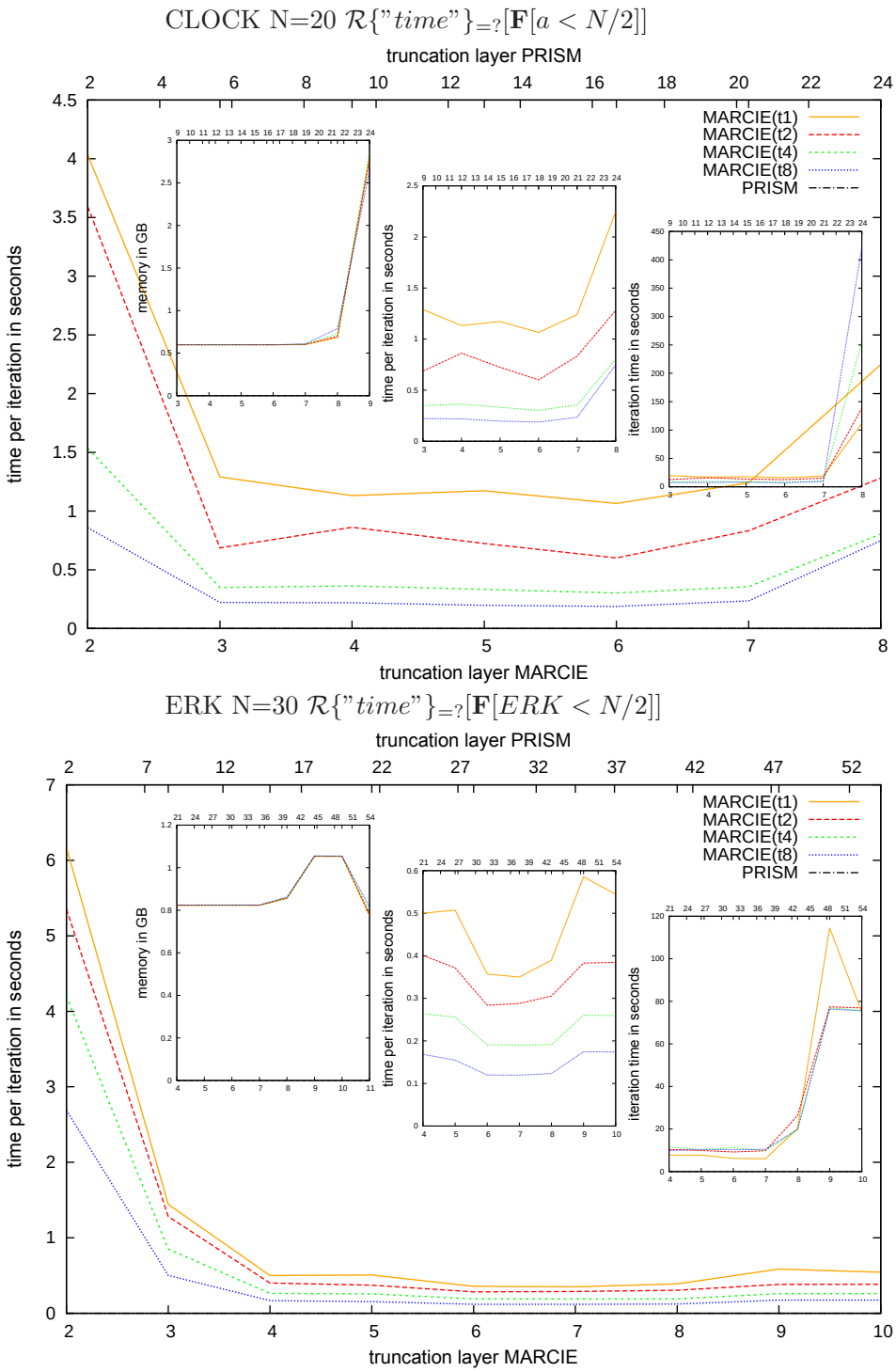


Figure 6.19: $\mathcal{R}\{\varrho\}_{=?}[\mathbf{F}\,\phi\,]$ – CLOCK and ERK

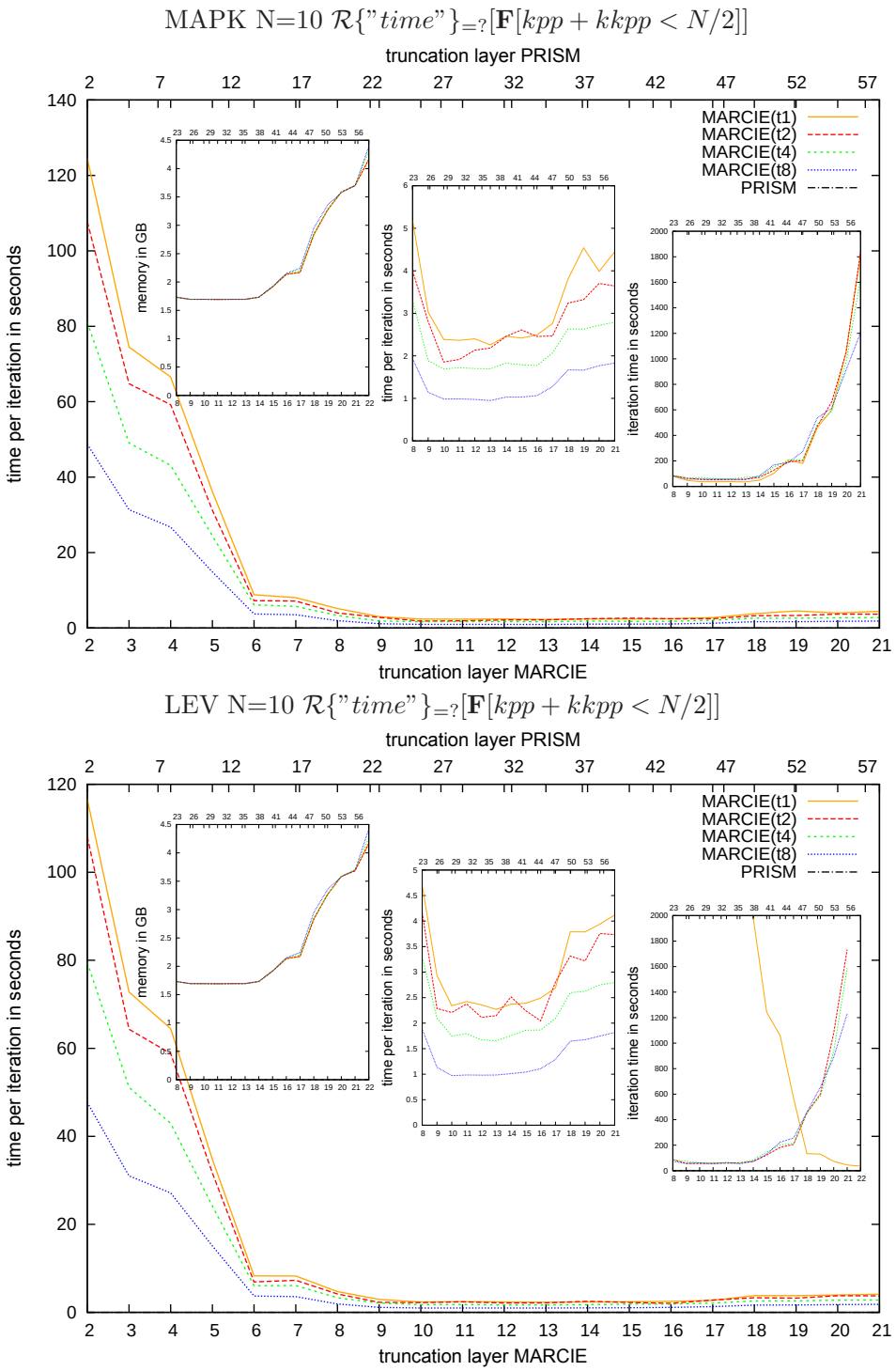


Figure 6.20: $\mathcal{R}\{\varrho\}_{=?}[\mathbf{F}\,\phi\,]$ – MAPK and LEV

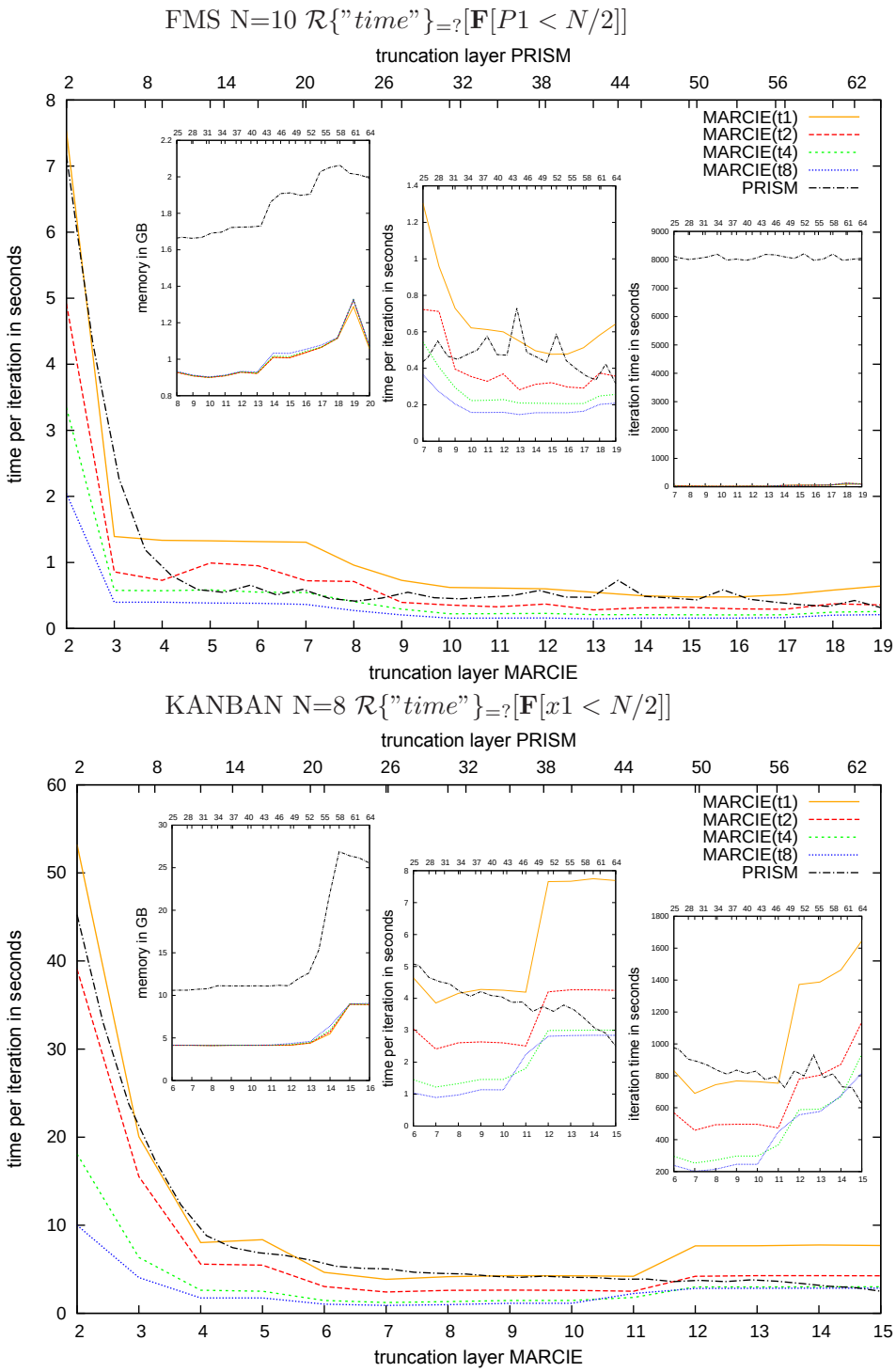
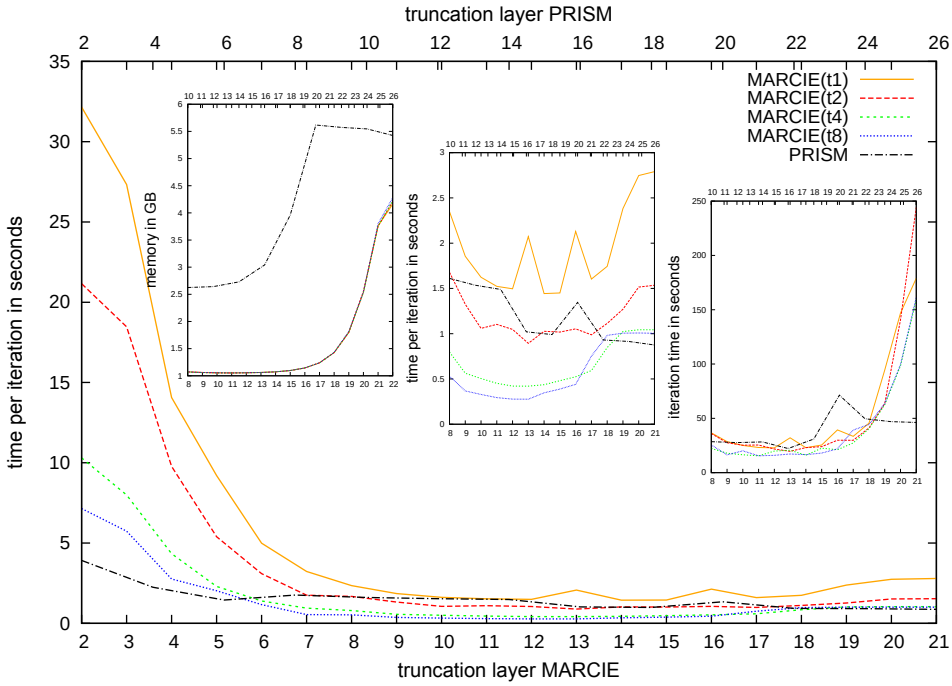


Figure 6.21: $\mathcal{R}\{\varrho\}_{=?}[\mathbf{F}\,\phi\,]$ – FSM and KANBAN

PSS N=20 $\mathcal{R}\{\text{"}time\text{"}\}_{=?}[\mathbf{F}[s1 = 1 \wedge \neg[s = 1 \wedge a = 1]]]$



WC N=128 $\mathcal{R}\{\text{"}time\text{"}\}_{=?}[\mathbf{F}[Down\_4 = 0]]$

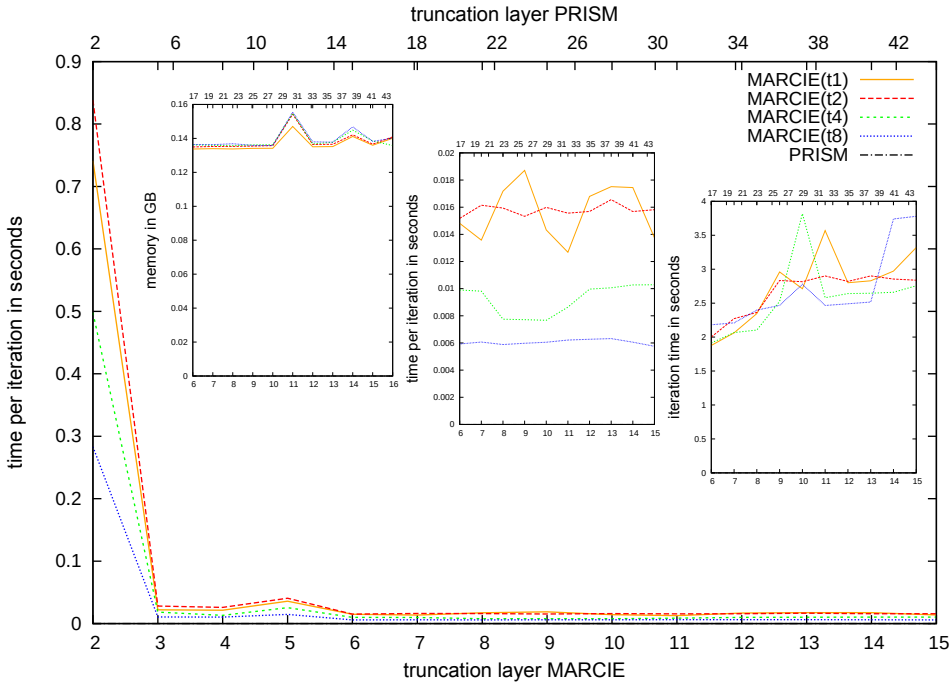


Figure 6.22: $\mathcal{R}\{\varrho\}_{=?}[\mathbf{F}\,\phi\,]$ – PSS and WC

Discussion. Most of the presented plots do not contain figures for PRISM. In particular I obtained only results for the FMS, KANBAN and the PSS models. For the biological models, PRISM terminated with an internal memory error[7]. For the WC model, the limited runtime of at most three hours did not suffice to initialize the required data structures. I observed the same effects for the biological models when investigating smaller scaling factors (e.g. ERK N=10). For the models where PRISM is able to deliver results, the average time per iteration is comparable to *MARCIE*[8]. Again, multi-threading decreases significantly the runtime, though not linear to the number of threads. For the FMS system, the initialization effort inhibits an effective analysis with PRISM. *MARCIE* requires generally the half of the memory compared with PRISM.

