

Simulative Analysis of Coloured Extended Stochastic Petri Nets

Von der Fakultät für MINT - Mathematik, Informatik, Physik,
Elektro- und Informationstechnik
der Brandenburgischen Technischen Universität
Cottbus–Senftenberg
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Diplominformatiker

Christian Rohr

geboren am 07.01.1979 in Altdöbern

Gutachter: Prof. Dr.-Ing. Monika Heiner

Gutachter: Prof. Dr. rer. nat. Wolfgang Marwan

Gutachter: Prof. Dr. David Gilbert

Tag der mündlichen Prüfung: 10.01.2017

Thanks to all who made it possible.

Zusammenfassung

Stochastische Modellierung von biochemischen Reaktionsnetzwerken wird immer populärer. Im Laufe der letzten Jahrzehnte hat sich sowohl die Größe als auch die Komplexität typischer biologischer Modelle immer weiter erhöht, der Grund dafür sind die Fortschritte in System- und Molekularbiologie, insbesondere durch die Hochdurchsatz-Technologien. Hier werden biochemische Netzwerke unterschiedlicher Detailebenen modelliert, beginnend mit einfachen chemischen Reaktionen und Signaltransduktionsnetzwerken, bis zu einzelnen Zellen und ganzen Organismen. Ein zweiter Bereich von Interesse ist die Entwicklung von zuverlässigen Hard- und Softwaresystemen, die zunehmend an Bedeutung in unserer hochtechnisierten Gesellschaft gewinnen, weil IT-Systeme heute in allen Bereichen eingesetzt werden. Validierung und Zuverlässigkeitsbewertung des Systems sollte bereits in der Planungsphase durchgeführt werden, wenn es sich um kritische Systeme wie in der Medizin, Automobilindustrie oder der Kontrolle von Industrieanlagen handelt, um sicherzustellen, dass definierte Toleranzen eingehalten werden oder keine ungültigen Systemzustände erreicht werden können. Dadurch erhöht sich die Nachfrage nach leistungsfähigen Analyseverfahren. Eine Modellierungsumgebung basierend auf Petri-Netzen ist in der Lage, alle diese Szenarien zu bedienen.

Ein Petri-Netz ist eine mathematische Modellierungssprache für die Beschreibung des nebenläufigen Verhaltens von verteilten Systemen. Ein Vorteil ist die einfache Skalierbarkeit der Modelle sowohl in Bezug auf die Struktur des Netzes als auch auf ihren Zustandsraum. Um die Zuverlässigkeit des Modells und damit des modellierten Systems sicherzustellen ist es notwendig, mehrere Analysen durchzuführen. Es gibt etablierte Methoden zur qualitativen und quantitativen Analyse von Petri-Netzen. Zu den qualitativen Analysen zählen die Berechnung von Invarianten, und wenn möglich, die Erzeugung des Erreichbarkeitsgraphen. Mit Hilfe dessen die temporalen Logiken Computation Tree Logic (CTL) oder Linear-time Temporal Logic (LTL) angewendet werden können, beispielsweise um die Erreichbarkeit bestimmter Zustände zu gewährleisten. All dies geschieht jedoch unabhängig von der Zeit. So kann man nur Aussagen machen, ob ein Ereignis eintritt oder nicht. Aber man kann nicht sagen, wie wahrscheinlich es ist. Dieser Aspekt wurde mit der Einführung von Zeit in den stochastischen Petri-Netzen berücksichtigt. Jetzt können

Wahrscheinlichkeiten für das Auftreten von bestimmten Ereignissen oder die Erreichbarkeit von Zuständen bestimmt werden. Numerische Methoden, die den gesamten Zustandsraum des Petri-Netzes in Betracht ziehen, stehen für diese quantitative Analyse zur Verfügung. Ein begrenzter Zustandsraum mit bis zu 1×10^9 Zuständen (nach aktuellem Stand der Speichertechnologie) ist eine Voraussetzung dafür. Systeme mit mehr Zuständen oder sogar unbegrenztem Zustandsraum können numerisch nicht innerhalb einer angemessenen Zeit analysiert werden. An diesem Punkt kann eine Analyse nur durch stochastische Simulationsalgorithmen durchgeführt werden.

In dieser Arbeit stellen wir verschiedene stochastische Simulationsalgorithmen vor, sowohl exakte wie auch approximative Verfahren. Darüber hinaus stellen wir einen Ansatz vor, um die Effizienz der stochastischen Simulation für große und dichte Netzwerke durch einen neuen approximativen stochastischen Simulation Algorithmus, genannt *discrete-time leap method*, zu verbessern. Wir zeigen das breite Spektrum der simulativen Analysen komplexer stochastischer Systeme von der Pfaderzeugung bis zur Berechnung von transienten Lösungen und stationärer Verteilungen. Wir legen erweiterte Analysen von stochastischen Modellen mit Hilfe von simulativer Modellprüfung dar. Wir integrieren die Continuous Stochastic (Reward) Logic (CS(R)L) und die Probabilistic Linear-time Temporal Logic with constraints (PLTLc) für die simulative Modellprüfung. Simulative Modellprüfung hat einige Einschränkungen im Vergleich zu den numerischen Methoden, so ist es im Prinzip möglich, verschachtelte probabilistische Formeln in CS(R)L zu betrachten, aber es ist nicht praktikabel, da die Berechnung in einer angemessenen Zeitspanne nicht möglich ist. Neben der Transientenanalyse ist oftmals die Analyse der stationären Verteilung von Interesse, deshalb haben wir zwei Methoden zur Erkennung des stationären Zustandes implementiert. Die erste basiert auf einem “sample batch means” Algorithmus und wird in der linearen temporalen Logik verwendet. Die zweite approximiert die stationäre Verteilung und prüft auf Konvergenz. Wir wenden die oben genannten Techniken auf mehrere Fallstudien aus Systembiologie und technischen Systemen an.

Der größte Beitrag dieser Arbeit zum aktuellen Stand von Wissenschaft und Technik liegt in der Entwicklung der *discrete-time leap method* zur Simulation stochastischer Modelle, der Approximationen transienter Lösungen und statio-

närer Verteilungen mittels stochastischer Simulation für stochastische Modelle und Markov-Rewardmodelle, der Entwicklung eines nicht zeitbeschränkten Algorithmus zur simulativen Modellprüfung, der die stationäre Eigenschaft für PLTLc und CSL ausnutzt, und des ersten Algorithmus zur simulativen Modellprüfung für CSRL, der Zustands- und Impulsrewards enthält. Alle vorgestellten Algorithmen und Methoden sind in dem Analyse-Tool MARCIE implementiert.

Schlagwörter: gefärbte erweiterte stochastische Petri Netze, stochastische Simulation, δ -leaping, simulatives Modelchecking

Abstract

Stochastic modelling of biochemical reaction networks is getting more and more popular. Throughout the past decades typical biological models increased in their size and complexity, because of advances in systems and molecular biology, in particular through the high-throughput omic technologies. Here biochemical networks of different levels of detail are modelled, starting with simple chemical reactions and signal transduction networks, up to individual cells and entire organisms. A second area of interest is the design of reliable hardware and software systems, which is becoming increasingly important in our highly technological society, because IT systems are now used in all areas. Validation and reliability assessment of the system should be done already in the design phase, if it is too critical systems such as in medicine, the automotive industry or the control of industrial plants, in order to ensure that defined tolerances are met or no invalid system states can be reached. This increases the demand for efficient analysis methods. A modelling framework based on Petri nets is able to serve all of these scenarios.

A Petri net is a mathematical modelling language for the description of concurrent behaviour of distributed systems. Its advantage is the ease of scalability of the models, which relates to the network's state space, as well as the structure of the network itself. To ensure the reliability of the model and therefore of the modelled system, it is necessary to perform several analyses. There are established methods for qualitative and quantitative analysis of Petri nets. The qualitative analyses include the calculation of invariants, and if possible the generation of the reachability graph, by means of which temporal logics such as computation tree logic (CTL) or linear-time temporal logic (LTL) can be applied, for example, to ensure reachability of certain states. All of this happens, however, independent of time. So one can only make statements if an event occurs or not. But one can not say how likely it occurs. With the introduction of time in the stochastic Petri nets this aspect was taken into account. Now, probabilities can be determined for the occurrence of certain events or the reachability of states. Numerical methods, which consider the entire state space of the Petri net, are available for this quantitative analysis. A bounded state space with up to 1×10^9 states (the state of the current memory technology) is a prerequisite for this. Systems with more states, or even

unbounded state space can not be analysed numerically within a reasonable amount of time. At this point, an analysis can only be carried out by means of stochastic simulation algorithms.

In this work, we recall several stochastic simulation algorithms, e.g., exact as well as approximate methods. Furthermore, we introduce an approach to improve the efficiency of stochastic simulation for large and dense networks by a new approximate stochastic simulation algorithm called *discrete-time leap method*. We depict the wide range of simulative analyses of complex stochastic systems ranging from trace generation to the computation of transient solutions and steady state distributions. We set forth advanced analysis of stochastic models by means of simulative model checking. For the use of simulative model checking, we integrate the continuous stochastic (reward) logic (CS(R)L) and the probabilistic linear-time temporal logic with constraints (PLTLc). Simulative model checking has some limitations compared to the numerical methods, e.g., in principle it is possible to consider nested probabilistic formulas in CS(R)L, but not practical, since the calculation is not feasible in a reasonable period of time. In addition to the transient analysis, the steady state analysis is often of interest, therefore we have implemented two on-the-fly steady state detection methods. The first one is based on a “sample batch means” algorithm and is used in the linear-time temporal logic. The second approximates the steady state distribution and checks for convergence. We apply the aforementioned techniques to several case studies from systems biology and technical systems.

The main contributions of this thesis to scientific knowledge are the development of the *discrete-time leap method* for the simulation of stochastic models, the approximations of transient solutions and steady state distributions by use of stochastic simulation for stochastic models and Markov reward models, the development of an infinite time horizon model checking algorithm exploiting the steady state property for PLTLc and CSL, and the first simulative model checking algorithm for CSRL incorporating state and impulse rewards. All presented algorithms and methods are implemented in the advanced analysis tool MARCIE.