In this discipline, *MARCIE's* on-the-fly engine is clearly superior to PRISM's hybrid engine.

6.5 Markovian Approximation

In this section I present experimental results for the computation of the distribution of the accumulated reward deploying the CSRL-pattern

$$\mathcal{P}\{\varrho\}_{=?}[\mathbf{F}^{[\tau,\tau]}_{[0,y]}\phi].$$

It differs slightly from the syntax definition given in Section 3.4. The additional specification of the reward structure ϱ enables to change conveniently the considered MRM. For each experiment I consider a model-specific reward structure representing the frequency of a selected Petri net transition[9] (see Section 3.1). The formula pattern can be understood as follows: "What is the probability to be at time τ in a ϕ-state under the constraint that the specified transition has fired at most y times."

There is no published tool which could be used for a comparison with *MARCIE*. The only existing CSRL model checker MRMC [69] does not support the path formula $[\Phi\mathbf{U}^{[\tau,\tau]}_{[0,y]}\Psi]$. Further, it deploys techniques to compute the distribution which can not be used for the considered state space sizes [107]. However, to

[7] The reported error is caused by the CUDD library. One could try to use the option `-cuddmaxmem` to allow PRISM to allocate more memory.

[8] On could easily improve PRISM's implementation by a representation of the embedded Markov chain using a policy similar to *EMV*.

[9] *MARCIE* defines implicitly for all transitions of a Petri net reward structures which can be used by the related transition name.

permit a judgement, I consider also SPN models with *MARCIE* and PRISM[10] which approximate **explicitly** the MRMs (see Section 3.2.2).

For all experiments, I set the reward bound y to three and specified 30 discretization steps which gives for the constant Δ a value of 0.1. Please note that the choice of the number of steps affects the state space size of the underlying approximating CTMC. The experiments compare the transient analysis of the approximating SPN implicitly (IMP) and explicitly (EXP) with *MARCIE* and explicitly with PRISM. The model-specific settings are summarized in Table 6.6.

Table 6.6: Overview of experiments $\mathcal{P}\{\varrho\}_{=?}[\mathbf{F}^{[\tau,\tau]}_{[0,y]}\phi]$.

Model	N	$\lvert\mathcal{S}^{\mathcal{A}}\rvert$	#	ϕ	τ	it	plot
AKAP	3	52,231,168	525	$cAMP < N/2$	0.001	11	6.23
CLOCK	10	20,614,528	78	$a < N/2$	0.005	9	6.24
ERK	20	54,291,776	894	$ERK < N/2$	0.00625	19	6.24
MAPK	6	43,936,832	31	$kpp + kkpp < N/2$	0.01	59	6.25
LEV	6	43,936,832	98	$kpp + kkpp < N/2$	0.01	13	6.25
FMS	6	17,208,576	58	$P1 < N/2$	0.05	11	6.26
KANBAN	6	14,543,200	15	$x1 < N/2$	0.1	17	6.26
PSS	15	23,592,960	5	$s1 = 1 \wedge \neg(s = 1 \wedge a = 1)$	0.01	13	6.27
WC	32	958,880	78	$Down\_4 = 0$	0.01	705	6.27

'$\lvert\mathcal{S}^{\mathcal{A}}\rvert$' *is the number of reachable states of the approximating CTMC. '#' is the number of terminal nodes in the MTBDD representation. 'it' is the number of iterations.*

[10] *MARCIE* offers export functions to generate the approximating SPN of a given SRN (undocumented feature).

AKAP N=3 $\mathcal{P}\{\text{"}f\_pPDE8\_d\_cAMP\text{"}\}_{=?}[\mathbf{F}^{[0.001,0.001]}_{[0,3]}[cAMP < N/2]]$

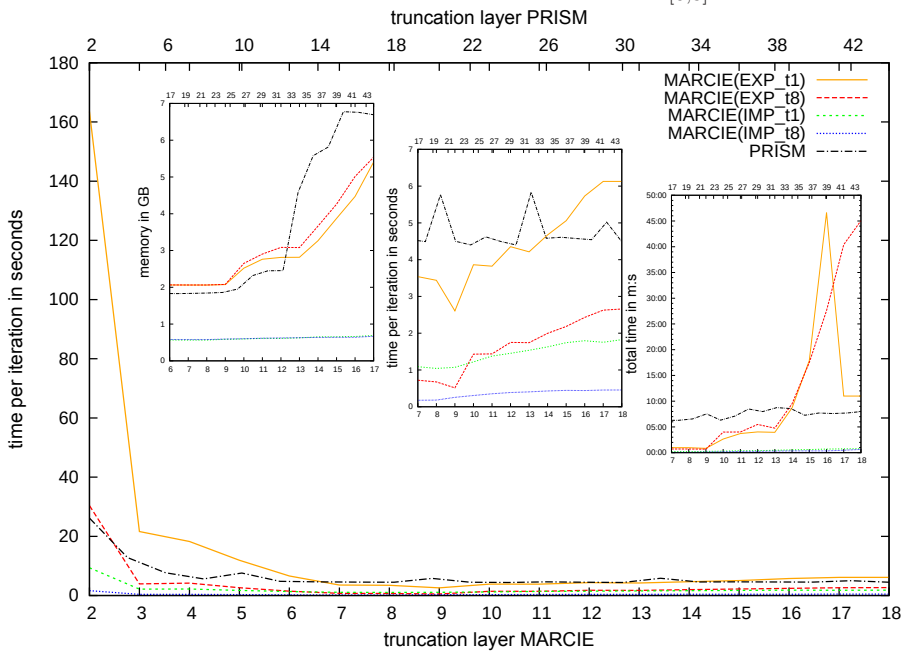


Figure 6.23: $\mathcal{P}\{\varrho\}_{=?}[\mathbf{F}^{[\tau,\tau]}_{[0,y]}\phi]$ – AKAP

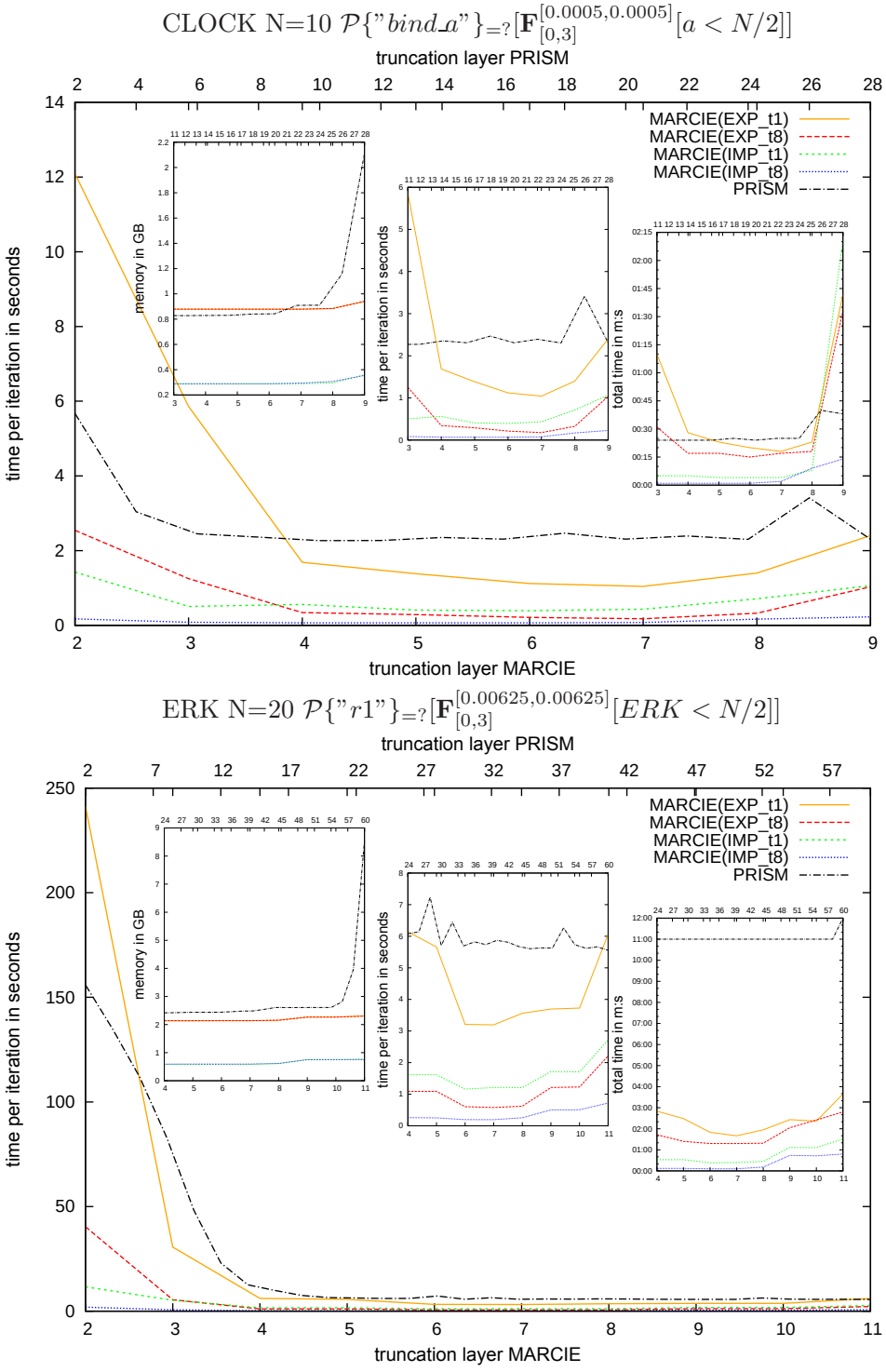


Figure 6.24: $\mathcal{P}\{\varrho\}_{=?}[\mathbf{F}^{[\tau,\tau]}_{[0,y]}\phi]$ – CLOCK and ERK

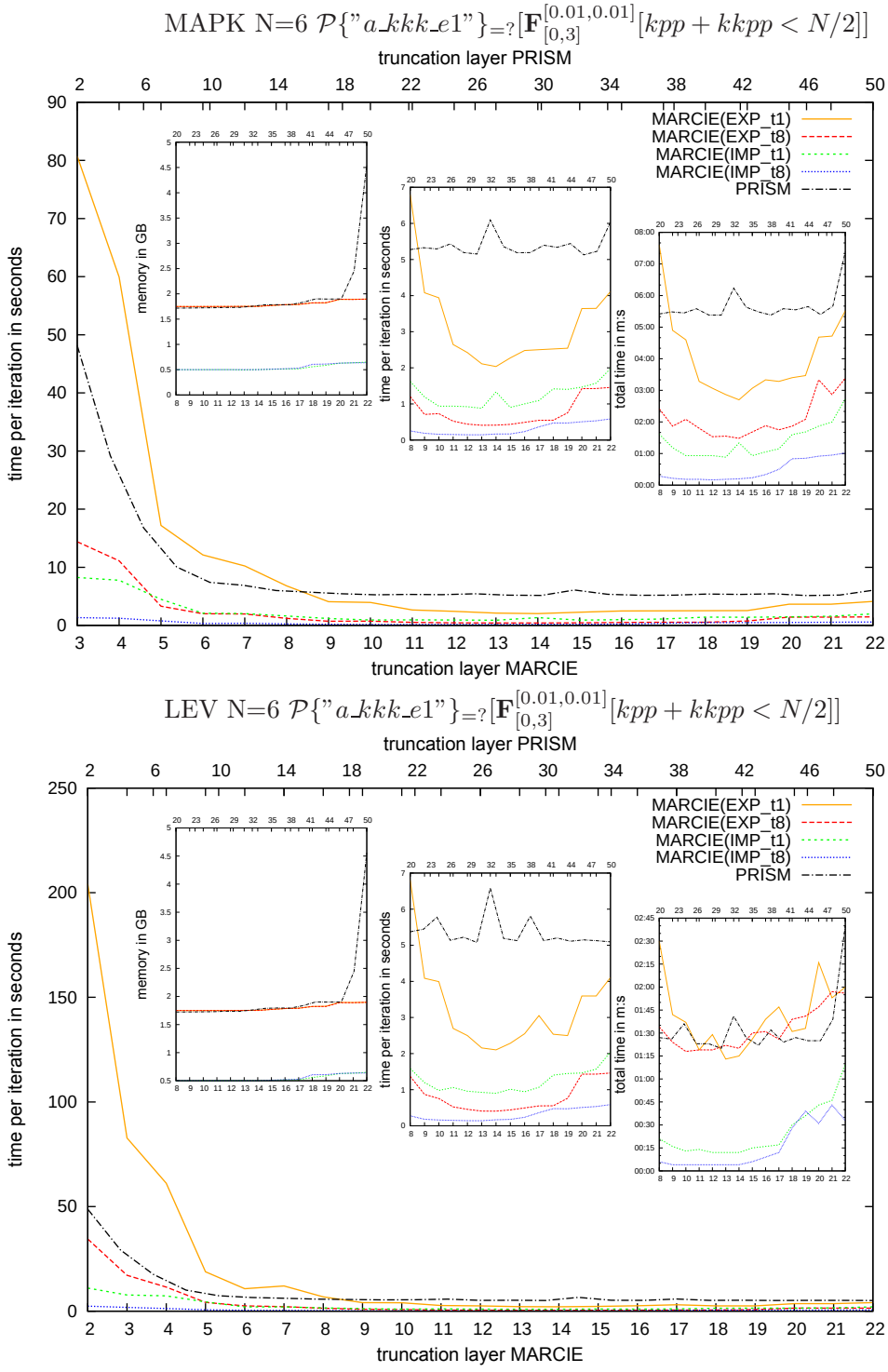


Figure 6.25: $\mathcal{P}\{\varrho\}_{=?}[\mathbf{F}^{[\tau,\tau]}_{[0,y]}\phi]$ – MAPK and LEV