Keywords: coloured eXtended Stochastic Petri Nets, stochastic simulation, δ -leaping, simulative model checking

Contents

1	Introduction	1
2	Preliminaries	7
2.1	Petri Net	7
2.2	Reachability Graph	12
2.3	Extended Petri Net	13
2.4	Marking-dependent Extended Petri Net	16
2.5	Stochastic Petri Net	19
2.6	Continuous-Time Markov Chain	21
2.7	Generalised Stochastic Petri Net	22
2.8	Extended Stochastic Petri Net	24
2.9	Coloured Petri Net	26
2.10	Closing Remarks	27
3	Stochastic Simulation	29
3.1	Stochastic Simulation Algorithm	30
3.2	Direct Method	33
3.3	Optimised Direct Method	35
3.4	First Reaction Method	38
3.5	Next Reaction Method	39
3.6	Tau-Leaping Method	39
3.7	Discrete-Time Leap Method	41
3.7.1	Transition firing	42
3.7.2	Dependent Subnets	45
3.7.3	Algorithm	47
3.7.4	Caveat	48

3.8	Extensions	49
3.8.1	Immediate Transitions	50
3.8.2	Deterministic and Scheduled Transitions	51
3.9	Random Number Generation	53
3.10	Closing Remarks	54
4	Simulative Analysis	57
4.1	Trace Generation	58
4.2	Transient Solutions	61
4.3	Steady State Distribution	65
4.4	Observers	69
4.5	Closing Remarks	72
5	Simulative Model Checking	73
5.1	Simulative PLTLc Model Checking	74
5.1.1	Time-bounded Formula	76
5.1.2	Time-unbounded Formula	78
5.1.3	Steady State Operator	82
5.2	Simulative CSL Model Checking	84
5.2.1	Nested Probabilistic Operator	87
5.2.2	Time-bounded Formula	87
5.2.3	Time-unbounded Formula	89
5.2.4	Steady State Operator	91
5.3	Simulative Reward Computation	92
5.4	Simulative CSRL Model Checking	97
5.5	Closing Remarks	100
6	Case Studies	101
6.1	RKIP inhibited ERK pathway	102
6.2	Mitogen-activated Protein Kinase	106
6.3	Angiogenesis	111
6.4	Simplified Repressilator	115
6.5	<i>E.coli</i> K-12 Metabolic model	118
6.5.1	Reduced <i>E.coli</i> K-12 Metabolic model	118
6.5.2	<i>E.coli</i> K-12 Genome Scale Metabolic model	120

6.6	Flexible Manufacturing System	121
6.7	Cyclic Server Polling System	125
6.8	Closing Remarks	131
7	Conclusions and Outlook	133
7.1	Conclusions	133
7.2	Outlook	136
A	Appendix	139
A.1	ANDL Syntax of RKIP inhibited ERK pathway	139
A.2	ANDL Syntax of Mitogen-activated Protein Kinase	141
A.3	ANDL Syntax of Angiogenesis	145
A.4	CANDL Syntax of Repressilator	152
A.5	CANDL Syntax of Flexible Manufacturing System	153
A.6	CANDL Syntax of Cyclic Server Polling System	156
	Bibliography	159

List of Figures

2.1	Elements of a Petri net.	10
2.2	Producer & Consumer as \mathcal{PN}	10
2.3	Reachability graph of Example 1	13
2.4	Additional arc types of an extended Petri net.	15
2.5	Producer & Consumer as \mathcal{XPN}	16
2.6	Reachability Graph of Producer & Consumer \mathcal{XPN}	16
2.7	Producer & Consumer as \mathcal{MXPN}	19
2.8	Reachability Graph of Producer & Consumer \mathcal{MXPN}	19
2.9	Additional elements of an extended stochastic Petri net.	26
2.10	Overview of net classes defined in this section, with the extensions leading from one net class to the other.	28
3.1	Averaged number of tokens of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.	33
3.2	First order reaction: $P1 \rightarrow P2$	44
3.3	Second order reaction: $P1 + P2 \rightarrow P3$	44
3.4	Conflict: $P1 \rightarrow P2$ and $P1 \rightarrow P3$	45
3.5	Sequence: $P1 \rightarrow P2 \rightarrow P3$	46
3.6	Simplified birth-death process, it shows the results for different rate constants of $T2$, i.e., $c_{T2} = 1$ (blue) and $c_{T2} = 0.5$ (green)	48
4.1	Mean and standard deviation over time of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.	60
4.2	Averaged number of transition firings of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.	61
4.3	Transient solutions up to $\tau = 100$ of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.	65

4.4	Averaged waiting time for <i>consumer</i> to receive a token up to $\tau = 100$ of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.	72
5.1	Stochastic Petri net demonstrating the issue of not terminating verification of time-unbounded until formulas.	80
6.1	Stochastic Petri net of the RKIP inhibited ERK pathway, including textual representation of the chemical reactions [HDG10].	103
6.2	\mathcal{SPN}_{ERK} with $N = 100$ and 1 000 000 simulation runs	104
6.3	Transient analysis for different initial markings N of \mathcal{SPN}_{ERK} . The total run-time is given for several number of workers.	105
6.4	Steady state analysis for different initial markings N of \mathcal{SPN}_{ERK} . The total run-time is given for different numbers of workers.	106
6.5	Stochastic Petri net of the mitogen-activated protein kinase [HGD08].	107
6.6	\mathcal{SPN}_{MAPK} with $N = 100$ and 1 000 000 simulation runs	108
6.7	Transient analysis up to time point $\tau = 1$ for different initial markings N of \mathcal{SPN}_{MAPK} . The total run-time is given for several number of worker threads after 6 634 234 simulation runs.	110
6.8	Steady state analysis for different initial markings N of \mathcal{SPN}_{MAPK} . The total run-time is given for different numbers of workers after 128 simulation runs.	110
6.9	Stochastic Petri net of the angiogenesis process [Nap+09].	112
6.10	Transient analysis for different initial markings N of \mathcal{SPN}_{ANG} . The total run-time is given for several number of workers.	113
6.11	Steady state analysis for different initial markings N of \mathcal{SPN}_{ANG} . The total run-time is given for different numbers of workers.	114
6.12	Coloured Stochastic Petri Net of the simplified Repressilator.	115
6.13	Stochastic simulations of the simplified repressilator for 1 copy, 1000 copies per gene and for 1, 1000 simulation runs.	116
6.14	Probabilities of the value ranges on the places p_i up to time point $\tau = 10\,000$	117

6.15	Steady state probability distribution of the number of tokens on place p_1	117
6.16	Petri net (a) and connectivity (b) of the reduced <i>E.coli</i> K-12 core model.	119
6.17	<i>E.coli</i> core with $N = 1000$ and 1 000 000 simulation runs	119
6.18	Petri net (a) and connectivity (b) of the <i>E.coli</i> K-12 genome scale metabolic model.	120
6.19	<i>E.coli</i> K-12 with $N = 100$ and 100 000 simulation runs	121
6.20	Coloured Stochastic Petri net of the flexible manufacturing system.	123
6.21	Transient analysis up to time point $\tau = 1$ for different number of items N of \mathcal{GSPN}_{FMS} . The total run-time is given for several number of workers after 6 634 234 simulation runs.	124
6.22	Steady state analysis for different number of items N of \mathcal{GSPN}_{FMS} . The total run-time is given for different numbers of workers after 128 simulation runs.	125
6.23	Coloured Stochastic Petri Net of the Cyclic Server Polling System.	126
6.24	Transient analysis up to time point $\tau = 10$ for different number of stations N of \mathcal{SPN}^c_{CSPS} . The total run-time is given for several number of workers after 6 634 234 simulation runs.	128
6.25	Steady state analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The total run-time is given for different numbers of workers after 128 simulation runs.	129
6.26	Reward analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The total run-time is given for different numbers of workers after 6 634 234 simulation runs.	130
6.27	Performability analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The total run-time is given for different numbers of workers after 6 634 234 simulation runs.	131
7.1	The peak memory consumption is given for transient analysis up to different time points τ for $N = 10$ stations of \mathcal{SPN}^c_{CSPS} using simulative CSL and PLTLc model checking.	135

List of Tables

2.1	The rate functions and parameters of the „Producer & Consumer” \mathcal{SPN}	21
6.1	The size of the state space for different initial markings of \mathcal{SPN}_{ERK} computed with MARCIE’s symbolic state space generation. . .	102
6.2	Comparison of run-times for the direct method and δ -leaping. \mathcal{SPN}_{ERK} was parametrised with N and simulated with 1 000 000 simulation runs.	104
6.3	Steady state analysis for different initial markings N of \mathcal{SPN}_{ERK} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine.	105
6.4	The size of the state space for different initial markings of \mathcal{SPN}_{MAPK} computed with MARCIE’s symbolic state space generation. . .	106
6.5	Comparison of run-times for the direct method and δ -leaping. \mathcal{SPN}_{MAPK} was parametrised with N and simulated with 1 000 000 simulation runs.	108
6.6	Transient analysis up to time point $\tau = 1$ for different number of stations N of \mathcal{SPN}_{MAPK} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.	109
6.7	Steady state analysis for different number of stations N of \mathcal{SPN}_{MAPK} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 128 simulation runs.	111