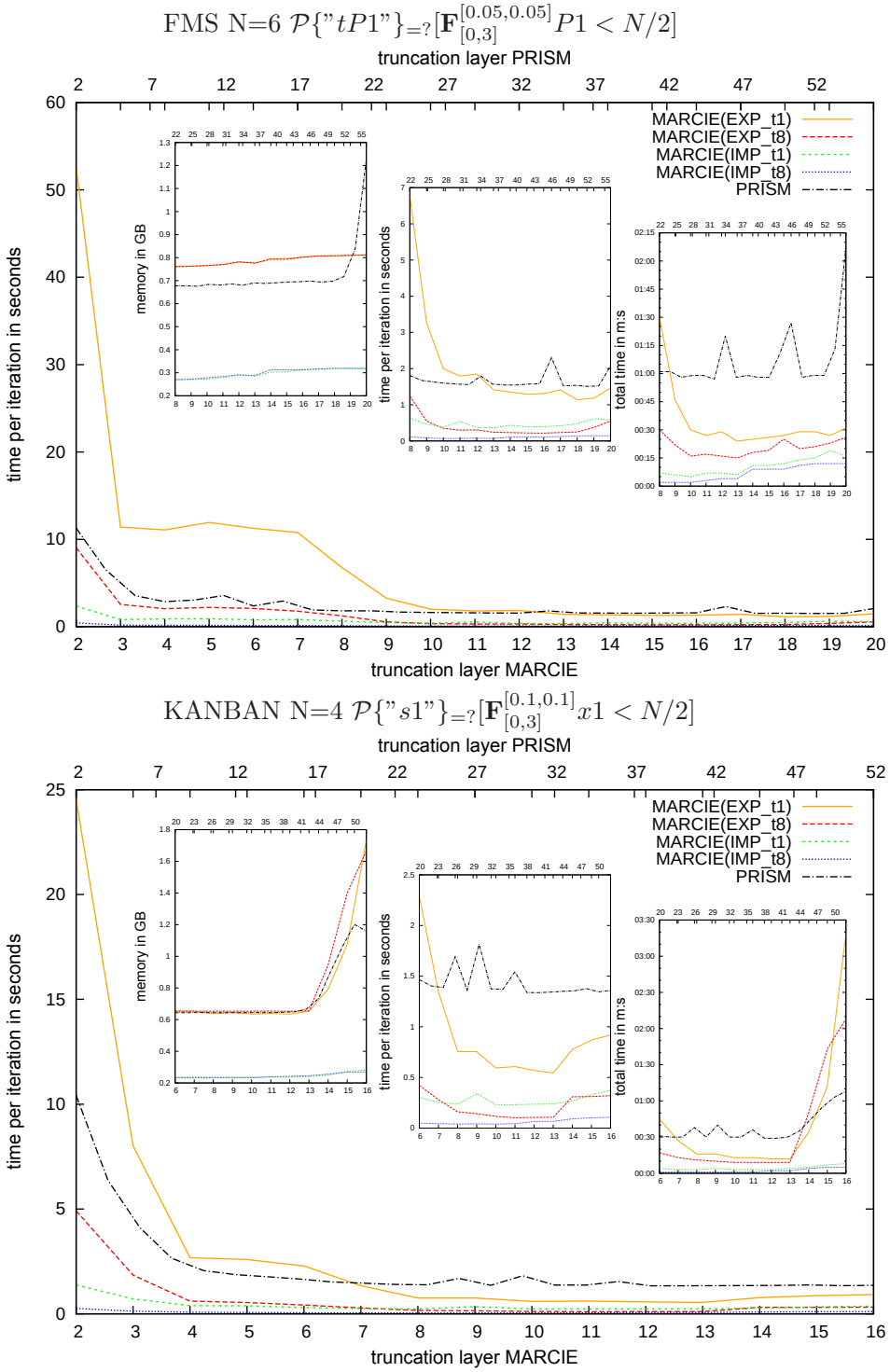
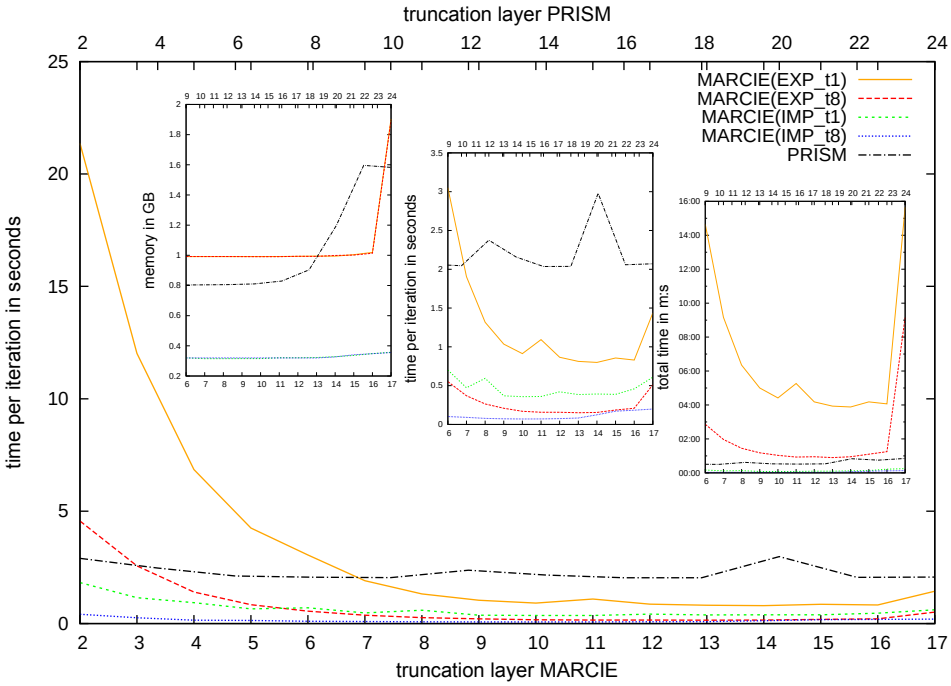


Figure 6.26: $\mathcal{P}\{\varrho\}_{=?}[\mathbf{F}^{[\tau,\tau]}_{[0,y]}\phi]$ – FMS and KANBAN

PSS N=15 $\mathcal{P}\{\text{"}serve1\text{"}\}_{=?}[\mathbf{F}^{[0.01,0.01]}_{[0,3]} s1 = 1 \wedge \neg[s = 1 \wedge a = 1]]$



WC N=32 $\mathcal{P}\{\text{"}Fail\_0\text{"}\}_{=?}[\mathbf{F}^{[0.1,0.1]}_{[0,3]}[Down\_4 = 0]]$

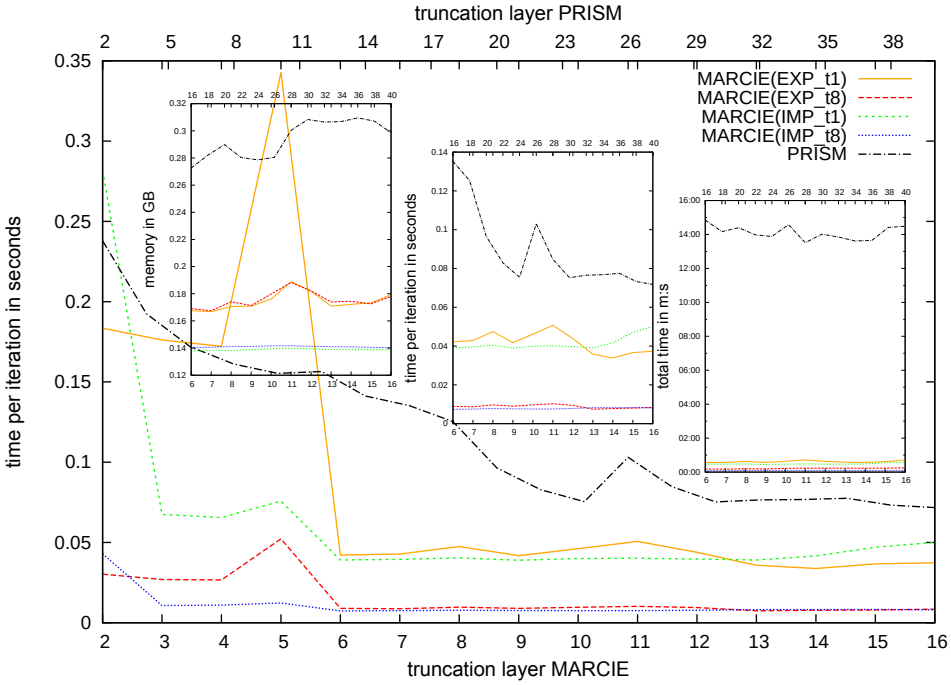


Figure 6.27: $\mathcal{P}\{\varrho\}_{=?}[\mathbf{F}^{[\tau,\tau]}_{[0,y]}\phi]$ – PSS and WC

Discussion. The analysis of the implicit representation of the approximating SPN clearly outperforms the explicit representation concerning runtime as well as memory consumption. Again, deploying multi-threading speeds up the numerical computation. With regard to the explicit representation, the comparison of *MARCIE* with PRISM fits into the picture we got in Section 6.2.

6.6 GSPN versus SPN

Finally, I present some figures for the analysis of GSPN. I run experiments deploying transient analysis as described in Section 6.2 for the GSPN and the related SPN of the FMS (N=4) and WC (N=256) models. We can expect a high runtime and of course a high memory consumption due to the consideration of the vanishing states and the iterative propagation of probabilities.

The obtained figures are shown in Figure 6.28 and I think they speak for themselves.

6.7 Summary

In this chapter I provided results of an experimental comparison of *MARCIE's* symbolic on-the-fly engine with the hybrid engine of the probabilistic model checker PRISM. The presented figures reveal that the on-the-fly approach can in all circumstances compete with established state-of-the-art techniques. Multi-threading always permits to decrease the computation time. With exception of the steady state analysis, *MARCIE* outperforms PRISM's hybrid engine. With regard to the computation of the distribution of the accumulated reward, *MARCIE* is currently the most efficient software.

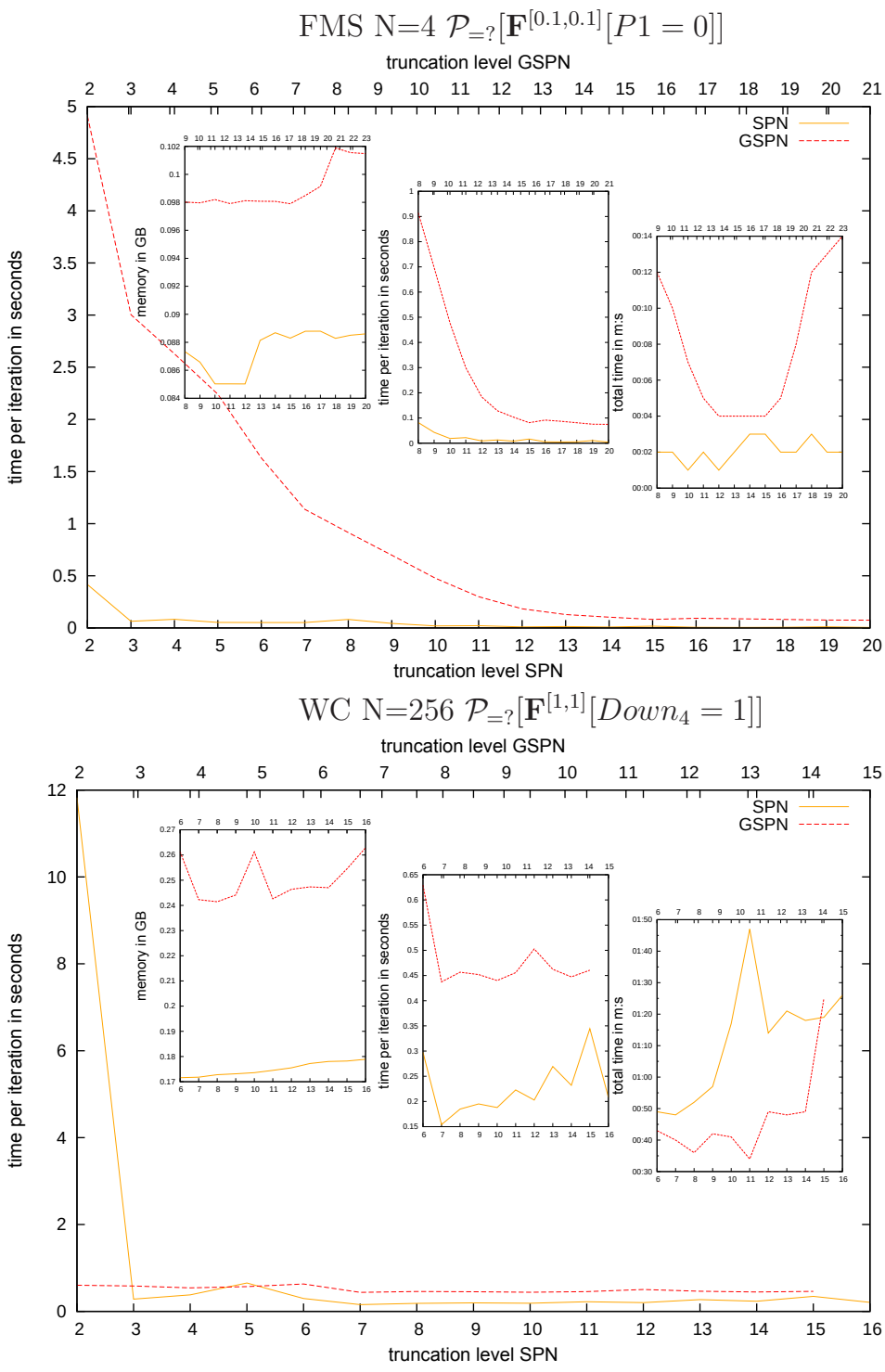


Figure 6.28: $\mathcal{P}_{=?}[\mathbf{F}^{[\tau,\tau]}\phi]$ – FMS and WC – GSPN versus SPN

7 Conclusions and Outlook

7.1 Conclusions

In this thesis I developed an efficient approach for the numerical analysis of stochastic Petri nets whose semantics are Continuous-time Markov chains. The motivation was basically not the lack of tools for this purpose. For the analysis of large Markov models there are tools like PRISM [78] or SMART [29] which implement advanced techniques as Multi-terminal Binary Decision Diagrams or Kronecker expression represented as Matrix Diagrams. My work was rather motivated by the restrictions which are sometimes imposed by these techniques. The application of Kronecker-based techniques requires structured models, whose specification may be challenging, especially when dealing with biologic networks. MTBDD-based approaches suffer from an increase of the BDD variables when enlarging the range of model variables and from an high number of distinct values of the encoded matrix. Therefor my main motivation was to study the combination of an on-the-fly enumeration of the entries of the rate matrix (and of course secondary matrices) with the potential of a symbolic Interval Decision Diagram (IDD)-based state space representation.