6.8	The size of the state space for different initial markings of \mathcal{SPN}_{ANG} computed with MARCIE's symbolic state space generation. The places <i>Akt</i> , <i>DAG</i> , <i>Gab1</i> , <i>KdStar</i> , <i>Pip2</i> , <i>P3k</i> , <i>Pg</i> and <i>Pten</i> carry initially N tokens.	111
6.9	Transient analysis for different initial markings N of \mathcal{SPN}_{ANG} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine.	113
6.10	Steady state analysis for different initial markings N of \mathcal{SPN}_{ANG} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine.	114
6.11	Comparison of run-times for the direct method (a) and δ -leaping (b). \mathcal{SPN}_{CORE} was parametrised with N and simulated with several number of simulation runs. † is placed, if the simulation did not finish in reasonable time (>40 days).	119
6.12	Comparison of run-times for the direct method (a) and δ -leaping (b). \mathcal{SPN}_{ECOLI} was parametrised with N and simulated with several number of simulation runs. † is placed, if the simulation did not finish in reasonable time (>40 days).	121
6.13	The size of the state space for different initial markings of \mathcal{GSPN}_{FMS} computed with MARCIE's symbolic state space generation. . .	122
6.14	Transient analysis up to time point $\tau = 1$ for different number of items N of \mathcal{GSPN}_{FMS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.	122
6.15	Steady state analysis for different number of items N of \mathcal{GSPN}_{FMS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 128 simulation runs.	124
6.16	The size of the state space for different number of stations N of \mathcal{SPN}^c_{CSPS} computed with MARCIE's symbolic state space generation.	127

6.17	Transient analysis up to time point $\tau = 10$ for different number of stations N of \mathcal{SPN}^c_{CSPS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.	127
6.18	Steady state analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 128 simulation runs.	128
6.19	Reward analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The expected reward value R is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.	129
6.20	Performability analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.	130

List of Algorithms

1	Generic stochastic simulation algorithm	32
2	Direct method	34
3	Optimised direct method using 2D - search	37
4	First reaction method	38
5	Next reaction method	40
6	Weighted random shuffle	46
7	δ -leaping algorithm	47
8	Check immediate transitions	50
9	Check deterministic and scheduled transitions	52
10	Evaluate time-bounded formula	79
11	Evaluate time-unbounded formula	81
12	Steady state computation for one simulation run	84
13	Evaluate time-bounded path formula	88
14	Evaluate state formula	89
15	Evaluate time-unbounded path formula	90
16	Steady state computation for one simulation run	92
17	Evaluate reward formula	96
18	Evaluate time-bounded reward path formula	99

Chapter 1

Introduction

The traditional approach to investigate the time evolution of biochemical reaction networks is by solving a set of coupled ordinary differential equations. One equation per species; each equation embodies the change of the specie's concentration over time with respect to the stoichiometry and kinetic rate constants of the chemical reactions it is involved in. This deterministic formulation is valid in most situations, but there are cases, e.g., non-linear systems, where it is not. In such cases stochastic formulation of the chemical kinetics gives correct results. Moreover, the stochastic approach is valid in the same situations as the deterministic approach, but it is sometimes even valid, when the deterministic is not, see [OSW69; Kur72]. Therefore, stochastic modelling has become an important tool to fully understand the system behaviour of such reaction networks. Moreover, biochemical reactions are inherently stochastic at the molecular level, thus the application of stochastic modelling is most natural.

Beyond that the field of performance evaluation is traditionally based on stochastic modelling. Complex technical systems usually comprise multiple processes that act concurrently. They may precede each other or synchronise in certain situations. Many of them compete with different priorities for limited resources, what can be modelled stochastically [Haa04]. In addition communication protocols, manufacturing systems or concurrent programs were modelled and analysed stochastically [CT93; Ajm+95; Ger01]. Hence stochastic modelling is getting more and more popular.

This increases the demand for efficient analysis of such models. While small

and medium-sized models can be analysed numerically, we focus on large or unbounded models. Our method of choice is stochastic simulation to overcome the problem of state space explosion.

We use stochastic Petri nets (\mathcal{SPN}) [Ajm+95; Ger01] as modelling paradigm, which gives us a completely formalised and standardised framework, as well as an intuitive way of modelling concurrent behaviour. The semantics of such a \mathcal{SPN} is defined as continuous-time Markov chain (CTMC).

The dynamic behaviour of stochastic models can be analysed in different ways. We showed in [Hei+10] that numerical analysis is efficient up to 1×10^9 states with current computer techniques. Beyond this limit, stochastic simulation remains the only possible technique. Stochastic simulation may be performed with approximate or exact methods. But approximate methods are of limited use for simulative model checking, because we need to know the exact occurrences of each transition, that means the simulation has to compute real (exact) paths through the state space of the net. Therefore, exact simulation algorithms are more suitable for the purpose of simulative model checking, like Gillespie’s direct method [Gil76] or the next reaction method by Gibson & Bruck [GB00].

Simulative model checking of time-bounded temporal formulas is well known and produces reasonable results and performs well in comparison to numerical methods [Hei+10]. The main issue with verifying time-unbounded formulas is: “When to stop the simulation trace?” Naive solutions like a fixed, large number of simulation steps or a fixed, long end time for the simulation trace, are not suitable. To compute the transient probability of the formula $\mathcal{P}_{=?}[\phi_1 \text{ U } \phi_2]$ in state s means to compute the probability distribution starting in s and making states absorbing, which satisfy $\neg\phi_1 \vee \phi_2$. The resulting linear system of equations can be solved numerically by iterative methods like Gauss-Seidel or Jacobi. There are several tools available that support such solvers, among them MARCIE [SRH11]. The drawback of numerical solvers is their restriction to bounded CTMCs. On the other hand they compute an “exact” result. The same methods were used to compute the steady state distribution of bounded CTMCs for computing the steady state probability of formulas like $\mathcal{S}_{=?}[\phi]$.

Statistical model checking [Bal+09; BGH09; YCZ11] is a quite similar approach to simulative model checking, but differs in some details. Hypothesis

testing, i.e., sequential probability ratio test (SPRT), has good performance compared to the computation of point estimates, but it can only check formulas like $\mathcal{P}_{\bowtie x}$. In the end, the user gets a result of *true* or *false* and has no idea of the scale of the estimated probability.

Rabih et al. [RP09] developed a different simulation-based approach to verify time-unbounded Until formulas. Their algorithm is based on perfect simulation. The approach works well if the CTMC is monotone. In the other case the algorithm is practically useless. The authors did not show, how to determine, whether a CTMC is monotone or not. Therefore it is not clear whether this approach is generally applicable or not.

The on-the-fly probabilistic model checker MIRACH, developed by Koh et al. [Koh+11], implements simulation-based PLTL model checking of quantitative pathway models, defined in SBML [Huc+03]. The model checking capabilities are limited to an upper time bound, due to the requirement of specifying a time limit for the trace generation.

The Monte Carlo Model Checker MC2 [DG08b] computes a point estimate of a Probabilistic LTL logic (with numerical constraints) formula to hold for a model. MC2 does not include any simulation engine, but works off-line by taking a set of sampled trajectories generated by any simulation engine or ODE solver, or even time lines measured in the wet lab.

Last not least, a combination of discrete event simulation and reachability analysis were used to compute time-unbounded formulas in [YCZ11; Zap08]. But this approach suffers from the same restrictions of bounded state spaces as the numerical methods.

Contribution

The research in this thesis will focus on stochastic simulation algorithms, simulative analysis and model checking of coloured extended stochastic Petri nets. We introduce an approach to improve the efficiency of stochastic simulation for large and dense networks by a new approximate stochastic simulation algorithm called *discrete-time leap method*. This algorithm enables us to simulate genome scale metabolic models in a reasonable amount of time. Moreover, we present simulative analysis of stochastic Petri nets and we show that simulative analysis is not restricted to trace generation. We are able to compute

approximations of transient solutions and steady state distributions. In case of transient solutions, we introduce some optimizations to make the computation more efficient. The computation of derived measures (observers) paves us a way to a whole new class of models, namely Markov reward models. We develop an infinite time horizon model checking algorithm plus steady state operator for probabilistic linear-time temporal logic. In addition, we show simulative model checking algorithms of continuous stochastic logic formulas including reward extensions and time-unbounded temporal operators. To the best of our knowledge, we develop the first simulative continuous stochastic reward logic model checking algorithm. Five biochemical case studies and two technical systems are going to demonstrate the capabilities of simulative analysis and simulative model checking.

The material of the thesis has been organised in a textbook style, to make it self-contained and easy to read without any pre-knowledge.

In the following we emphasize the novel contributions to scientific knowledge made throughout the thesis:

1. We introduce a new approximate stochastic simulation algorithm called *discrete-time leap method*, see Section 3.7.
2. We show how to compute approximations of transient solutions and steady state distributions by use of stochastic simulation. We introduce some optimizations to make the computation of transient solutions more efficient, see Section 4.2 and 4.3.
3. We show how to apply simulative analysis to Markov reward models, see Section 4.4.
4. We exploit the steady state property to develop an infinite time horizon model checking algorithm including steady state operator for probabilistic linear-time temporal logic, see Section 5.1.
5. We make use of the steady state property to develop an infinite time horizon model checking algorithm including steady state operator for continuous stochastic logic, see Section 5.2.
6. We develop the first simulative model checking algorithm for continuous stochastic reward logic. It incorporates state and impulse reward

functions, see Section 5.4.

Publications

We list the publications of the thesis' author, which are grouped by their impact to the content of the thesis. We start with the highest impact publications that were written solely by Christian Rohr.

[Roh10]: This publication sets the starting point in developing simulative model checking algorithms presented in this thesis. It contains an algorithm for model checking continuous stochastic logic formulas with the following constraints: no nested formulas, no steady state operator and only time bounded temporal operators.

[Roh12]: This paper comprises an infinite time horizon model checking algorithm plus steady state operator for probabilistic linear-time temporal logic.

[Roh13]: This follow-up publication extends the previous one by additional explanations and experimental results.

[Roh16]: The new approximate stochastic simulation algorithm *discrete time leap method* is introduced in this paper.

The next group of publications [RMH10; Hei+10; SRH11; MRH12; Hei+12; HLR14] were written in collaboration and present methods and tools for stochastic modelling and analysis. Christian Rohr was responsible for any aspects involving stochastic simulation.

The last group of publications [Blä+13; Blä+14; BR15; Hei+16] is not related to the contents of the thesis and presents additional research interests.

Outline

The thesis is organised as follows:

Chapter 2: In this chapter we give the necessary definitions of the used Petri net classes. We focus on the stochastic Petri net classes and on their coloured counterpart. This includes stochastic Petri nets, generalized stochastic Petri nets and extended stochastic Petri nets.

Chapter 3: We give an overview on the existing stochastic simulation algorithms. There are two kinds of algorithms, the exact simulation algorithms that compute an exact path through the continuous-time Markov chain and the approximate algorithms that leap over states and thus have better computational performance, but at the expense of exactness. Furthermore, we present a new approximate stochastic simulation algorithm, called *discrete-time leap method*. We show that the approximation is very close to the exact simulation algorithm. This algorithm allows us to simulate genome scale metabolic models in reasonable time.

Chapter 4: When talking about simulative analysis, trace generation is the most common technique. But there is much more possible, as we show in this chapter. We demonstrate how to compute transient solutions and steady state distribution. Moreover, we enrich the simulative analysis by the addition of observers.

Chapter 5: Simulative model checking is an advanced analysis technique, which we present in this chapter. We use the probabilistic linear-time temporal logic with numerical constraints and extend it by means of the steady state operator. Besides that, we demonstrate how to apply simulative model checking to continuous stochastic (reward) logic formulas. Stochastic simulation based techniques can compute just finite paths, so time unbounded properties can not be verified easily. We apply the steady state property on our model checking procedure to overcome this issue.

Chapter 6: We exemplify the presented simulative analysis techniques on several case studies in this chapter. Furthermore, we compare the runtime performance of the introduced discrete-time leap method with the well known and widely used direct method on models of different size. We use models ranging from just a few nodes and arcs up to some thousand nodes and tens of thousand arcs. The models originate from different areas, e.g., systems biology, technical systems.

Chapter 7 This chapter summarises the achieved results and provides some ideas for future research.

Chapter 2

Preliminaries

In this chapter we describe the Petri net formalism used throughout this thesis. We give the definitions of all net classes and needed supplementaries.

The beginnings of the Petri net theory go back to the dissertation of Carl Adam Petri [Pet62]. He described the basic ideas from which place/transition nets, also known nowadays as Petri nets, emerged. They are well suited to model asynchronous, concurrent, non-deterministic or parallel systems.

Since its creation, there have been many extensions of Petri nets. The list of extensions ranges from additional arc types (e.g., read, inhibitor and reset arcs), via Petri nets with priorities, timed Petri nets (e.g., deterministically and stochastic) to coloured Petri nets. They have been used in many areas to model and analyse systems, e.g., academia, technical systems, protocol engineering, software design, systems biology, and work flow management.

Petri nets have an intuitive graphical representation even favouring its use by non-experts. They provide an unambiguous framework for interdisciplinary collaboration with profound mathematical foundation.

2.1 Petri Net

A **Petri net** is a weighted, directed, bipartite graph. It consists of two types of nodes, called places and transitions. Transitions can be seen as events that may occur, a rectangle is their typical graphical representation. Places can be seen as conditions for the events, they are usually shown as circles. A place can carry an arbitrary number of tokens, they are represented by dots or as a

digit on this place. The number of tokens on all places of a network is called marking, and it defines the current state of the network. Places and transitions are connected by directed arcs which show the way of token flow. Arcs go only from places to transitions or vice versa, never between nodes of the same type. The arc inscription denotes the amount of tokens that flows over it.

A transition becomes enabled if the amount of tokens on its pre-places is greater or equal than the respective arc inscription. If a transition is enabled then it may fire and it consumes the number of tokens from its pre-places with respect to the arc inscription and produces as many tokens on its post-places as denoted by the particular arc inscription. After firing a transition the network reaches a new marking (state), this reflects the dynamics of the network. The initial state is defined by the initial marking. The dynamic behaviour of the network may be represented by the set of markings reachable from the initial marking.

Definition 1 (Net). A *net* is a 3-tuple $\mathcal{N} = (P, T, A)$ where:

1. $P = \{p_1, p_2, \dots, p_m\}$ is a finite set of places.
2. $T = \{t_1, t_2, \dots, t_n\}$ is a finite set of transitions.
3. P and T satisfy $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$.
4. $A: ((P \times T) \cup (T \times P)) \rightarrow \mathbb{N}_0$ is a multi-set of arcs. It assigns a natural number (*arc weight, multiplicity*) to every arc of the net. «

From now on, we consider only those elements of A with $A(x, y) > 0$ as *arcs*. The *flow relation* is the set of arcs $F = \{(x, y) \mid A(x, y) > 0\}$. The arcs of the net describe the pre- and post-sets of a net.

Definition 2 (Pre- and Post-set). Let $\mathcal{N} = (P, T, A)$ be a net. For a node $x \in P \cup T$ two sets of nodes are defined:

1. $\bullet x = \{y \in P \cup T \mid A(y, x) > 0\}$ is the *pre-set* of x ,
2. $x \bullet = \{y \in P \cup T \mid A(x, y) > 0\}$ is the *post-set* of x . «

In summary we get four types of node sets:

- $\bullet t$, the pre-places, input places of transition t ,

- t^\bullet , the post-places, output places of transition t ,
- $^\bullet p$, the pre-transitions of place p , producing tokens on p ,
- p^\bullet , the post-transitions of place p , consuming tokens from p .

For a set of nodes $X \subseteq P \cup T$ we define the set of all pre-nodes $^\bullet X = \bigcup_{x \in X} ^\bullet x$ and the set of all post-nodes $X^\bullet = \bigcup_{x \in X} x^\bullet$.

Places of a net are marked with zero or more *tokens*. The distribution of tokens over the places of the net is called *marking* or *state* of the net.

Definition 3 (Marking). Let $\mathcal{N} = (P, T, A)$ be a net. Any marking m is a mapping

$$m: P \rightarrow \mathbb{N}_0 .$$

It maps the set of places onto the set of natural numbers, where $m(p)$ defines the number of tokens in place $p \in P$. The set of all possible markings is denoted by $M = \mathbb{N}_0^{|P|}$. «

If appropriate and the context precludes confusion, we treat a place like a (integer) variable and write simply p instead of $m(p)$.

A place $p \in P$ with $m(p) = 0$ is called *clean* or *unmarked* in m , otherwise it is called *marked* in m . A net is called *clean* if all of its places are *clean*, otherwise *marked*.

Any marking of the net can be represented as a vector

$$\underline{m} = (m(p_1), m(p_2), \dots, m(p_m)) \in \mathbb{N}_0^{|P|} .$$

For simplicity we do not differentiate between these two representations any more. For vectors we define comparison and addition place-wise:

$$\begin{aligned} m_1 \leq m_2 &\iff \forall p \in P: m_1(p) \leq m_2(p) \\ m_1 < m_2 &\iff m_1 \leq m_2 \text{ and } \exists p \in P: m_1(p) < m_2(p) \\ m = m_1 + m_2 &\iff \forall p \in P: m(p) = m_1(p) + m_2(p) . \end{aligned}$$

Definition 4 (Submarking). Let $\mathcal{N} = (P, T, A)$ be a net, m a marking of \mathcal{N} , and Q a subset of P . We define the submarking m^Q with $Q \subseteq P$ as

$$m^Q: Q \rightarrow \mathbb{N}_0 .$$

The set of all submarkings is denoted by $M^Q = \mathbb{N}_0^{|Q|}$. «

Definition 5 (Petri net). A Petri net (\mathcal{PN}) is a tuple $\mathcal{PN} = (P, T, A, m_0)$ where:

1. $\mathcal{N} = (P, T, A)$ is a net.
2. m_0 is an initial marking. «

Figure 2.1 shows the typical graphical representation of Petri net elements. Places are represented as circles, transitions as rectangles, and they are connected via directed arcs. An arc weight of 1 is usually omitted; that means an arc showing up in a net without an arc weight has a multiplicity of 1. The number of tokens on a place are represented by the same amount of \bullet inside the circle, but if $m(p) > 4$ the actual number is given.

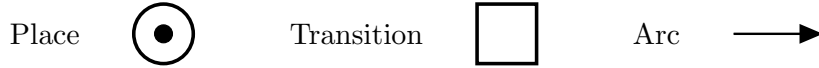


Figure 2.1: Elements of a Petri net.

Example 1. The classical „Producer & Consumer” with a buffer of size $B \in \mathbb{N}^+$ modelled as a Petri net is shown in Figure 2.2. The Petri net is defined

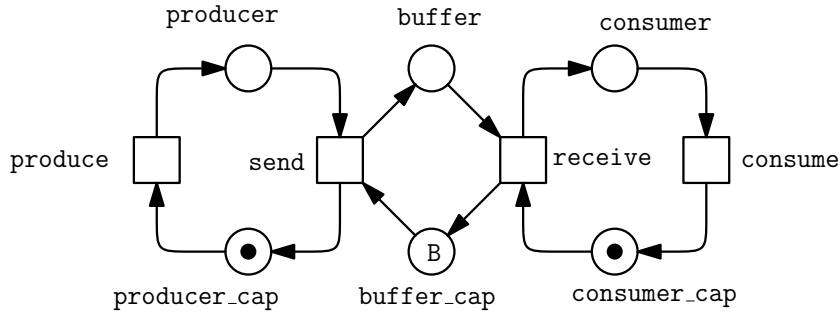


Figure 2.2: Producer & Consumer modelled as Petri net.

with:

- $P = \{\text{producer}, \text{producer_cap}, \text{buffer}, \text{buffer_cap}, \text{consumer}, \text{consumer_cap}\}$
- $T = \{\text{produce}, \text{send}, \text{receive}, \text{consume}\}$

- $A(\text{producer}, \text{send}) = 1, A(\text{producer_cap}, \text{produce}) = 1,$
 $A(\text{buffer_cap}, \text{send}) = 1, A(\text{buffer}, \text{receive}) = 1,$
 $A(\text{consumer_cap}, \text{receive}) = 1, A(\text{consumer}, \text{consume}) = 1,$
 $A(\text{produce}, \text{producer}) = 1, A(\text{send}, \text{producer_cap}) = 1,$
 $A(\text{send}, \text{buffer}) = 1, A(\text{receive}, \text{consumer}) = 1,$
 $A(\text{receive}, \text{buffer_cap}) = 1, A(\text{consume}, \text{consumer_cap}) = 1,$
 in all other cases $A(x, y) = 0$
- $m_0 = (0, 1, 0, B, 0, 1), B \in \mathbb{N}^+$

After defining the structure of Petri nets, we discuss their dynamic behaviour now.