Nevertheless, the resulting tool *MARCIE* offers additional features which have been indeed discussed in the literature, but are not part of available tools. *MARCIE* offers the multi-threaded computation of (cumulative) transient, steady state and performability measures for large Markov models. A model checker for the Continuous Stochastic Reward Logic works on top of the related numerical engine.

To be self-contained I provided in Chapter 2 and Chapter 3 background knowledge without digging to much into detail. In Chapter 4 I discussed the symbolic on-the-fly technique and important improvements. In Chapter 5 I gave some insights to the implementation of *MARCIE's* multi-threaded numerical solvers. In Chapter 6 I presented an elaborated comparison with the hybrid engine of the widely used probabilistic model checker PRISM by the help of eight (nine)

case studies[1]. It proves empirically the efficiency of the proposed approach for transient and steady state analysis, performability computation and the analysis based on the embedded Markov chain. We have seen that multi-threading enables to speed-up substantially every kind of analysis I considered. But also in the single-threaded case *MARCIE's* numerical engine is competitive and with exception of the steady-state computation clearly superior to the hybrid MTBDD-engine of the PRISM model checker.

7.2 Outlook

Of course, there are several directions for possible future work.

On the one hand, it would be interesting to investigate the potential of the proposed on-the-fly approach with regard to other formalisms as for instance Continuous-time Markov decision processes as a more general semantics of generalized stochastic Petri nets [13]. It is also desirable that *MARCIE* supports probabilistic model checking of the Linear Time Logic. Further, the existing implementation offers a lot of potential for improvements. The solvers using the Gauss-Seidel and the Pseudo-Gauss-Seidel method are currently restricted to single-threading. The implementation of the Gauss-Seidel solver is in general quite slow. The sparse matrix engine, which is a good choice for small state spaces, could be complemented by a variant exploiting the computation power of graphical processing units as it is discussed in [16] for the PRISM model checker.

On the other hand, the capabilities of the numerical analysis stay still behind the potential of symbolic state space representation techniques.
Indeed, Interval Decision Diagrams, as a representative, enable an extreme compact representation of huge sets of states [115], and the combination of such representation with an on-the-fly enumeration of the matrix entries gives a very memory-efficient approach for the numerical analysis, if the DD-traversal is truncated near the terminal node layer. The problematic increase of the runtime can be reduced on modern work stations by applying multi-threading. **But**, a severe problem remains. The computation vectors have to be in the size of the reachable states.

To deal with this problem we have to increase the available memory, either by using disc memory (out-of-core techniques), or the aggregated main memory

[1] Eight with regard to the model structure. Nine with regard to the rate matrix.

of several processing units (distributed). Out-of-core techniques were basically developed to store the generator $\mathbf{Q}$ [46]. But the technique was also used with a symbolic MTBDD-based matrix representation to store the computation vectors [87].

Distributed numerical analysis with an explicit representation of states and state transitions are reported in [74] and [14]. Both approaches use also out-of-core techniques to store the generator.

In my opinion, further research should be directed to **distributed symbolic** techniques and I want to make a specific suggestion:
The existing distributed approaches are based on an explicit state space representation where the states and, thus implicitly, the state transition relation and the computation vectors, are distributed over a set of processors. The mapping of states to processors is generally realized by a hash-based partition function. The efficiency of a distributed analysis approach depends on the ratio of local and remote computations [74, 14]. Local computation is carried out on the data a specific processor can access without any communication. [74] suggests for instance a row (state) reordering to approximate the locality which can be achieved by applying a sequential BFS state space generation.

I propose to deploy the on-the-fly technique with a specialization of the lexicographic partition, described in Section 5.1.2. The on-the-fly approach enables to partition the state transition relation without a genuine partition of the states. Due to the symbolic IDD encoding, every processor could possess a copy of the LIDD representation $\hat{G}_{\mathcal{S}}$, although its numerical computation would be restricted to the indices of one set in the partition $\mathcal{P}_N$ [2].

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_0	1209	34	0	0	0	0	0	0
P_1	102	1211	34	0	0	0	0	0
P_2	0	102	1211	34	0	0	0	0
P_3	0	0	102	1212	34	0	0	0
P_4	0	0	0	102	1211	34	0	0
P_5	0	0	0	0	102	1211	34	0
P_6	0	0	0	0	0	102	1212	34
P_7	0	0	0	0	0	0	102	1204

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_0	772	0	138	272	0	274	135	0
P_1	34	772	0	137	272	0	274	135
P_2	133	34	736	0	137	136	0	272
P_3	263	134	34	772	0	0	0	0
P_4	0	263	134	34	772	0	0	0
P_5	275	0	129	0	34	736	0	1
P_6	102	307	0	0	0	34	772	0
P_7	1	102	271	0	0	0	34	766

Figure 7.1: The distribution of the non zero entries in the rate matrix of the running example for different variable orders.

In the context of this thesis the ordering/indexing of states is induced by the variable order of the LIDD. In Example 12 I illustrated that Petri net transitions

[2] This holds of course for every symbolic representation of the state space and the state transition relation.

affecting the places at the top layers generate state transitions which represent remote computations. If the place with the maximal boundedness degree is set to the top of the order, the lexicographic partition is based on its possible markings. Petri net transitions which do not affect this place define local state transitions. For the SPN of the running example in Figure 3.3 with $N = 32$ and a state transition partition of size $k = 8$, the variable order

$$\pi_1 = b1 < b2 < res < to1 < to2 < item < ready < req$$

gives 91% local state transitions (Figure 7.1 left), whereas the variable order

$$\pi_2 = to2 < to1 < item < ready < req < res < b2 < b1$$

gives only 57% local transitions (Figure 7.1 right). I am convinced that an consideration of this idea may deliver good results.

A Appendix

A.1 Abstract Net Description Language

The graphical representation is next to the formal semantics the strength of the Petri net approach. Various available tools offer a convenient graphical user interface to specify Petri net models as for instance SNOOPY [58].

However, sometimes it is also advantageous to use a compact textual description language. The Abstract Net Description Language (ANDL) [106] is a lightweight and human readable description language which complements the model specification with bloated XML-based languages and serves as exchange format between *MARCIE* and SNOOPY [58]. It has similarities to command guarded languages as the PRISM language [96] which is in turn inspired by the *reactive modules* formalism [3]. The case studies in Appendix A.2 are given in ANDL and as graphics.

The ANDL representation of a GSPN model consists of three lists. **Constant declarations** are crucial for scalable models as they permit to parametrize initial token values, arc weights, firing rates of stochastic transitions, or weights of immediate transitions. The constants are followed by the **place** and **transition declarations**, which have both to be non-empty. Each declaration block is opened by a keyword – **constants**, **places**, and **transitions**.

Constants can either be declared to be of type *integer* or of type *double*. *Integer* constants can be used as initial token values, as arc weights or as part of firing rates of timed transitions or weights of immediate transitions. *Double* constants can only be used for the definition of rate or weight functions. Constant declarations in ANDL resemble those in C-like programming languages.

A place declaration resembles a variable initialization: the place name followed by an assignment of the initial number of tokens. A transitions declaration consists the following four parts:

1. A transition name.
2. A ′&′-separated list of Boolean guard expressions. Each expression repre-

sents the connection to a place by a weighted read arc, a weighted inhibitor arc, a weighted equal arc (graphic: —●–●)[1], or an unweighted modifier arc, which in fact allows four expression types:

- *place* < *number* for an inhibitor arc,
- *number* <= *place* for a read arc,
- *number* <= *place* < *number* for the combination of a read and an inhibitor arc,
- *number* = *place* for an equal arc.
- *place* for a modifier arc.

The *place* is the identifier of a place. The *number* can be an arbitrary meaningful arithmetic expression containing also constants.

3. A '&'-separated list of arithmetic update expressions. Each expression represents the connection to a place by a weighted standard arc. Therefor we have again two expression types:
   - *place* – *number* for a decrease of tokens,
   - *place* + *number* for an increase of tokens,

   again with *place* as the identifier of a place and *number* as a meaningful arithmetic expression defining a natural number.
4. A function, which can be an arbitrary arithmetic expression containing place and constant names. The function may represent the firing rate of a timed transition or the weight of an immediate transition. To distinguish the transition types, ANDL defines the keywords **stochastic** and **immediate**. When specifying biological networks the rate functions often represent specific kinetics as mass-action. ANDL offers a pre-defined *MassAction(c)* macro, where c is a meaningful arithmetic expression defining a real number.

Although the concept of guards and update expressions is semantically and syntactically close to the specification style of commands in the PRISM language, there are important differences:

1. The guards and updates encode the set of arcs of the net.
2. There is no force to bound the value range of variables (places).

[1] An equal arc with weight w can be emulated by a read arc with weight w and an inhibitor arc with weight $w + 1$. Equal arcs are syntactic sugar.

3. Update expressions implicitly have the semantics of transition firing. A transition which decreases the amount of tokens on a place is only enabled, if there are enough tokens. No further guard is required.

Example 14

The ANDL specification of the running example (Figure 3.1) looks as follows.

```
gspn [running_example_gspn] {
constants:
    int N;
    double i2 = 3;
    double i1 = 2;
    double c1 = 0.9;
    double c2 = 1- c1;
    double f1 = 0.8;
    double f2 = 1 - f2;
    double cr = 1;
    double sr = 1;
places:
    msg = 0;
    b1 = 0;
    b2 = 0;
    req = 1;
    response = 0;
    to1 = 0;
    to2 = 0;
    item = 0;
    ready = 1;
transitions: //stochastic by default
   consume  :  /* empty */ : [req + 1] & [response - 1] : cr;
   send  : [b1] & [b2]  : [msg + 1] & [item + 1] & [ready - 1] :
   sr/(1+b1+b2);
   insert_b1 : [b1 < N]   : [b1 + 1] & [ready + 1] & [item - 1] & [to1 - 1] :
   i1*(1+b1);
   insert_b2 : [b2 < N]  : [b2 + 1] & [ready + 1] & [item - 1] & [to2 - 1] :
   i2*(1+b2);
immediate:
   fetch_b1 :  /* empty */ : [response + 1] & [b1 - 1] & [req - 1] :  f1;
   fetch_b2 :  /* empty */ : [response + 1] & [b2 - 1] & [req - 1] : f2;
   choose_b2 : /* empty */  : [to2 + 1] & [msg - 1] : c2;
   choose_b1 : /* empty */ : [to1 + 1] & [msg - 1] : c1;

}
```

★

In ANDL a reward structure is defined next to the Perti net model. The keyword **rewards** indicates a reward structure definition followed by the reward structure name. Curly brackets enclose the set of reward items. A reward item consists of an interval logic expression, the guard, and an arithmetic expression, the rate reward, separated by a colon. The reward items are separated by semicolons.

Example 15

The reward structure given in Example 6 looks as follows:

```
//waiting time
rewards [ wt ] {
  to1 > 0 & b1 = N & b2 < N : 1 ;
  to2 > 0 & b2 = N & b1 < N : 1 ;
}
```

★

A.2 Case studies

In the following I give the ANDL specifications of the case studies I deployed for the experimental evaluation presented in Chapter 6. All models are taken from the literature and edited as SPN or GSPN. The scalable Petri nets were created with the Petri net editor Snoopy [58]. Its export to ANDL yields the following specifications. I considered four biological networks and four technical models. The state space sizes for selected settings are given in Table 2.1 and Table 2.2. The technical models represent established CTMC benchmarks. I do not present here a detailed explanation of the models and refer to the related literature for background reading.

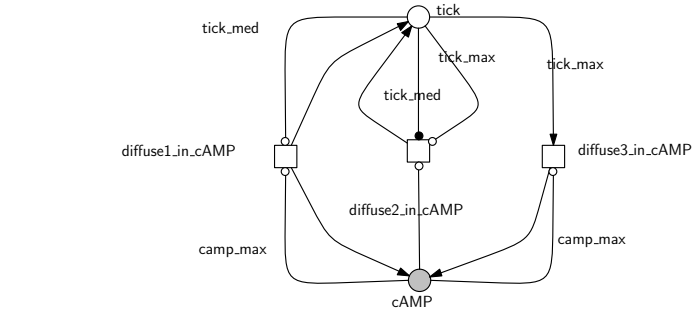
A.2.1 Biological Networks

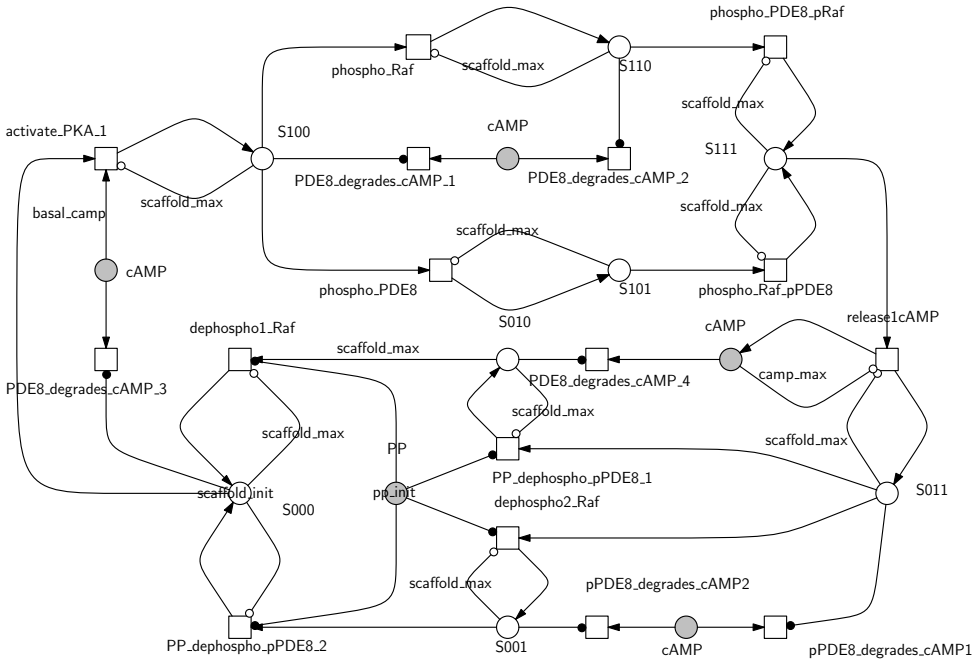
A-kinase anchoring protein (AKAP)