Definition 6 (Enabling and firing vectors). Let $\mathcal{PN} = (P, T, A, m_0)$ be a Petri net. For every transition $t \in T$ we define the following mappings for every place $p \in P$:

$$\begin{aligned} t^-(p) &= A(p, t) \\ t^+(p) &= A(t, p) \\ \Delta t(p) &= t^+(p) - t^-(p). \end{aligned}$$

For a Petri net \mathcal{PN} with places $P = \{p_1, p_2, \dots, p_m\}$, the mappings can be considered as vectors

$$t^- \in \mathbb{N}_0^{|P|}, t^+ \in \mathbb{N}_0^{|P|} \text{ and } \Delta t \in \mathbb{N}_0^{|P|}. \quad \ll$$

Definition 7 (Enabling condition). Let $\mathcal{PN} = (P, T, A, m_0)$ be a Petri net and m a marking of \mathcal{PN} . A transition $t \in T$ is called enabled in marking m , if $m \geq t^-$. «

Definition 8 (Firing rule). Let $\mathcal{PN} = (P, T, A, m_0)$ be a Petri net and m a marking of \mathcal{PN} . If a transition $t \in T$ is enabled in m , it may fire. When t in m fires, a new marking $m' = m + \Delta t$ is reached. This is denoted by $m \xrightarrow{t} m'$. The firing itself does not consume any time and takes place atomically. «

The firing of a transition t occurs in two parts. First, the transition removes the amount of tokens $t^-(p)$ from each of its pre-places. Second, it adds the

amount of tokens $t^+(p)$ to each of its post-places. Please note that a transition is never compelled to fire.

Definition 9 (Reachability relation). Let $\mathcal{PN} = (P, T, A, m_0)$ be a Petri net and m a marking of \mathcal{PN} . We denote a firing sequence of transitions as $\delta = \langle t_{i_1}, t_{i_2}, \dots, t_{i_n} \rangle$ such that $M \xrightarrow{t_{i_1}} m_1 \wedge m_1 \xrightarrow{t_{i_2}} m_2 \wedge \dots \wedge m_{n-1} \xrightarrow{t_{i_n}} m_n$. The set of all firing sequences, including the empty sequence ϵ , is denoted as δ^* . The *reachability relation* $\xrightarrow{*}$ for two markings m and m' of \mathcal{PN} is now defined as $m \xrightarrow{*} m'$ iff $\exists \delta \in \delta^*: m \xrightarrow{\delta} m'$. A marking m' is called *reachable* from m iff $m \xrightarrow{\delta} m'$. «

Definition 10 (Reachability set). Let $\mathcal{PN} = (P, T, A, m_0)$ be a Petri net. We define the *reachability set* of \mathcal{PN} as $\mathcal{R}_{\mathcal{PN}}(m_0) = \{m' \mid m_0 \xrightarrow{*} m'\}$. It shall be called *state space*, too. «

Example 2. The set of reachable markings of the Petri net from Example 1 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ is

$$\begin{aligned} \mathcal{R}_{\mathcal{PN}}(m_0) = \{ & (0, 1, 0, 1, 0, 1), (0, 1, 1, 0, 0, 1), (0, 1, 0, 1, 1, 0), (0, 1, 1, 0, 1, 0), \\ & (1, 0, 0, 1, 0, 1), (1, 0, 1, 0, 0, 1), (1, 0, 0, 1, 1, 0), (1, 0, 1, 0, 1, 0) \} . \end{aligned}$$

2.2 Reachability Graph

A **reachability graph** represents the dynamic behaviour of a Petri net with respect to the *interleaving semantics*. It consists of one type of nodes which are connected with directed arcs. Each node contains a marking $m \in \mathcal{R}_{\mathcal{PN}}(m_0)$ and an arc between two nodes m and m' is labelled with transition $t \in T$ if $m \xrightarrow{t} m'$. We call two transitions $t_1, t_2 \in T$ *parallel*, iff $m \xrightarrow{t_1} m' \wedge m \xrightarrow{t_2} m'$. Without loss of generality, we do not allow *parallel* transitions, because they don't bring new behaviour w.r.t. the interleaving semantics.

Definition 11 (Reachability graph). Let $\mathcal{PN} = (P, T, A, m_0)$ be a Petri net. A *reachability graph* (\mathcal{RG}) of a Petri net \mathcal{PN} is a tuple $\mathcal{RG}_{\mathcal{PN}} = (\mathcal{R}_{\mathcal{PN}}(m_0), \mathbf{R}, m_0)$. with $\mathcal{R}_{\mathcal{PN}}(m_0)$ denoting the state space of the underlying net and m_0 the initial state.

$$\mathbf{R} : \mathcal{R}_{\mathcal{PN}}(m_0) \times \mathcal{R}_{\mathcal{PN}}(m_0) \rightarrow T \cup \{0\}$$

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

«

Example 3. The Reachability graph of the Petri net from Example 1 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ is shown in Figure 2.3.

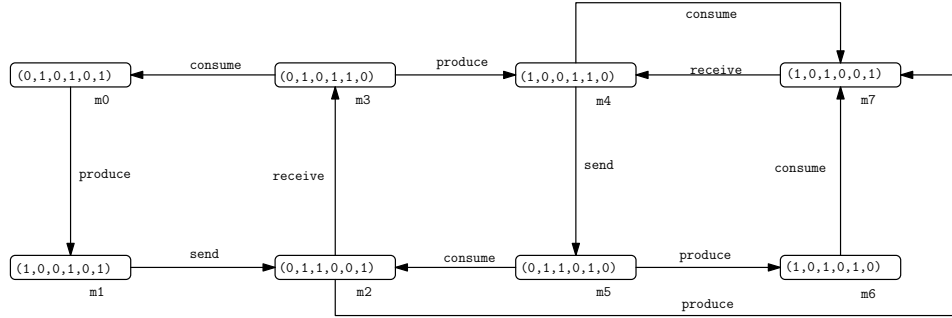


Figure 2.3: Reachability graph of the Petri net from Example 1 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$.

2.3 Extended Petri Net

An **extended Petri net** (\mathcal{XPN}) builds on \mathcal{PN} enriched by four special arc types: *read*, *inhibitor*, *equal*, and *reset*. They always go from a place to a transition. The first three arcs establish additional side conditions for the enabledness of a transition, but upon firing, the marking on the tested place is not changed. Contrary, the reset arc does not influence the enabledness, but upon firing all tokens on the tested place are removed.

Definition 12 (Extended Petri net). An extended Petri net (\mathcal{XPN}) is a tuple $\mathcal{XPN} = (P, T, A, m_0)$ where:

1. $P = \{p_1, p_2, \dots, p_m\}$ is a finite set of places.
2. $T = \{t_1, t_2, \dots, t_n\}$ is a finite set of transitions.
3. P and T satisfy $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$.
4. $A = A_s \cup A_r \cup A_i \cup A_e \cup A_z$ is a multi-set of arcs with:

- $A_s: ((P \times T) \cup (T \times P)) \rightarrow \mathbb{N}_0$ a set of *standard* arcs,
- $A_r: (P \times T) \rightarrow \mathbb{N}_0$ a set of *read* arcs,
- $A_i: (P \times T) \rightarrow \mathbb{N}_0$ a set of *inhibitor* arcs,
- $A_e: (P \times T) \rightarrow \mathbb{N}_0$ a set of *equal* arcs,
- $A_z: (P \times T) \rightarrow \{0, 1\}$ a set of *reset* arcs.

An \mathcal{XPN} can be equally well called a Petri net with extended arcs. «

The *read* arc (sometimes called *test* arc) enables a transition, if the amount of tokens on its pre-place is greater or equal than the arc weight. The *inhibitor* arc was introduced by Flynn and Agerwala in [FA73]. It switches around the meaning of the arc weight, in the manner that the transition is enabled if there are less tokens on the pre-place than the inhibitor arc weight. The *equal* arc enables a transition if its pre-place has exactly the same amount of tokens as the arc weight. In fact the equal arc is syntactic sugar, because this could be expressed by a combination of a read arc and an inhibitor arc, too. The *reset* arc has no impact on the enabledness of a transition, but when the transition fires all tokens are removed from its pre-place. It holds for the extended arcs as well that only arc weights of $A(x, y) > 0$ are considered in terms of the flow relation.

Special arcs enhance the modelling comfort, but the inhibitor arc brings the Turing power, which destroys the general decidability of non-trivial behavioural properties. In models with finite state space, special arcs can be simulated by standard arcs and some kind of extracting of the tested places. However, this might lessen the analysis efficiency as discussed in [SH09]. Special arcs do not cause any trouble for dynamic, i.e. state-space-related analysis techniques for finite state spaces as long as we consider interleaving semantics.

The typical graphical representation of the additional arc types of an extended Petri net is shown in Figure 2.4. Read arcs end with a black filled circle, inhibitor arcs with a hollowed circle, equal arcs with two black filled circles, and reset arcs end with two black arrows.

The enabling vectors (Def. 6), the enabling condition (Def. 7) and the firing rule (Def. 8) need to be adapted for the new arc types.

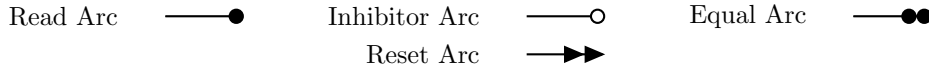


Figure 2.4: Additional arc types of an extended Petri net.

Definition 13 (Extended enabling vectors). Let $\mathcal{XPN} = (P, T, A, m_0)$ be an extended Petri net. We adapt the mappings of Def. 6 for every transition $t \in T$:

$$\begin{aligned}
 t^-(p) &= A_s(p, t) \\
 t^+(p) &= A_s(t, p) \\
 t_r^-(p) &= A_r(p, t) \\
 t_i^-(p) &= \begin{cases} A_i(p, t) & \text{if } A_i(p, t) > 0 \\ \infty & \text{otherwise} \end{cases} \\
 \Delta t(p) &= t^+(p) - t^-(p).
 \end{aligned}$$

«

There is no need to define a mapping for the equal arcs, because they will be treated as a combination of read and inhibitor arcs. Indeed

$$A_e(p, t) = n \equiv A_r(p, t) = n \wedge A_i(p, t) = n + 1 .$$

Definition 14 (Extended enabling condition). Let $\mathcal{XPN} = (P, T, A, m_0)$ be an extended Petri net and m a marking of \mathcal{XPN} . A transition $t \in T$ is called *enabled* in marking m , iff $m \geq t^- \wedge m \geq t_r^- \wedge m < t_i^-$. «

Definition 15 (Extended firing rule). Let $\mathcal{XPN} = (P, T, A, m_0)$ be an extended Petri net and m a marking of \mathcal{XPN} . If a transition $t \in T$ is enabled in m , it may fire. When t in m fires, a new marking m' is reached. This takes place in the following way:

$$m'(p) = \begin{cases} t^+(p) & \text{if } A_z(p, t) > 0 \\ m(p) + \Delta t(p) & \text{otherwise.} \end{cases}$$

«

Example 4. The Petri net from Example 1 can be modelled as an extended Petri net using inhibitor arcs. Figure 2.5 shows the adapted net. The places

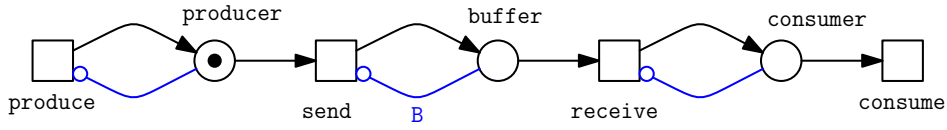


Figure 2.5: Producer & Consumer modelled as extended Petri net using inhibitor arcs.

defining the capacities were removed and inhibitor arcs are used to model the capacities. The reachability graph of this net with $B = 2$ is shown in Figure 2.6.

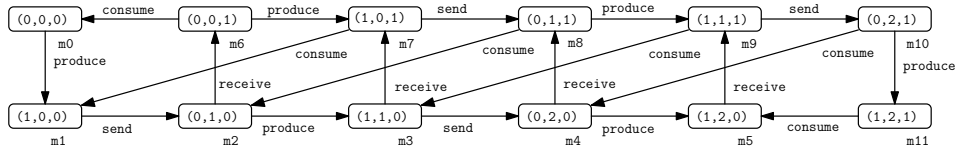


Figure 2.6: Reachability graph of Producer & Consumer \mathcal{KPN} with $B = 2$.

2.4 Marking-dependent Extended Petri Net

The need for marking-dependent arc weights arises, if one wants to model, e.g., the diffusion of tokens or the transfer of, e.g., all tokens from one place to an other place in one step. It is possible to model this for a known finite number of tokens on that place by taking each possible marking into account. But this blows up the net structure and is an error-prone and inefficient approach. If the maximum number of tokens on that place is not known, e.g., it is infinite, this is not possible at all.

Valk introduced an extension to Petri nets, called self-modifying nets [Val78]. Using this extension it's now possible to model the above mentioned cases. Example 5 shows usage of marking-dependent arc weights for the transfer of all tokens on the place *buffer* in one step to the place *consumer*. Later on, Ciardo presented a hierarchy of nets with marking-dependent arc weights and showed that such nets are Turing-equivalent [Cia94]. The definitions of Ciardo and Valk allow the use of any place of the net for marking-dependent arc

weights, but we restrict this to the pre-places of each transition. We added this to make the dependencies clearly visible in the net. In the end, it is no restriction at all, because one can overcome it by adding a read arc from the needed place with an arc weight set to this place.¹

Definition 16 (Marking-dependent arc weight). Given the set of all markings M , we define a marking-dependent (m-dependent for short) arc weight as

$$g: M \rightarrow \mathbb{N}_0 .$$

A m-dependent arc weight will be restricted to the pre-places of the connected transition $t \in T$. Therefore g_t will be defined on the submarkings $M^{\bullet t}$ as

$$g_t: M^{\bullet t} \rightarrow \mathbb{N}_0 .$$

The set of all marking-dependent arc weights is denoted by

$$G = \bigcup_{t \in T} g_t . \quad \ll$$

We permit as definition of marking-dependent arc weights arithmetic functions over the pre-places of a transition.

Definition 17 (Marking-dependent extended Petri net). An extended Petri net with marking-dependent arc weights (\mathcal{MXPN}) is a tuple $\mathcal{MXPN} = (P, T, A, G, m_0)$ where:

1. $P = \{p_1, p_2, \dots, p_m\}$ is a finite set of places.
2. $T = \{t_1, t_2, \dots, t_n\}$ is a finite set of transitions.
3. P and T satisfy $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$.
4. m_0 is an initial marking.
5. G is the set of marking-dependent arc weights.
6. $A = A_s \cup A_r \cup A_i \cup A_e \cup A_z$ is a multi-set of arcs with:

¹In Section 2.5 we introduce *modifier* arcs for that purpose.

- $A_s: ((P \times T) \cup (T \times P)) \rightarrow G$ a set of *m-dependent standard arcs* and
 $A_s(p, t) = A_s(t, p) = g_t$,
- $A_r: (P \times T) \rightarrow G$ a set of *m-dependent read arcs* and
 $A_r(p, t) = g_t$,
- $A_i: (P \times T) \rightarrow G$ a set of *m-dependent inhibitor arcs* and
 $A_i(p, t) = g_t$,
- $A_e: (P \times T) \rightarrow G$ a set of *m-dependent equal arcs* and
 $A_e(p, t) = g_t$,
- $A_z: (P \times T) \rightarrow \{0, 1\}$ a set of *reset arcs*. «

The definition of \mathcal{MXPN} (Def. 17) incorporates \mathcal{XPN} as a special case. To be specific, if all arc weights are constant functions, we get an extended Petri net. As soon as one arc weight is depending on a place, we get a marking-dependent \mathcal{XPN} .

Definition 18 (M-dependent enabling and firing vectors). Let $\mathcal{MXPN} = (P, T, A, G, m_0)$ be a marking-dependent extended Petri net. We extend the mappings of Def. 13 for every transition $t \in T$:

$$\begin{aligned}
 t_m^-(p) &= A_s(p, t)(m^{\bullet t}) \\
 t_m^+(p) &= A_s(t, p)(m^{\bullet t}) \\
 t_{m_r}^-(p) &= A_r(t, p)(m^{\bullet t}) \\
 t_{m_i}^-(p) &= \begin{cases} A_i(p, t)(m) & \text{if } A_i(p, t)(m^{\bullet t}) > 0 \\ \infty & \text{otherwise} \end{cases} \\
 \Delta t_m(p) &= t_m^+(p) - t_m^-(p). \quad \ll
 \end{aligned}$$

Example 5. We modify Example 4 in the way that the *Consumer* receives the whole buffer contents in one step, no matter how much tokens the buffer contains. Therefore the arcs from *buffer* to *receive* and from *receive* to *consumer* need a marking-dependent arc weight. In our case, it's just the label *buffer*. The reachability graph of this net with $B = 2$ is shown in Figure 2.8. If we compare the reachability graph of this net ($\mathcal{RG}_{\mathcal{MXPN}}$) with the one from the \mathcal{XPN} version ($\mathcal{RG}_{\mathcal{XPN}}$ in Figure 2.6), we see that $\mathcal{RG}_{\mathcal{MXPN}}$ has fewer

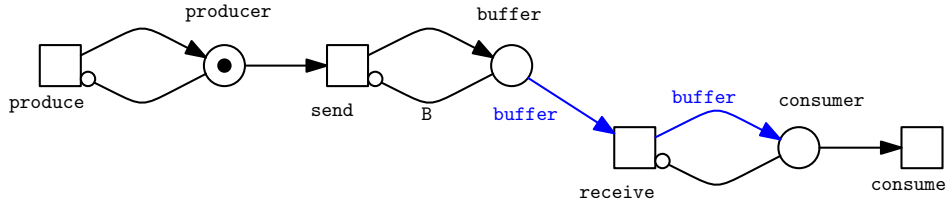


Figure 2.7: Producer & Consumer modelled as m-dependent \mathcal{XPN} using marking-dependent arc weights.

states, because the whole *buffer* contents will be *received* in one step, rather than one token after the other.

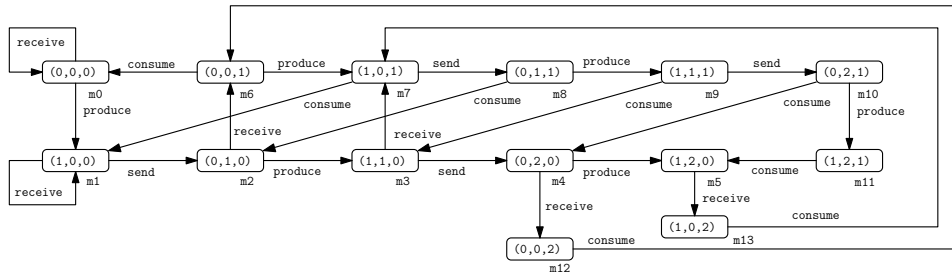


Figure 2.8: Reachability graph of Producer & Consumer \mathcal{MXP} with $B = 2$.