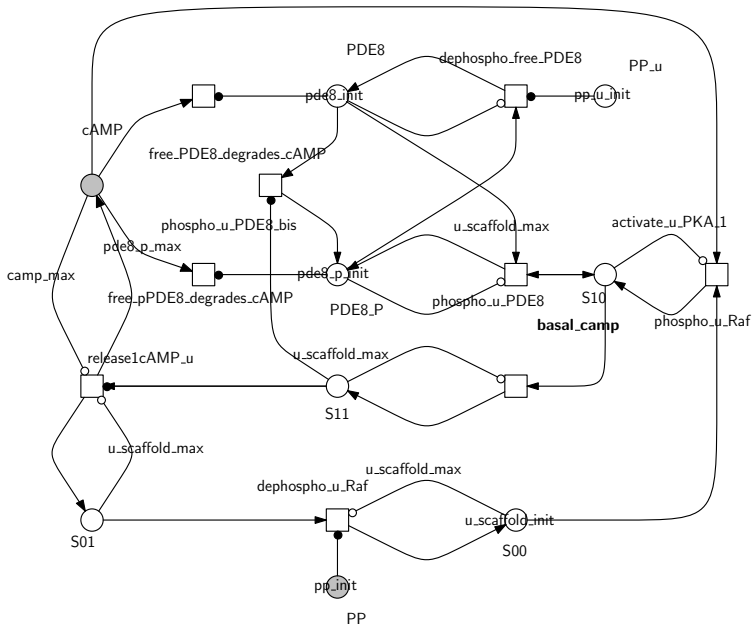
An SPN modeling the scaffold-mediated crosstalk between the cyclic adenosine monophosphate (cAMP) and the Raf-1/MEK/ERK pathways. The model is taken from [4] where it is given in the PRISM language. The model is scalable by the initial number of tokens on the places `PDE8` and `S000` and by the arc weights of inhibitor arcs specifying the boundedness degree of the places represented by the model constant N.

```
spn [  AKAP ] {
constants:
 double CONC = 12 ;          int N ;                    double H = CONC/N ;
 int camp_max = 10*N ;       int camp_init = 0 ;        int basal_camp   = 1 ;
 int scaffold_init = N ;     int u_scaffold_init = N ;  int scaffold_max = N ;
 int u_scaffold_max = N ;

 int pde8_init = N/2 ;    int pde8_max = N/2 ;
 int pde8_p_init = 0 ;    int pde8_p_max = N/2 ;

 int pp_init = scaffold_init ;                  int  pp_u_init = u_scaffold_init ;
 double diffuse_rate_cAMP = 1 ;                 double diffuse_rate_PP = 1 ;
 double pka_activate1 = 1 ;                     double camp_inhib = 1 ;
 double camp_more_inhib = 2.5 ;                 double raf_phospho = 1;
 double raf_dephospho = 1 ;                     double release_camp1 = 1 ;
 double release_camp2 = 1.5*release_camp1 ;     double release_camp3 = 2.0*release_camp2;
 double pde8_phospho = raf_phospho ;            double pde8_dephospho = raf_dephospho ;
 double p_pde8_degrade = raf_dephospho ;        double pde8_degrade =  raf_dephospho/3 ;
 double free_pde8_phospho = pde8_phospho/3 ;    double free_pde8_dephospho = free_pde8_phospho ;
 double free_p_pde8_degrade = p_pde8_degrade/3 ; double free_pde8_degrade = pde8_degrade/3 ;

 int p = 2 ;  int tick_max = 10 ;  int tick_med = 5 ;
places:
  cAMP   = 0;     tick  = 0;      S00 = u_scaffold_init;
  S10   = 0;      S01   = 0;      S11 = 0;
  S000   = scaffold_init;         S100 = 0;
  S101   = 0;     S110  = 0;      S011 = 0;
  S010   = 0;     S001  = 0;      S111 = 0;
  PP_u   = pp_u_init;             PP = pp_init ;
  PDE8_P = pde8_p_init;           PDE8 = pde8_init;

transitions:
  diffuse1_in_cAMP : [ tick < tick_med ] & [ cAMP < camp_max ] : [tick + 1]& [cAMP + 1] :
     diffuse_rate_cAMP / H  ;
  diffuse3_in_cAMP : [tick  = tick_max ] & [ cAMP < camp_max ] : [tick -tick_max ] & [cAMP + 1] :
     diffuse_rate_cAMP / H ;
  diffuse2_in_cAMP : [ tick_med <= tick < tick_max ] & [ cAMP < (camp_max -1 ) ] : [tick + 1] :
     diffuse_rate_cAMP / H ; activate_PKA_1 : [ S100 < scaffold_max ] & [
  basal_camp <= cAMP ] : [ S100 + 1 ] & [ S000 - 1 ] & [cAMP - 1] :
     pka_activate1*H*S000*cAMP  ;
  activate_u_PKA_1 : [ S10 < u_scaffold_max ] & [ basal_camp <= cAMP ] :
     [cAMP - 1] & [ S10 + 1 ] & [ S00 - 1 ] : pka_activate1*H*S00*cAMP ;
  release1cAMP : [ S011 < scaffold_max ] & [ cAMP < camp_max ] :
     [ S111 - 1 ] & [ S011 + 1 ] & [cAMP + 1] : release_camp1*S111 ;
  release1cAMP_u : [ S01 < u_scaffold_max ] & [ cAMP < camp_max ] :
     [ S01 + 1 ] & [ S11 - 1 ] & [ cAMP + 1 ] : release_camp1*S11  ;
  free_pPDE8_degrades_cAMP : [ 1 <=  PDE8_P ] : [cAMP - 1] : free_p_pde8_degrade*H*PDE8_P*cAMP;
  pPDE8_degrades_cAMP1 : [ 1 <= S011  ] : [cAMP - 1 ] : p_pde8_degrade*H*S011*cAMP;
  pPDE8_degrades_cAMP2 : [ 1 <= S001] : [cAMP - 1] : p_pde8_degrade*H*S001*cAMP;
  PDE8_degrades_cAMP_1 : [ 1 <= S100 ] : [cAMP - 1] : pde8_degrade*H*S100*cAMP;
  PDE8_degrades_cAMP_2 : [ 1 <= S110 ] : [cAMP - 1] : pde8_degrade*H*S110*cAMP;
  PDE8_degrades_cAMP_3 : [ 1 <= S000 ] : [cAMP - 1] : pde8_degrade*H*S000*cAMP;
  PDE8_degrades_cAMP_4 : [ 1 <= S010 ] : [cAMP - 1] : pde8_degrade*H*S010*cAMP;
  free_PDE8_degrades_cAMP : [ 1 <= PDE8 ] : [cAMP - 1] : free_pde8_degrade*H*PDE8*cAMP;
  phospho_u_Raf : [ S11 < u_scaffold_max ] : [ S10 - 1] & [ S11 + 1] : raf_phospho*S10 ;
  dephospho2_Raf : [ 1 <= PP] & [ S001 < scaffold_max ] : [ S001 + 1] & [ S011 - 1] :
      raf_dephospho*H*PP *S011 ;
  dephospho_u_Raf : [ 1 <= PP] & [ S00 < u_scaffold_max ] : [ S00 + 1] & [ S01 - 1] :
```

```
   raf_dephospho*H*PP *S01 ;
phospho_Raf : [ S110 < scaffold_max ] : [ S110 + 1] & [ S100 - 1] : raf_phospho*S100 ;
phospho_Raf_pPDE8 : [ S111 < scaffold_max ] : [ S111 + 1] & [ S101 - 1] : raf_phospho*S101 ;
phospho_PDE8 : [ S101 < scaffold_max ] : [ S100 - 1 ] & [ S101 + 1 ] : pde8_phospho*S100 ;
phospho_PDE8_pRaf : [ S111 < scaffold_max ] : [ S110 - 1] & [ S111 + 1] : pde8_phospho*S110 ;
phospho_u_PDE8 : [ PDE8_P < pde8_p_max ] & [ 1 <= S10 ] : [ PDE8_P +1 ] & [ PDE8-1] :
    free_pde8_phospho*H*PDE8*S10  ;
phospho_u_PDE8_bis : [ 1 <= S11 ] : [ PDE8_P +1 ] & [ PDE8 -1 ] :
    free_pde8_phospho*PDE8*H *S11  ;
PP_dephospho_pPDE8_1 : [ 1 <= PP] & [ S010 < scaffold_max ] : [ S010 + 1] & [ S011 - 1] :
    pde8_dephospho*PP*S011  ;
PP_dephospho_pPDE8_2 : [ 1 <= PP] & [ S000 < scaffold_max ] : [ S001 - 1] & [ S000 + 1] :
    pde8_dephospho*PP*S001;
dephospho_free_PDE8 : [ 1 <= PP] & [ PDE8 < pde8_max ] : [ PDE8_P - 1] & [ PDE8 + 1] :
    free_pde8_dephospho*H*PP*PDE8_P ;
}
```

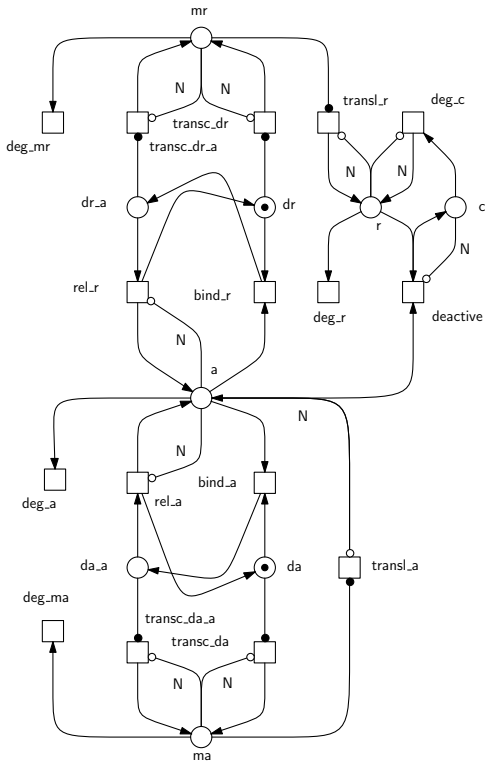

tick
tick_med
tick_max
tick_max
tick_med
diffuse1_in_cAMP
diffuse3_in_cAMP
diffuse2_in_cAMP
camp_max
camp_max
cAMP


phospho_PDE8_pRaf
phospho_Raf
scaffold_max
S110
scaffold_max
activate_PKA_1
S100
cAMP
S111
PDE8_degrades_cAMP_1
PDE8_degrades_cAMP_2
scaffold_max
basal_camp
scaffold_max
cAMP
phospho_PDE8
S101
phospho_Raf_pPDE8
S010
release1cAMP
dephospho1_Raf
cAMP
scaffold_max
PDE8_degrades_cAMP_4
camp_max
PDE8_degrades_cAMP_3
scaffold_max
scaffold_max
PP
PP_dephospho_pPDE8_1
scaffold_max
scaffold_init
S000
pp_init
dephospho2_Raf
S011
scaffold_max
pPDE8_degrades_cAMP2
PP_dephospho_pPDE8_2
S001
cAMP
pPDE8_degrades_cAMP1


PDE8
dephospho_free_PDE8
PP_u
pde8_init
pp_u_init
cAMP
free_PDE8_degrades_cAMP
phospho_u_PDE8_bis
u_scaffold_max
activate_u_PKA_1
pde8_p_max
camp_max
pde8_p_init
free_pPDE8_degrades_cAMP
PDE8_P
phospho_u_PDE8
S10
basal_camp
phospho_u_Raf
release1cAMP_u
u_scaffold_max
S11
u_scaffold_max
u_scaffold_max
dephospho_u_Raf
u_scaffold_init
S01
S00
pp_init
PP

Circadian Clock (CLOCK)

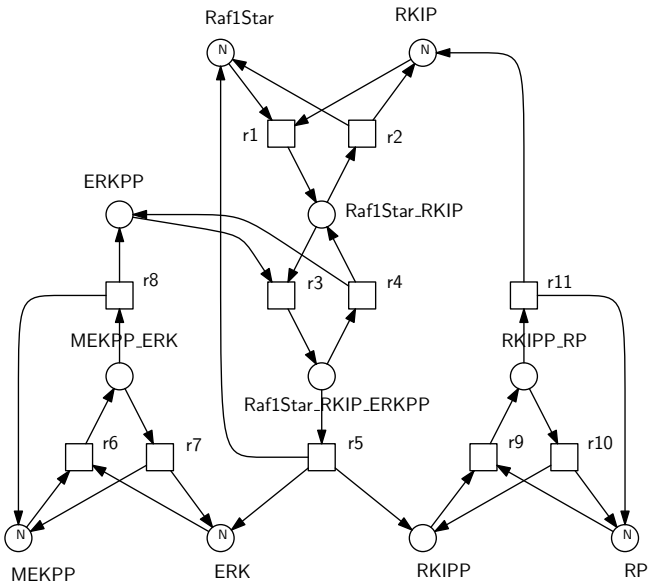
An SPN of the circadian clock model published in [117]. The model is scalable concerning the boundedness degree of the places `a,ma,mr,r` and `c` by means of inhibitor arc weights. The rate functions are specified by means of the `MassAction` function pattern.




```
spn [CLOCK_N] {
constants:
    int N ;
    double p_transc_da = 50;
    double p_transc_da_a = 500;
    double p_transc_dr = 0.01;
    double p_transc_dr_a = 50;
    double p_transl_a = 50;
    double p_bind_a = 1;
    double p_transl_r = 5;
    double p_bind_r = 1;
    double p_rel_a = 50;
    double p_rel_r = 100;
    double p_deactive = 2;
    double p_deg_a = 1;
    double p_deg_c = 1;
    double p_deg_r = 0.2;
    double p_deg_ma = 10;
    double p_deg_mr = 0.5;
places:
    a = 0;
    c = 0;
    da = 1;
    da_a = 0;
    dr = 1;
    dr_a = 0;
    ma = 0;
    mr = 0;
    r = 0;
transitions:
    bind_a :  : [da_a + 1] & [da - 1] & [a - 1] : MassAction(p_bind_a);
    bind_r :  : [dr_a + 1] & [a - 1] & [dr - 1] : MassAction(p_bind_r);
    deactive : [c < N] : [c + 1] & [a - 1] & [r - 1] : MassAction(p_deactive);
    deg_a :  : [a - 1] : MassAction(p_deg_a);
    deg_c : [r < N] : [r + 1] & [c - 1] : MassAction(p_deg_c);
    deg_ma :  : [ma - 1] : MassAction(p_deg_ma);
    deg_mr :  : [mr - 1] : MassAction(p_deg_mr);
    deg_r :  : [r - 1] : MassAction(p_deg_r);
    rel_a : [a < N] : [da + 1] & [a + 1] & [da_a - 1] : MassAction(p_rel_a);
    rel_r : [a < N] : [a + 1] & [dr + 1] & [dr_a - 1] : MassAction(p_rel_r);
    transc_da : [da >= 1] & [ma < N] : [ma + 1] : MassAction(p_transc_da);
    transc_da_a : [da_a >= 1] & [ma < N] : [ma + 1] : MassAction(p_transc_da_a);
    transc_dr : [dr >= 1] & [mr < N] : [mr + 1] : MassAction(p_transc_dr);
    transc_dr_a : [dr_a >= 1] & [mr < N] : [mr + 1] : MassAction(p_transc_dr_a);
    transl_a : [ma >= 1] & [a < N] : [a + 1] : MassAction(p_transl_a);
    transl_r : [mr >= 1] & [r < N] : [r + 1] : MassAction(p_transl_r);
}
```

ERK

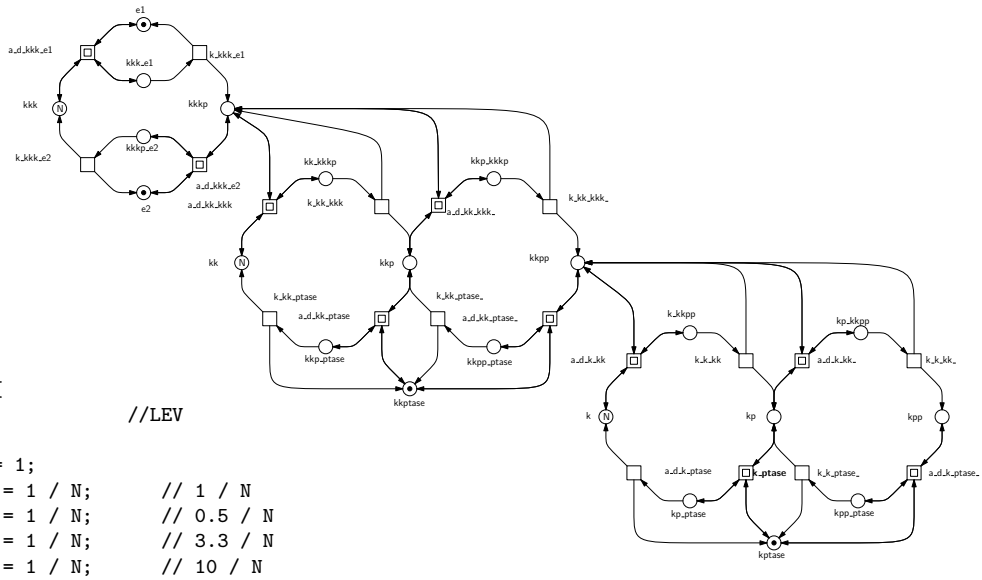
An SPN model of RKIP inhibited ERK pathway published in [26]. The model is scalable by the initial state. The number of tokens on the places `Raf1Star, MAKPP, RKIPP` and `RP` is represented by the constant N. The rate functions are specified by means of the `MassAction` function pattern.




```
spn [ERK_N]
{

constants:

    in  N = 1;

places:

    Raf1Star = N;
    RKIP = N;
    Raf1Star_RKIP = 0;
    ERKPP = 0;
    MEKPP_ERK = 0;
    Raf1Star_RKIP_ERKPP = 0;
    RKIPP_RP = 0;
    MEKPP = N;
    ERK = N;
    RKIPP = 0;
    RP = N;

transitions:

    r1 :  : [Raf1Star_RKIP + 1] & [Raf1Star - 1] & [RKIP - 1] : MassAction( 0.53);
    r2 :  : [Raf1Star + 1] & [RKIP + 1] & [Raf1Star_RKIP - 1] : MassAction( 0.0072 );
    r3 :  : [Raf1Star_RKIP_ERKPP + 1] & [Raf1Star_RKIP - 1] & [ERKPP - 1] : MassAction( 0.625 );
    r4 :  : [Raf1Star_RKIP + 1] & [ERKPP + 1] & [Raf1Star_RKIP_ERKPP - 1] : MassAction( 0.00245 );
    r6 :  : [MEKPP_ERK + 1] & [MEKPP - 1] & [ERK - 1] : MassAction( 0.8 );
    r7 :  : [ERK + 1] & [MEKPP + 1] & [MEKPP_ERK - 1] : MassAction( 0.0075 );
    r9 :  : [RKIPP_RP + 1] & [RP - 1] & [RKIPP - 1] : MassAction( 0.92 );
    r10 :  : [RP + 1] & [RKIPP + 1] & [RKIPP_RP - 1] : MassAction( 0.00122 );
    r5 :  : [ERK + 1] & [RKIPP + 1] & [Raf1Star + 1] & [Raf1Star_RKIP_ERKPP - 1] : MassAction( 0.0315  );
    r8 :  : [ERKPP + 1] & [MEKPP + 1] & [MEKPP_ERK - 1] : MassAction( 0.071 );
    r11 :  : [RKIP + 1] & [RP + 1] & [RKIPP_RP - 1] : MassAction( 0.87 );

}
```

Mitogen-Actived Protein Kinase (MAPK)

An SPN of the mitogen-actived protein kinase cascade (MAPK) derived from the PRISM specification taken from [77]. The layout of the SPN is taken from [51]. The rate functions are specified by means of the `MassAction` function pattern. I consider also a second version (LEV) with different rate-specific constants to define the mass-action kinetics which are deployed in [51]. The model is scalable by the number of initial tokens on the places `k,kk` and `kkk`.



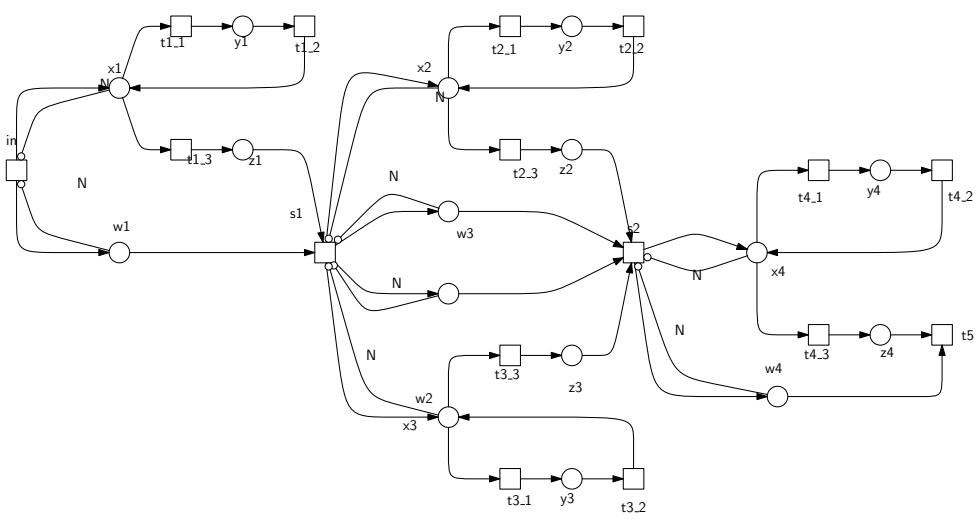

```
spn [MAPK_N] {
//MAPK                  //LEV
constants:
    double N = 1;
    double a1 = 1 / N;      // 1 / N
    double a2 = 1 / N;      // 0.5 / N
    double a3 = 1 / N;      // 3.3 / N
    double a4 = 1 / N;      // 10 / N
    double a5 = 1 / N;      // 3.3 / N
    double a6 = 1 / N;      // 10 / N
    double a7 = 1 / N;      // 20 / N
    double a8 = 1 / N;      // 5 / N
    double a9 = 1 / N;      // 20 / N
    double a10 = 1 / N;     // 5 / N
    double d1 = 150;        // 0.4
    double d2 = 150;        // 0.1
    double d3 = 150;        // 0.42
    double d4 = 150;        // 0.8
    double d5 = 150;        // 0.4
    double d6 = 150;        // 0.8
    double d7 = 150;        // 0.6
    double d8 = 150;        // 0.4
    double d9 = 150;        // 0.6
    double d10 = 150;       // 0.4
    double k1 = 150;        // 0.1
    double k2 = 150;        // 0.1
    double k3 = 150;        // 0.1
    double k4 = 150;        // 0.1
    double k5 = 150;        // 0.1
    double k6 = 150;        // 0.1
    double k7 = 150;        // 0.1
```

```
    double k8 = 150;        // 0.1
    double k9 = 150;        // 0.1
    double k10 = 150;       // 0.1
places:
    e1 = 1;
    e2 = 1;
    k = N;
    k_kkpp = 0;
    kk = N;
    kk_kkkp = 0;
    kkk = N;
    kkk_e1 = 0;
    kkkp = 0;
    kkkp_e2 = 0;
    kkp = 0;
    kkp_kkkp = 0;
    kkp_ptase = 0;
    kkpp = 0;
    kkpp_ptase = 0;
    kkptase = 1;
    kp = 0;
    kp_kkpp = 0;
    kp_ptase = 0;
    kpp = 0;
    kpp_ptase = 0;
    kptase = 1;
transitions:
    a_k_kk :  : [k_kkpp + 1] & [k - 1] & [kkpp - 1] : MassAction( a7  );
    a_k_kk_ :  : [kp_kkpp + 1] & [kkpp - 1] & [kp - 1] : MassAction( a9  );
    a_k_ptase :  : [kp_ptase + 1] & [kptase - 1] & [kp - 1] : MassAction( a8  );
    a_k_ptase_ :  : [kpp_ptase + 1] & [kptase - 1] & [kpp - 1] : MassAction( a10 );
    a_kk_kkk :  : [kk_kkkp + 1] & [kk - 1] & [kkkp - 1] : MassAction( a3 );
    a_kk_kkk_ :  : [kkp_kkkp + 1] & [kkkp - 1] & [kkp - 1] : MassAction( a5 );
    a_kk_ptase :  : [kkp_ptase + 1] & [kkp - 1] & [kkptase - 1] : MassAction( a4 );
    a_kk_ptase_ :  : [kkpp_ptase + 1] & [kkptase - 1] & [kkpp - 1] : MassAction( a6 );
    a_kkk_e1 :  : [kkk_e1 + 1] & [e1 - 1] & [kkk - 1] : MassAction( a1 );
    a_kkk_e2 :  : [kkkp_e2 + 1] & [e2 - 1] & [kkkp - 1] : MassAction( a2 );
    d_k_kk :  : [k + 1] & [kkpp + 1] & [k_kkpp - 1] : MassAction( d7 );
    d_k_kk_ :  : [kkpp + 1] & [kp + 1] & [kp_kkpp - 1] : MassAction( d9 );
    d_k_ptase :  : [kptase + 1] & [kp + 1] & [kp_ptase - 1] : MassAction( d8 );
    d_k_ptase_ :  : [kptase + 1] & [kpp + 1] & [kpp_ptase - 1] : MassAction( d10 );
    d_kk_kkk :  : [kk + 1] & [kkkp + 1] & [kk_kkkp - 1] : MassAction( d3 );
    d_kk_kkk_ :  : [kkkp + 1] & [kkp + 1] & [kkp_kkkp - 1] : MassAction( d5 );
    d_kk_ptase :  : [kkp + 1] & [kkptase + 1] & [kkp_ptase - 1] : MassAction( d4 );
    d_kk_ptase_ :  : [kkptase + 1] & [kkpp + 1] & [kkpp_ptase - 1] : MassAction( d6 );
    d_kkk_e1 :  : [e1 + 1] & [kkk + 1] & [kkk_e1 - 1] : MassAction( d1 );
    d_kkk_e2 :  : [e2 + 1] & [kkkp + 1] & [kkkp_e2 - 1] : MassAction( d2 );
    k_k_kk :  : [kp + 1] & [kkpp + 1] & [k_kkpp - 1] : MassAction( k7 );
    k_k_kk_ :  : [kpp + 1] & [kkpp + 1] & [kp_kkpp - 1] : MassAction( k9 );
    k_k_ptase :  : [kptase + 1] & [k + 1] & [kp_ptase - 1] : MassAction( k8 );
    k_k_ptase_ :  : [kp + 1] & [kptase + 1] & [kpp_ptase - 1] : MassAction( k10 );
    k_kk_kkk :  : [kkp + 1] & [kkkp + 1] & [kk_kkkp - 1] : MassAction( k3 );
    k_kk_kkk_ :  : [kkpp + 1] & [kkkp + 1] & [kkp_kkkp - 1] : MassAction( k5 );
    k_kk_ptase :  : [kk + 1] & [kkptase + 1] & [kkp_ptase - 1] : MassAction( k4 );
    k_kk_ptase_ :  : [kkp + 1] & [kkptase + 1] & [kkpp_ptase - 1] : MassAction( k6 );
    k_kkk_e1 :  : [kkkp + 1] & [e1 + 1] & [kkk_e1 - 1] : MassAction( k1 );
    k_kkk_e2 :  : [e2 + 1] & [kkk + 1] & [kkkp_e2 - 1] : MassAction( k2 );
}
```

A.2.2 Technical Systems

KANBAN

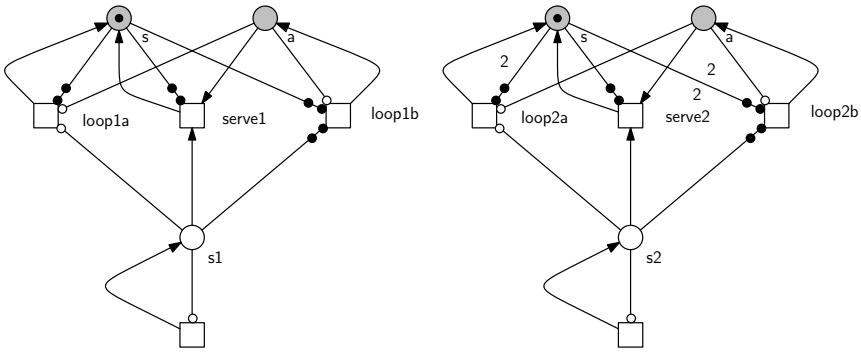
An SPN model of the Kanban system with four cells taken from [34]. The model is scalable concerning the capacity of the places `xi` and `wi` which is realized by inhibitor arcs with weight `N`.




```
spn [KANBAN_N] {

constants:
    int N = 5;

places:
    x1 = 0;
    y1 = 0;
    z1 = 0;
    x2 = 0;
    y2 = 0;
    z2 = 0;
    x3 = 0;
    y3 = 0;
    z3 = 0;
    x4 = 0;
    y4 = 0;
    z4 = 0;
    w1 = 0;
    w2 = 0;
    w3 = 0;
    w4 = 0;

transitions:
    in : [x1 < N] & [w1 < N] : [x1 + 1] & [w1 + 1] : 1 ; // in1
    t1_1 :  : [y1 + 1] & [x1 - 1] : 0.36 ; // redo1
    t1_2 :  : [x1 + 1] & [y1 - 1] : 0.3 ;  // back2
    t1_3 :  : [z1 + 1] & [x1 - 1] : 0.84 ; // ok1
    t2_1 :  : [y2 + 1] & [x2 - 1] : 0.42 ; // redo2
    t2_2 :  : [x2 + 1] & [y2 - 1] : 0.3 ;  // back2
    t2_3 :  : [z2 + 1] & [x2 - 1] : 0.98;   // ok2
    s2 : [w4 < N] & [x4 < N] :
        [w4 + 1] & [x4 + 1] & [z2 - 1] & [z3 - 1] & [w2 - 1] & [w3 - 1] : 0.5 ; //synch234;
    s1 : [w3 < N] & [w2 < N] & [x2 < N] & [x3 < N] :
        [x2 + 1] & [w2 + 1] & [w3 + 1] & [x3 + 1] & [z1 - 1] & [w1 - 1] : 0.4 ; //synch123;
    t3_1 :  : [y3 + 1] & [x3 - 1] : 0.39 ; // redo3
    t3_3 :  : [z3 + 1] & [x3 - 1] : 0.91 ; // ok3
    t3_2 :  : [x3 + 1] & [y3 - 1] : 0.3 ;  // back3
    t4_1 :  : [y4 + 1] & [x4 - 1] : 0.33 ; // redo4
    t4_2 :  : [x4 + 1] & [y4 - 1] : 0.3 ;  // back4
    t4_3 :  : [z4 + 1] & [x4 - 1] : 0.77 ; // ok4
    t5 :  : [z4 - 1] & [w4 - 1] : 0.9 ;    // out4;

}
```

Polling Server System (PSS)

An SPN of a polling server [65] and two clients. In the figure, the grey colored places `s` and `a` are logical places and model the server instance. The places `si` represent each a client which either needs service (a token on `si`) or not (`si` empty). The model is scalable by increasing the number of clients which means to change the net structure. A recommendable approach to define a scalable polling server system is to deploy colored Petri nets [82] which are also supported by Snoopy. For the experiments I used a model with 20 and 15 clients, respectively.