2.5 Stochastic Petri Net

A **stochastic Petri net** (\mathcal{SPN}) builds on \mathcal{PN} , but transitions have an exponentially distributed firing delay, characterised by the firing rate λ . A transition may lose its enabledness while waiting for the delay to expire. The firing itself does not consume time and follows the standard Petri net firing rule. The firing rates are typically transition-specific and marking-dependent and defined by stochastic firing rate functions, also known as propensity or hazard functions. The mapping $V : T \rightarrow H$, where H is the set of *hazard* functions, associates to each transition a function h_t from H . We deal with biologically interpreted stochastic Petri nets; thus we consider besides arbitrary arithmetic functions specifically propensity functions representing *mass action semantics* and *level interpretation semantics*. All these functions have in common that the domain is restricted to the pre-places of the corresponding transition; see [GHL07]. Additionally, *modifier arcs* are introduced to keep the locality principle of Petri nets. The modifier arc does neither restrict the

enabledness nor does it change the tokens upon firing. But the firing rate may depend on the current marking of the connected place. Keeping the locality principle is crucial for the efficiency of our analysis tools.

Definition 19 (Marking dependent function). Given the set of all markings M , we define the marking-dependent function as

$$h: M \rightarrow \mathbb{R}_0^+ .$$

A marking-dependent function will be restricted to the pre-places of the associated transition $t \in T$. Therefore h_t will be defined on the submarking $M^{\bullet t}$ as

$$h_t: M^{\bullet t} \rightarrow \mathbb{R}_0^+ .$$

The set of all marking-dependent functions is denoted by

$$H = \bigcup_{t \in T} h_t . \quad \ll$$

Mass-action kinetics can be achieved by using the following rate function [HGD08]:

$$h_t = c_t \cdot \prod_{p \in \bullet t} \binom{m(p)}{f(p, t)}, \quad (2.1)$$

where c_t is the real-valued, transition-specific rate constant. The binomial coefficient describes the number of unordered combinations of the required $f(p, t)$ molecules, out of the $m(p)$ available ones.

Definition 20 (Stochastic Petri net). A stochastic Petri net (\mathcal{SPN}) is a tuple $\mathcal{SPN} = (P, T, A, V, H, m_0)$ where:

1. (P, T, A, m_0) is a Petri net.
2. $A = A_s \cup A_m$ is a multi-set of arcs with:
 - $A_s: ((P \times T) \cup (T \times P)) \rightarrow \mathbb{N}_0$ a set of *standard* arcs,
 - $A_m: (P \times T) \rightarrow \{0, 1\}$ a set of *modifier* arcs.
3. H is the set of marking-dependent functions.
4. $V: T \rightarrow H$ assigns to each transition a stochastic firing rate functions $V(t) = h_t$.

Table 2.1: The rate functions and parameters of the „Producer & Consumer” \mathcal{SPN} .

Transition	Rate function	Parameter	Value
produce	p_rate	p_rate	0.1
send	$s_rate * buffer_cap$	s_rate	0.2
receive	$r_rate * buffer$	r_rate	0.2
consume	c_rate	c_rate	0.3

The semantics of a \mathcal{SPN} is a continuous time Markov chain (CTMC), introduced in the next section. «

Example 6. We extend Example 1 to an \mathcal{SPN} by adding stochastic firing rate functions to each transition. The rate functions are shown in Table 2.1. The transitions *produce* and *consume* got a constant rate function, i.e., the firing rate of these transitions is constant over time and does not depend on the current marking. In contrast, the transitions *send* and *receive* got marking dependent rate functions, i.e., the firing rate of these transitions changes with respect to the current marking. To be precise, the firing rate of *send* increases while the number of tokens on *buffer* decreases and the firing rate of *receive* increases while the number of tokens on *buffer* increases.

2.6 Continuous-Time Markov Chain

A **continuous-time Markov chain** (CTMC) represents the dynamic behaviour of a stochastic Petri net. It is a stochastic process with an exponential probability distribution that has the *Markov property*. That means, its future behaviour depends only on its current state and not on former behaviour. The definition of a CTMC is comparable with the reachability graph, except that the arcs are labelled with the stochastic firing rate of the transition. We forbid *parallel* transitions here as well as for the reachability graph.

Definition 21 (Continuous-time Markov chain). A continuous-time Markov chain (CTMC) of a stochastic Petri net is a tuple $\text{CTMC}_{\mathcal{SPN}} = (\mathcal{R}_{\mathcal{SPN}}(m_0), \mathbf{Q}, m_0)$ with $\mathcal{R}_{\mathcal{SPN}}(m_0)$ denoting the state space of the underlying net and m_0 the ini-

tial state.

$$\begin{aligned}
\mathbf{Q} &: \mathcal{R}_{SPN}(m_0) \times \mathcal{R}_{SPN}(m_0) \rightarrow \mathbb{R}_0^+ \\
\mathbf{Q}(m, m') &= \begin{cases} h_t(m) & \exists t \in T : m \xrightarrow{t} m' \\ 0 & \text{otherwise} . \end{cases} \\
\mathbf{Q}(m, m) &= -E(m) \\
E(m) &= \sum_{m' \in \mathcal{R}_{SPN}(m_0)} \mathbf{Q}(m, m'), \text{ exit rate of state } m .
\end{aligned}$$

Let

$$1 - e^{-\mathbf{Q}(m, m') \cdot n} \quad (2.2)$$

be the probability of a transition t enabled in state m to fire (which results in state m') within n time units. «

Example 7. The continuous-time Markov chain regarding to the \mathcal{SPN} in Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ contains the same set of states as the reachability graph in Example 3. The difference lays in the labelling of the arcs, i.e., they are labelled with the stochastic firing rate of the transitions. The transition rate matrix of the CTMC is given below.

$$\mathbf{Q} = \begin{matrix} & \begin{matrix} m_0 & m_1 & m_2 & m_3 & m_4 & m_5 & m_6 & m_7 \end{matrix} \\ \begin{matrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \end{matrix} & \begin{pmatrix} -E(m_0) & p_rate & & & & & & \\ & -E(m_1) & s_rate & & & & & \\ & & -E(m_2) & r_rate & & & & p_rate \\ c_rate & & & -E(m_3) & p_rate & & & \\ & & & & -E(m_4) & s_rate & & c_rate \\ & & c_rate & & & -E(m_5) & p_rate & \\ & & & & & & -E(m_6) & c_rate \\ & & & & r_rate & & & -E(m_7) \end{pmatrix} \end{matrix}$$

2.7 Generalised Stochastic Petri Net

A **generalised stochastic Petri net** (\mathcal{GSPN}) builds on \mathcal{SPN} enriched by extended arc types (Def. 12) and *immediate transitions*. Immediate transitions have a zero firing delay. Thus, they fire immediately after getting enabled and always prior to stochastic transitions. Consequently, getting enabled and the firing itself coincide, if not prevented by another competing immediate

transition. Immediate transitions have a weight function associated with them, defining the relative amount of probability mass, transferred by itself. Conflicts between them are solved according the transitions weights. As for stochastic transitions, the firing follows the standard Petri net firing rule. A cyclic system behaviour involving only the firing of immediate transitions corresponds to an infinite behaviour without time progress – we get a new type of modelling fault, the time deadlock. Immediate transitions may help to avoid stiff systems by using them for transitions with extremely high rates (non-significant delay) compared to the other transitions in the system.

Definition 22 (Generalised stochastic Petri net). A generalised stochastic Petri net (\mathcal{GSPN}) is a tuple $\mathcal{GSPN} = (P, T, A, V, m_0)$ where:

1. P is a finite set of places
2. $T = T_s \cup T_i$ is a finite set of transitions with:
 - T_s a set of stochastic transitions
 - T_i a set of immediate transitions
3. P and T satisfy $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$
4. m_0 is an initial marking
5. $A = A_s \cup A_r \cup A_i \cup A_e \cup A_z \cup A_m$ is a multi-set of arcs
6. H is the set of marking-dependent functions
7. $V = V_s \cup V_i$ is a set of functions with:
 - $V_s: T_s \rightarrow H$ assigns to each stochastic transition a stochastic rate function $V_s(t) = h_{t_s}$,
 - $V_i: T_i \rightarrow H$ assigns to each immediate transition a weight function $V_i(t) = h_{t_i}$.

The semantics of a \mathcal{GSPN} is semi-Markovian. «

The addition of immediate transitions in \mathcal{GSPN} leads to two different kinds of states in the underlying stochastic process. We distinguish between transient (vanishing) and non-transient (tangible) states. A system never spends time in

a transient state before changing into another state. Thus, the time spent (sojourn time) in transient states is always zero, and not exponentially distributed any more. Consequently, the underlying semantics is not a continuous-time Markov chain any more. It is called semi-Markovian, because the transient states may be removed, under certain conditions, such that the reduced reachability graph corresponds again to a continuous-time Markov chain [Ajm+95].

2.8 Extended Stochastic Petri Net

An **extended stochastic Petri net** (\mathcal{XSPN}) builds on \mathcal{GSPN} enriched by marking-dependent arc weights (Def. 16) and deterministically timed transitions, which come in two flavours.

Deterministic transitions fire after a deterministic firing delay. The delay is always relative to the time point where a transition gets enabled. The transition may lose its enabledness while waiting for the delay to expire. Deterministic transitions may be useful to reduce networks, e.g. by replacing a linear sequence of stochastic transitions by one deterministic transition with the delay set to the sum of the expectation values of the transition sequence.

Scheduled transitions fire according to a schedule specifying absolute time points of the simulation time. A schedule can specify just a single time point, or equidistant time points within a given interval, triggering the potential firing (if the transition is enabled) once or periodically. They support the straightforward modelling of wet-lab experiment scenarios. The core model can be disturbed at well-defined time points as it is done experimentally with the actual biological system under investigation in the wet-lab. Scheduled transitions can be simulated by net components deploying immediate and deterministic transitions, see [Hei+09].

There is a standard firing rule, applying equally to all \mathcal{XSPN} transition types; specifically, a transition may lose its enabledness while waiting for the delay to expire, and the firing itself does never consume time. Both transition types have a higher priority than stochastic transitions, but a lower priority than immediate transitions. Conflicts between deterministic and scheduled transitions are solved non-deterministically.

When referring to \mathcal{XSPN} , we do not usually distinguish whether a model

incorporates marking-dependent arc weights or not, but if one wants to make it explicit, the appropriate name is $\mathcal{M}\mathcal{XSPN}$ (\mathcal{XSPN} with marking-dependent arc weights).