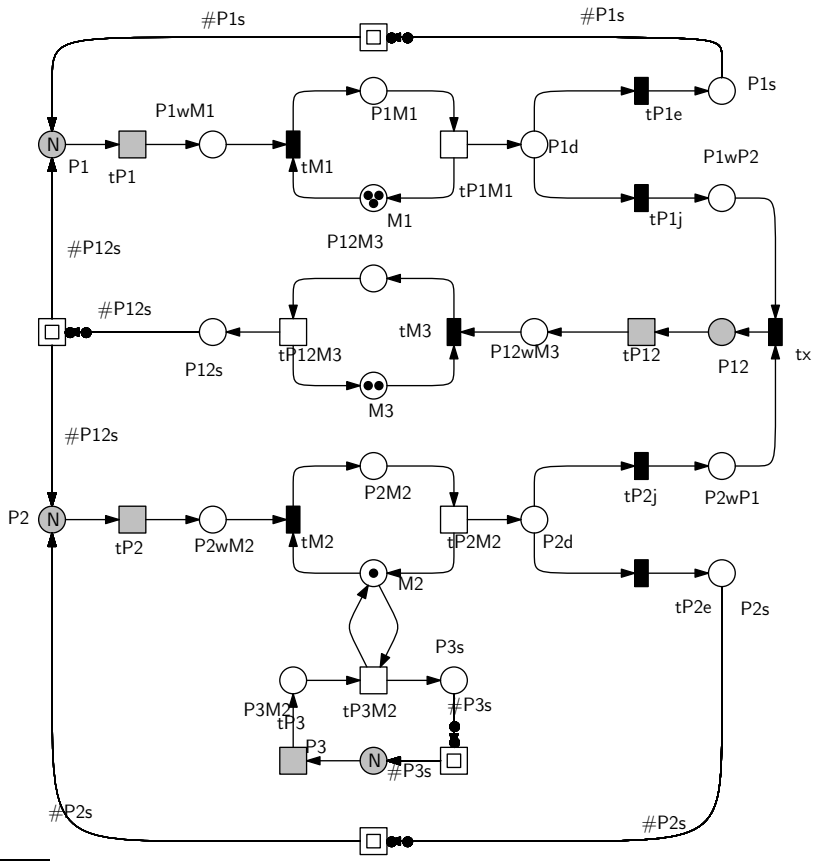



```
 spn [ polling_N2 ] {

constants:

  int N =2 ;
  double gamma = 200 ;
  double mu = 1.0 ;
  double lambda = mu / N;

  places:
    s = 1 ;
    a = 0 ;
    s1 = 0 ;
    s2 = 0 ;

  transitions:

    loop1a : [ s1 < 1 ] & [ a < 1 ] & [ s = 1 ] : [ s + 1 ] : gamma;
    serve1 : [ s = 1 ] : [ s1 - 1 ] & [a - 1 ] & [ s + 1 ] : mu  ;
    loop1b : [ s1 = 1 ] & [ a < 1 ] & [ s = 1 ] : [ a + 1 ] :gamma ;
    t1 : [ s1 < 1  ] : [ s1  + 1 ]  : lambda ;
    loop2a : [ s2 < 1 ] & [ a < 1 ] & [ s = 2 ]: [ s - 1 ] : gamma ;
    serve2 : [s = 2 ]  : [ s2 - 1 ] & [ a -1 ] & [ s - 1 ] : mu  ;
    loop2b : [ s2 = 1 ] & [ a < 1 ] & [ s = 2 ] : [ a + 1 ] : gamma ;
    t2 : [ s2 < 1  ] : [ s2  + 1 ]  : lambda ;

}
```

Flexible Manufacturing System (FMS)

A GSPN model of the flexible manufacturing system with four machines taken from [35]. The model is scalable concerning the initial states by defining different amount of tokens on the places `P1,P2,P3` and `P12`. For the experiments I used the stochastically equivalent SPN[2]. The original FMS system contains transitions which have adjacent arcs with state-dependent weights (`tP1s,tP12s,tP2s,tP3s`). For instance the firing of transition `tP1s` consumes all available tokens on place `P1s` and produces the same amount of tokens on place `P1`. *MARCIE* does not support state-dependent arcs weights. However, assuming an upper bound *max* for the range of `N` we can introduce `max` different transitions for `tP1s`. Each transition `P1s_i` represents the case that there are exactly `i` tokens on place `P1s` which will be consumed and created on `P1`. Therefor we need only standard and inhibitor arcs. The same adaption has to be applied for the transitions `tP12s,tP2s` and `tP3s`.

```
gspn [FMS_GSPN_Nless10] {

constants:
    int N = 10;
    int np = 3*N/2;

places:
    P1 = N;
    P1wM1 = 0;
    P1M1 = 0;
    M1 = 3;
    P1s = 0;
    P12s = 0;
    P12M3 = 0;
    M3 = 2;
    P12wM3 = 0;
    P12 = 0;
    P1wP2 = 0;
    P2wP1 = 0;
    P2 = N;
    P2wM2 = 0;
    P2M2 = 0;
    M2 = 1;
    P2s = 0;
    P3 = N;
    P3M2 = 0;
    P3s = 0;
    P1d = 0;
    P2d = 0;
```

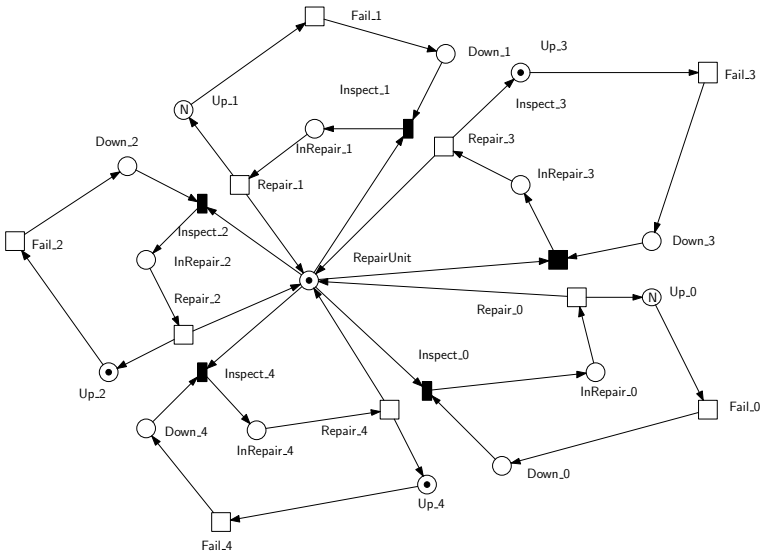


[2]available on request

```
transitions:
    tP3 : [P1] & [P2] & [P12] : [P3M2 + 1] & [P3 - 1] : P3*max(1,np/(P1+P2+P3+P12));
    tP3M2 :  : [M2 + 1] & [P3s + 1] & [M2 - 1] & [P3M2 - 1] : 0.5;
    tP1 : [P2] & [P12] & [P3] : [P1wM1 + 1] & [P1 - 1] : P1*max(1,np/(P1+P2+P3+P12));
    tP2 : [P1] & [P12] & [P3] : [P2wM2 + 1] & [P2 - 1] : P2*max(1,np/(P1+P2+P3+P12));
    tP12 : [P1] & [P2] & [P3] : [P12wM3 + 1] & [P12 - 1] : P12*max(1,np/(P1+P2+P3+P12));

    P1s_eq_1 : [P1s = 1] : [P1 + 1] & [P1s - 1] : 1/60;
    P1s_eq_2 : [P1s = 2] : [P1 + 2] & [P1s - 2] : 1/60;
    ...
    P1s_eq_10 : [P1s = 10] : [P1 + 10] & [P1s - 10] : 1/60;

    P12s_eq_1 : [P12s = 1] : [P1 + 1] & [P2 + 1] & [P12s - 1] : 1/60;
    P12s_eq_2 : [P12s = 2] : [P1 + 2] & [P2 + 2] & [P12s - 2] : 1/60;
    ...
    P12s_eq_10 : [P12s = 10] : [P1 + 10] & [P2 + 10] & [P12s - 10] : 1/60;

    P3s_eq_1 : [P3s = 1] : [P3 + 1] & [P3s - 1] : 1/60;
    P3s_eq_2 : [P3s = 2] : [P3 + 2] & [P3s - 2] : 1/60;
    ...
    P3s_eq_10 : [P3s = 10] : [P3 + 10] & [P3s - 10] : 1/60;

    P2s_eq_1 : [P2s = 1] : [P2 + 1] & [P2s - 1] : 1/60;
    P2s_eq_2 : [P2s = 2] : [P2 + 2] & [P2s - 2] : 1/60;
    ...
    P2s_eq_10 : [P2s = 10] : [P2 + 10] & [P2s - 10] : 1/60;

  immediate:
    tM1 :  : [P1M1 + 1] & [P1wM1 - 1] & [M1 - 1] : 1;
    tP1e :  : [P1s + 1] & [P1d - 1] : 0.8;
    tP1j :  : [P1wP2 + 1] & [P1d - 1] : 0.2;
    tx :  : [P12 + 1] & [P1wP2 - 1] & [P2wP1 - 1] : 1;
    tM3 :  : [P12M3 + 1] & [P12wM3 - 1] & [M3 - 1] : 1;
    tM2 :  : [P2M2 + 1] & [P2wM2 - 1] & [M2 - 1] : 1;
    tP2j :  : [P2wP1 + 1] & [P2d - 1] : 0.4;
    tP2e :  : [P2s + 1] & [P2d - 1] : 0.6;
}
```

Workstation Cluster (WC)

A GSPN model of a workstation cluster taken from [56]. The given net has been created by flattening a colored Petri net. The single components are structurally identical and represent two sets of workstations (suffix 0 and 1) connected by two switches (suffix 2 and 3) and a backbone. Each component may fail because of an error and only a single repair unit is available for repair. The model is scalable concerning the initial number of tokens N on the places `Up_0` and `Up_1`, which represent the number of available workstations. For the experiments I deploy the stochastically equivalent SPN[3].




```
gspn [ WC ] {
constants:
    int N;
    double bb_fail = 0.0002;
    double ws_fail = 0.002;
    double s_fail =  0.00025;
places:
    Down_0 = 0;
    Down_1 = 0;
    Down_2 = 0;
    Down_3 = 0;
    Down_4 = 0;
    InRepair_0 = 0;
    InRepair_1 = 0;
    InRepair_2 = 0;
    InRepair_3 = 0;
    InRepair_4 = 0;
    RepairUnit = 1;
    Up_0 = N;
    Up_1 = N;
    Up_2 = 1;
    Up_3 = 1;
    Up_4 = 1;
transitions:
    Fail_0 :  : [Down_0 + 1] & [Up_0 - 1] : ws_fail*Up_0;
    Fail_1 :  : [Down_1 + 1] & [Up_1 - 1] : ws_fail*Up_1;
    Fail_2 :  : [Down_2 + 1] & [Up_2 - 1] : s_fail;
    Fail_3 :  : [Down_3 + 1] & [Up_3 - 1] : s_fail;
    Fail_4 :  : [Down_4 + 1] & [Up_4 - 1] : bb_fail;
    Repair_0 :  : [Up_0 + 1] & [RepairUnit + 1] & [InRepair_0 - 1] : 2;
    Repair_1 :  : [Up_1 + 1] & [RepairUnit + 1] & [InRepair_1 - 1] : 2;
    Repair_2 :  : [Up_2 + 1] & [RepairUnit + 1] & [InRepair_2 - 1] : 0.25;
    Repair_3 :  : [Up_3 + 1] & [RepairUnit + 1] & [InRepair_3 - 1] : 0.25;
    Repair_4 :  : [Up_4 + 1] & [RepairUnit + 1] & [InRepair_4 - 1] : 0.125;
  immediate:
    Inspect_0 :  : [InRepair_0 + 1] & [Down_0 - 1] & [RepairUnit - 1] : 1;
    Inspect_1 :  : [InRepair_1 + 1] & [Down_1 - 1] & [RepairUnit - 1] : 1;
    Inspect_2 :  : [InRepair_2 + 1] & [Down_2 - 1] & [RepairUnit - 1] : 1;
    Inspect_3 :  : [InRepair_3 + 1] & [Down_3 - 1] & [RepairUnit - 1] : 1;
    Inspect_4 :  : [InRepair_4 + 1] & [Down_4 - 1] & [RepairUnit - 1] : 1;
}
```

[3] available on request

Bibliography

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[3] R. Alur and T. A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[4] O. Andrei and M. Calder. A Model and Analysis of the AKAP Scaffold. *Electronic Notes in Theoretical Computer Science*, 268:3–15, 2010.

[5] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Proc. International Conference Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.

[6] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous time Markov chains. *Transactions on Computational Logic (TOCL)*, 1(1), 2000.

[7] C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic Model Checking for Probabilistic Processes. In *Proc. International Colloquium Automata, Languages and Programming (ICALP 1997)*, volume 1256 of *LNCS*, pages 430–440. Springer, 1997.

[8] C. Baier, L. Cloth, B. R. Haverkort, H. Hermanns, and J. P. Katoen. Performability assessment by model checking of Markov reward models. *Formal Methods in System Design*, 36(1):1–36, 2010.

[9] C. Baier, B. Haverkort, H. Hermanns, and J. P. Katoen. Model checking Contiuous-Time Markov Chains by transient Analysis. In *Proc. International Conference Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 358–372. Springer, 2000.

[10] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. On the Logical Characterisation of Performability Properties. In *Automata, Languages*

and Programming, volume 1853 of *LNCS*, pages 780–792. Springer, 2000.

[11] C. Baier, B. Haverkort, H. Hermanns, and J. P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[12] C. Baier, H. Hermanns, and J. P. Katoen. Approximative Symbolic Model checking of Contiuous-Time Markov Chains. In *Proc. International Conference of Concurrency Theory (CONCUR)*, volume 1664 of *LNCS*, pages 163–175. Springer, 1999.

[13] C. Baier, H. Hermanns, J. P. Katoen, and B. R. Haverkort. Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. *Theoretical Computer Science*, 345(1):2–26, 2005.

[14] A. Bell and B. R. Haverkort. Distributed disk-based algorithms for model checking very large Markov chains. *Formal Methods in System Design*, 29(2):177–196, 2006.

[15] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W. J. Stewart. The PEPS Software Tool. In *Computer Performance Evaluations, Modelling Techniques and Tools*, volume 2794 of *LNCS*, pages 98–115. Springer, 2003.

[16] D. Bosnaki, S. Edelkamp, D. Sulewski, and A. Wijs. Parallel probabilistic model checking on general purpose graphics processors. *Software Tools for Technology Transfer (STTT)*, 13:21–35, 2011.

[17] K. S. Brace and R. L. R. R. E. Bryant. Efficient implementation of a BDD package. In *Proc. International Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.

[18] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[19] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of Memory-Efficient Kronecker Operations with Applications to the solution of markov models. *INFORMS Journal on Computing*, 12(3):203–222, 2000.

[20] P. Buchholz, J. P. Katoen, P. Kemper, and C. Tepper. Model-checking large structured Markov chains. *Journal of Logic and Algebraic Programming*, 56(1-2):69–97, 2003.

[21] P. Buchholz and P. Kemper. A Toolbox for the Analysis of Discrete Event Dynamic Systems. In *Proc. International Conference Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 483–486. Springer, 1999.

[22] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^20 States and Beyond. In *Proc. IEEE Symposium on Logic in Computer Sience*, volume 5, pages 428–439, 1990.

[23] M. Calder, V. Vyshemirsky, D. Gilbert, and R. Orton. Analysis of signalling pathways using continuous time Markov chains. *Transactions on Computational Systems Biology*, 4220:44–67, 2006.

[24] D. Cerotti, S. Donatelli, A. Horváth, and J. Sproston. CSL Model Checking for Generalized Stochastic Petri Nets. In *Proc. International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 199–210. IEEE Computer Society, 2006.

[25] G. Chiola, S. Donatelli, and G. Franceschinis. GSPNs versus SPNs: what is the actual role of immediate transitions? . In *Proc. Internatinal Workshop Petri Nets and Performance Models (PNPM)*, pages 20–31, 1991.

[26] K. H. Cho, S. Y. Shin, H. W. Kim, O. Wolkenhauer, B. McFerran, and W. Kolch. Mathematical modeling of the influence of RKIP on the ERK signaling pathway. In *Proc. International Conference on Computational Methods in Systems Biology (CMSB)*, volume 2602 of *LNCS/LNBI*, pages 127–141. Springer, 2003.

[27] G. Ciardo. Data Representation and Efficient Solution: A Decision Diagram Approach. In *Formal Methods for Performance Evaluation (SFM 2007)*, volume 4486 of *LNCS*, pages 371–394. Springer, 2007.

[28] G. Ciardo, A. Blakemore, P. F. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using Stochastic Reward Nets. *IMA Volumes in Mathematics and its Applications: Linear Algebra, Markov Chains, and Queueing Models*, 48:145–191, 1993.

[29] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Performance Evaluation*, 63(1):578–608, 2006.

[30] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 328–342. Springer, 2001.

[31] G. Ciardo, G. Lüttgen, and A. J. Yu. Improving static variable orders via invariants. In *Proc. Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN*, volume 4546 of *LNCS*, pages 83–103.

Springer, 2007.

[32] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. International Workshop on Petri Nets and Performance Models (PNPM)*, pages 22–31. IEEE Computer Society Press, 1999.

[33] G. Ciardo and R. Siminiceanu. Using Edge-Valued Decision Diagrams for Symbolic Generation of Shortest Paths. In *Proc. International Conference on Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 256–273. Springer, 2002.

[34] G. Ciardo and M. Tilgner. On the Use of Kronecker Operators for the Solution of Generalized Stocastic Petri Nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, 1996.

[35] G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Reward Net Models. *Performance Evaluation*, 18(1):37–59, 1993.

[36] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1982.

[37] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[38] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.

[39] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Proc. International Design Automation Conference*, DAC '93, pages 54–60. ACM, 1993.

[40] L. Cloth. *Model Checking Algorithms for Markov Reward Models*. PhD thesis, University of Twente, 2006.

[41] L. Cloth and B. R. Haverkort. Model checking for survivability! In *Quantitative Evaluation of Systems*, pages 145–154. IEEE Computer Society, 2005.

[42] L. Cloth and B. R. Haverkort. Five Performability Algorithms. A Comparison. In *MAM 2006: Markov Anniversary Meeting*, pages 39–54. Boson Books, 2006.

[43] T. Courtney, S. Gaonkar, K. Keefe, E. Rozier, and W. H. Sanders. Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In *DSN*, pages 353–358, 2009.

[44] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*, pages 395–410. Springer, 2000.

[45] D. Deavours and W. H. Sanders. "On-the-Fly" Solution Techniques for Stochastic Petri Nets and Extensions. In *IEEE Transactions on Software Engineering*, pages 132–141, 1997.

[46] D. D. Deavours and W. H. Sanders. An efficient disk-based tool for solving large Markov models. *Performance Evaluation*, 33(1):67–84, 1998.

[47] S. Donatelli. Superposed Generalized Stochastic Petri Nets: Definition and Efficient Solution. In *Proc. International Conference on Application and Theory of Petri Nets*, volume 815 of *LNCS*, pages 258–277. Springer, 1994.

[48] S. Donatelli, M. Ribaudo, and J. Hillston. A Comparison of Performance Evaluation Process Algebra and Generalized Stochastic Petri Nets. In *Proc. International Workshop on Petri Nets and Performance Models*, pages 158–168. IEEE Computer Society Press, 1995.

[49] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. In *Computer Aided Verification*, CAV '90, pages 136–145, London, UK, UK, 1991. Springer.

[50] B. L. Fox and P. W. Glynn. Computing Poisson probabilities. *Communications of the ACM*, 31:440–445, 1988.

[51] D. Gilbert, M. Heiner, and S. Lehrack. A Unifying Framework for Modelling and Analysing Biochemical Pathways Using Petri Nets . In *Proc. International Conference on Computational Methods in Systems Biology*, volume 4695 of *Lecture Notes in Computer Sience/LNBI*, pages 200–216. Springer, 2007.

[52] P. J. E. Goss and J. Peccoud. Quantitative modeling of stochastic systems in molecular biology by using stochastic Petri nets. *Proc. of the National Acadamy of Science of the United States of America*, 95:2340–2361, 1998.

[53] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Probabilistic analysis of large finite state machines. In *Proc. Design Automation Conference*,

DAC '94, pages 270–275, New York, NY, USA, 1994. ACM.

[54] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

[55] B. Haverkort, L. Cloth, H. Hermanns, J.-P. Katoen, and C. Baier. Model Checking Performability Properties. In *Dependable Systems and Networks*, DSN '02, pages 103–112. IEEE Computer Society, 2002.

[56] B. R. Haverkort, H. Hermanns, and J. P. Katoen. On the Use of Model Checking Techniques for Dependability Evaluation. In *Symposium on Reliable Distributed Systems*, SRDS '00, pages 228–237. IEEE Computer Society, 2000.

[57] B. R. Haverkort and K. S. Trivedi. Specification techniques for Markov reward models. *Discrete Event Dynamic Systems*, 3:219–247, 1993.

[58] M. Heiner, M. Herajy, F. Liu, C. Rohr, and M. Schwarick. Snoopy – a unifying Petri net tool. In *Proc. Application and Theory of Petri Nets*, volume 7347 of *LNCS*, pages 398–407. Springer, 2012.

[59] M. Heiner, C. Rohr, and M. Schwarick. MARCIE - Model checking And Reachability analysis done effiCIEntly. In *Proc. PETRI NETS 2013*, volume 7927 of *LNCS*, pages 389–399. Springer, 2013.

[60] M. Heiner, C. Rohr, M. Schwarick, and S. Streif. A Comparative Study of Stochastic Analysis Techniques . In *Proc. International Conference on Computational Methods in Systems Biology (CMSB)*, pages 96–106. ACM digital library, 2010.

[61] M. Heiner, M. Schwarick, and A. Tovchigrechko. DSSZ-MC-A Tool for Symbolic Analysis of Extended Petri Nets. In *Application and Theory of Petri Nets*, volume 5606 of *LNCS*, pages 323–332, 2009.

[62] H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1-2):43–87, 2002.

[63] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In *Proc. International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[64] J. Hillston. *A compositional approach to performance modelling*. Cambridge University Press, New York, NY, USA, 1996.

[65] O. Ibe and K. Trivedi. Stochastic Petri Net Models of Polling Systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[66] D. N. Jansen, J.-P. Katoen, M. Oldenkamp, M. Stoelingan, and Zapreev. How Fast and Fat Is Your Probabilistic Model Checker? In *Haifa Verification Conference 2007*, LNCS, pages 69–85. Springer, 2008.

[67] K Lampka. *A symbolic approach to the state graph based analysis of high-level Markov reward models*. PhD thesis, Universität Erlangen-Nürnberg, 2007.

[68] J. P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and Symbolic CTMC Model Checking. In *Proc. International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'01)*, volume 2165 of *LNCS*, pages 23–38. Springer, 2001.

[69] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The Ins and Outs of The Probabilistic Model Checker MRMC. In *Proc. International Conference on Quantitative Evaluation of Systems (QEST)*, pages 167–176. IEEE Computer Society, 2009.

[70] P. Kemper. Numerical Analysis of Superposed GSPNs. *IEEE Transactions on Software Engineering*, 22(9):615–628, 1996.

[71] P. Kemper. Parallel Randomization for Large Structured Markov Chains. In *Proc. International Conference on Dependable Systems and Networks*, DSN '02, pages 657–668. IEEE Computer Society, 2002.

[72] P. Kemper and R. Lübeck. Model checking based on Kronecker algebra. Technical report, Universität Dortmund, Fachbereich Informatik, 1998.

[73] W. J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine. University of London., 1999.

[74] W. J. Knottenbelt and P. G. Harrison. Distributed Disk-based Solution Techniques for Large Markov Models. In *Proc. International Workshop on the Numerical Solution of Markov Chains (NSMC)*, pages 58–75, 1999.

[75] M. Kwiatkowska, G. Norman, and A. Pacheco. Model checking expected time and expected reward formulae with random time bounds. *Computers and Mathematics with Applications*, 51(2):305–316, 2006.

[76] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic Model Checking. In *Formal Methods for the Design of Computer, Communication and Soft-*

ware Systems: Performance Evaluation (SFM'07), volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

[77] M. Kwiatkowska, G. Norman, and D. Parker. Using probabilistic model checking in systems biology. *ACM SIGMETRICS Performance Evaluation Review*, 35(4):14–21, 2008.

[78] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[79] M. Kwiatkowska, D. Parker, Y. Zhang, and R. Mehmood. Dual-Processor Parallelisation of Symbolic Probabilistic Model Checking. In *Proc. International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 123–130. IEEE Computer Society Press, 2004.

[80] Y. T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. International Design Automation Conference*, DAC '92, pages 608–613. IEEE Computer Society Press, 1992.

[81] K. Lautenbach and H. Ridder. A Completion of the S-invariance Technique by means of Fixed Point Algorithms. Technical report, Universität Koblenz-Landau, 1995.

[82] F. Liu. *Colored Petri Nets for Systems Biology* . PhD thesis, BTU Cottbus, Dep. of CS, 2012.

[83] V. Maisonneuve. Automatic heuristic-based generation of MTBDD variable orderings for PRISM models. Internship report, ENS Cachan & Oxford University Computing Laboratory, 2009.

[84] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing, John Wiley and Sons, 1995. 2nd Edition.

[85] M. A. Marsan, G. Conte, and G. Balbo. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *Transactions on Computer Systems*, 2(2):93–122, 1984.

[86] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[87] R. Mehmood. *Disk-based techniques for efficient solution of large Markov chains*. PhD thesis, University of Birmingham, 2004.

[88] R. Mehmood, D. Parker, and M. Kwiatkowska. An Efficient BDD-Based Implementation of Gauss-Seidel for CTMC Analysis. Technical Report CSR-03-13, School of Computer Science, University of Birmingham, 2003.

[89] J. F. Meyer. Performability: A Retrospective and Some Pointers to the Future. *Performance Evaluation*, 14(3-4):139–156, 1992.

[90] A. Miner and D. Parker. Symbolic Representations and Analysis of Large State Spaces. In *Proc. Validation of Stochastic Systems*, volume 2925 of *LNCS*, pages 296–338. Springer, 2004.

[91] A. S. Miner. *Data structures for the analysis of large structured markov models*. PhD thesis, The College of William and Mary, 2000.

[92] V. Moler and C. V. Loan. Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. *SIAM Review*, 45(1):3–49, 2003.

[93] M. K. Molloy. Performance Analysis Using Stochastic Petri Nets. *IEEE Transactions on Computers*, 31:913–917, 1982.

[94] L. Napione, D. Manini, F. Cordero, A. Horvath, A. Picco, M. D. Pierro, S. Pavan, M. Sereno, A. Veglio, F. Bussolino, and G. Balbo. On the Use of Stochastic Petri Nets in the Analysis of Signal Transduction Pathways for Angiogenesis Process. In *Proc. International Conference on Computational Methods in Systems Biology*, volume 5688 of *LNCS/LNBI*, pages 281–295. 5688, Springer, 2009.

[95] Noack. A ZBDD Package for Efficient Model Checking of Petri Nets (in German). Technical report, BTU Cottbus, Dep. of CS, 1999.

[96] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[97] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. *ACM SIGMETRICS Performance Evaluation Review*, 13:147–154, 1985.

[98] A. Pnueli. The Temporal Logic of Programs. In *Symposium on the Foundations of Computer Science*, volume 18, pages 46–57. IEEE Computer Society Press, 1977.

[99] L. Priese and H. Wimmel. *Theoretische Informatik: Petri-Netze*. Springer, 2002.

[100] A. Reibman and K. S. Trivedi. Transient Analysis of Cumulative Mea-

sures of Markov Model Behavior. *Communications in Statistics-Stochastic Models*, 5:683–710, 1989.

[101] A. Remke, B. R. Haverkort, and L. Cloth. Model Checking Infinite-State Markov Chains. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 3440 of *LNCS*, pages 237–252. Springer, 2005.

[102] H. Ridder. *Analysis of Petri Net Models with Decision Diagrams (in German)*. PhD thesis, Universität Koblenz-Landau, 1997.

[103] O. Roig, J. Cortadella, and E. Pastor. Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets. In *Proc. Interntional Conference on Application and Theory of Petri Nets*, volume 935 of *LNCS*, pages 374–391. Springer, 1996.

[104] R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Packag*. Kluwer Academic Publishers, 1996.

[105] M. Schwarick. IDD-MC - a model checker for bounded Stochastic Petri nets. In *Proc. Workshop AWPN*, volume 643, pages 80–87. CEUR-WS, 2010.

[106] M. Schwarick. *Manual: Marcie - An analysis tool for Generalized Stochastic Petri nets*. BTU Cottbus, Dep. of CS, 2010.

[107] M. Schwarick. Symbolic model checking of stochastic reward nets. In *Proc. International Workshop on Concurrency, Specification, and Programming (CS&P 2012)*, volume 928 of *CEUR Workshop*, pages 343–357. CEUR-WS.org, 2012.

[108] M. Schwarick and M. Heiner. CSL model checking of biochemical networks with Interval Decision Diagrams . In *Proc. International Conference on Computational Methods in Systems Biology*, volume 5688 of *LNCS/LNBI*, pages 296–312. Springer, 2009.

[109] M. Schwarick, C. Rohr, and M. Heiner. MARCIE – Model checking And Reachability analysis done effiCIEntly. In *Proc. International Conference on Quantitative Evaluation of SysTems (QEST 2011)*, pages 91–100. IEEE Computer Society Pres, 2011.

[110] M. Schwarick and A. Tovchigrechko. IDD-based model validation of biochemical networks. *Theoretical Computer Sience*, 412:2884–2908, 2010.

[111] R. I. Siminiceanu and Ciardo. New metrics for static variable ordering in

decision diagrams. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 3920 of *LNCS*, pages 328–342. Springer, 2006.

[112] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[113] K. Strehl and L. Thiele. Symbolic Model Checking Using Interval Diagram Techniques. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, 1998.

[114] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[115] A. Tovchigrechko. *Model Checking Using Interval Decision Diagrams*. PhD thesis, BTU Cottbus, Dep. of CS, 2008.

[116] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition, 2002.

[117] J. Vilar, H.-Y. Kueh, N. Barkai, and S. Leibler. Mechanisms of Noise-Resistance in Genetic Oscillators. *Proc. National Academy of Sciences of the United States of America*, 99(9):5988–5992, 2002.

[118] A. Williams. *C++ Concurrency in Action; Practical Multithreading*. Manning Publications Co., Shelter Island, USA, 2012.

[119] T. Yoneda, H. Hatori, A. Takahara, and S. Minato. BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of Petri Nets. In *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*, pages 435–449. Springer, 1996.