Definition 23 (Extended stochastic Petri net). An extended stochastic Petri net (\mathcal{XSPN}) is a tuple $\mathcal{XSPN} = (P, T, A, G, V, H, m_0)$ where:

1. P is a finite set of places.
2. $T = T_s \cup T_i \cup T_d$ is a finite set of transitions with:
 - T_s a set of stochastic transitions,
 - T_i a set of immediate transitions,
 - T_d a set of timed transitions with a deterministic time delay,
 - T_p a set of scheduled transitions, which may fire at predefined time points.
3. P and T satisfy $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$.
4. m_0 is an initial marking.
5. G is the set of marking-dependent arc weights.
6. $A = A_s \cup A_r \cup A_i \cup A_e \cup A_z \cup A_m$ is a multi-set of arcs.
7. H is the set of marking-dependent functions.
8. $V = V_s \cup V_i \cup V_d \cup V_p$ is a set of functions with:
 - $V_s: T_s \rightarrow H$ assigns to each stochastic transition a stochastic rate function $V_s(t) = h_{t_s}$,
 - $V_i: T_i \rightarrow H$ assigns to each immediate transition a weight function $V_i(t) = h_{t_i}$,
 - $V_d: T_d \rightarrow H$ assigns to each deterministic transition a deterministic time delay $V_d(t) = h_{t_d}$,
 - $V_p: T_p \rightarrow \mathbb{R}_0^+$ assigns to each scheduled transition an absolute time point.

The semantics of an \mathcal{XSPN} is non-Markovian.

«

The addition of transitions with deterministic time delays destroys the Markov property and thus the underlying process is non-Markovian. Nevertheless, it is possible to analyse extended stochastic Petri nets, for details see [Ger01; Haa03; Hei+09]. In this thesis we focus on simulative techniques.

Figure 2.9 shows the graphical representations of the net elements of an extended stochastic Petri net.

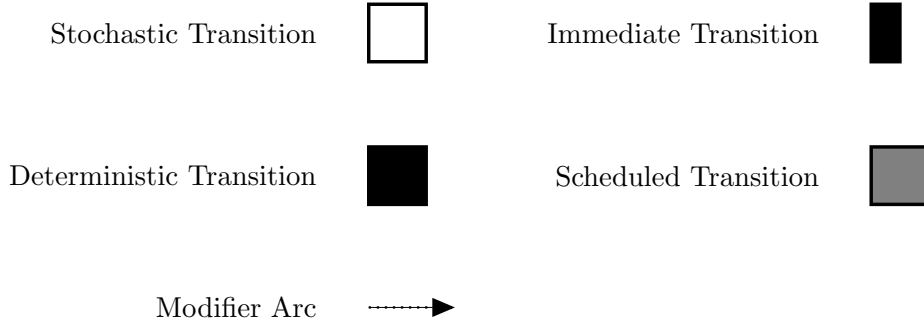


Figure 2.9: Additional elements of an extended stochastic Petri net.

2.9 Coloured Petri Net

The need for high-level description of complex systems arose over the time. Therefore Kurt Jensen introduced coloured Petri net in [Jen81] as a high-level description of Petri nets.

We briefly recall the definition of coloured Petri nets according to [Liu12].

Definition 24 (Coloured Petri net). A coloured Petri net (\mathcal{PN}^c) is a tuple $\mathcal{PN}^c = (P, T, A, \Sigma, C, g, f, m_0)$, where:

1. $P = \{p_1, p_2, \dots, p_m\}$ is a finite set.
2. $T = \{t_1, t_2, \dots, t_n\}$ is a finite set.
3. P and T satisfy $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$.
4. A is a finite set of directed arcs.
5. Σ is a finite, non-empty set of colour sets.

6. $C : P \rightarrow \Sigma$ is a colour function that assigns to each place $p \in P$ a colour set $C(p) \in \Sigma$.
7. $g : T \rightarrow EXP$ is a guard function that assigns to each transition $t \in T$ a guard expression of the Boolean type.
8. $f : A \rightarrow EXP$ is an arc function that assigns to each arc $a \in A$ an arc expression of a multiset type $C(p)_{MS}$, where p is the place connected to the arc a .
9. $m_0 : P \rightarrow EXP$ is an initialization function that assigns to each place $p \in P$ an initialization expression of a multiset type $C(p)_{MS}$. «

The other definitions of coloured net classes can be found in [Liu12]. Every Petri net with respect to [Liu12] can be unfolded into an uncoloured Petri net. This enables us to analyse each coloured Petri net with the same methods as an uncoloured Petri net that includes the methods presented in this thesis. The prerequisite for this is that the unfolding time is negligible compared to the analysis' run-time. In order to achieve this we recently developed an efficient unfolding method based on Interval Decision Diagrams (unpublished). Details are beyond the scope of this thesis. That means, if we refer to a specific Petri net class, e.g., stochastic Petri nets, then everything said applies to its coloured pendant as well.

2.10 Closing Remarks

Petri nets are a mathematical formalism to model and analyse concurrent systems. Due to several extensions they could be used in a wide range of scenarios, e.g., systems and synthetic biology [Nap+09; Blä+14; Bal+10; BP03; Cha07; Doi+99; GP98; GH06; HGD08; HDG10; HG11; LH14; Pec98], technical systems [Dur+04; GBC07; HDS99; IT90; ADN89; YV99], business processes [ADO00; TFZ09; HB03; DDO08] or software development [ACR01; Bal+04; ZC06] to name a few.

In this chapter, we defined the net classes used in this thesis. They range from standard Petri nets and stochastic Petri nets, via more expressive, but less common extended Petri nets and generalized \mathcal{SPN} , to rather unusual marking-dependent \mathcal{XPN} and \mathcal{XSPN} .

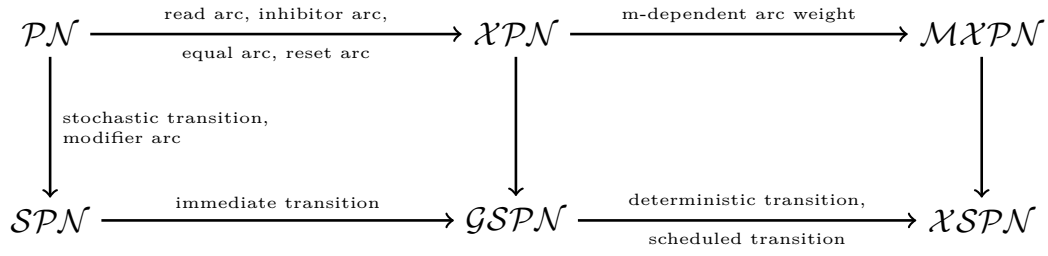


Figure 2.10: Overview of net classes defined in this section, with the extensions leading from one net class to the other.

Modelling and simulation of the net class \mathcal{MXP} (\mathcal{XSPN} with marking dependent arc weights) goes far beyond the standard expressibility of stochastic models; it is supported by Snoopy, but not by MARCIE; so not all material covered in this chapter is available in MARCIE. \mathcal{MXP} have been used in a couple of case studies, not reported in this thesis; see [HSH13]. Figure 2.10 gives an overview on the presented net classes and shows the extensions leading from one net class to the other, starting with Petri nets in the upper left.

The analysis of Petri nets based on the reachability graph and of stochastic Petri nets based on the continuous-time Markov chain is mostly limited to finite state spaces. Even there, the \mathcal{RG} or the CTMC may be too large to be computed. Symbolic techniques based on decision diagrams [Tov08; Sch14] have extended the limit, but there is still and will always be a limit. In order to analyse Petri nets with a huge state space ($|R| \gg 10^{10}$) or an infinite state space we have to use simulation techniques. In the next chapter we recall the most common stochastic simulation algorithms, introduce a new approximate stochastic simulation algorithm and discuss several optimizations and pitfalls.

Chapter 3

Stochastic Simulation

The traditional approach to investigate the time evolution of biochemical reaction networks is by solving a set of coupled ordinary differential equations. One equation per species; each equation embodies the change of the specie's concentration over time with respect to the stoichiometry and kinetic rate constants of the chemical reactions it is involved in. This deterministic formulation is valid in most situations, but there are cases, e.g., non-linear systems, where it is not. In such cases, stochastic formulation of the chemical kinetics gives correct results. Moreover, the stochastic approach is valid in the same situations as the deterministic approach, but it is sometimes even valid, when the deterministic is not, see [OSW69; Kur72]. Therefore, stochastic modelling has become an important tool to fully understand the system behaviour of such reaction networks. A summary of the various stochastic approaches and applications to chemical reaction networks was published by McQuarrie [McQ67]. That article gave rise to what is known today as the *chemical master equation* (CME) and the *stochastic simulation algorithm* (SSA) [Gil76].

In the following sections we review the stochastic simulation algorithm and its various derivatives, namely the *first reaction method*, the *direct method*, the *next reaction method* and the τ -*leaping* method, as well as several optimisation techniques applied to the SSA. Afterwards, we introduce the *discrete-time leap method* for the efficient simulation of stochastic Petri nets, developed by Christian Rohr. It is an approximate simulation method in the same sense as tau-leaping. Moreover, we show how to extend the stochastic simulation algorithms in order to handle not only \mathcal{SPN} but \mathcal{GSPN} and \mathcal{XSPN} . The

important topic of random number generation closes this chapter.

3.1 Stochastic Simulation Algorithm

In biochemical reaction networks, the molecular reactions between the species are random processes, because it is impossible to predict the time at which the next reaction will occur. The stochasticity can be described in a time-dependent manner by the chemical master equation [Gil76]. Gillespie published a rigorous derivation of the chemical master equation (3.1) and showed how the stochastic simulation algorithm fits into it [Gil92].

$$\begin{aligned} \frac{\delta}{\delta\tau} P(m, \tau \mid m_0, \tau_0) = \sum_{t \in T} [h_t(m - \Delta t) P(m - \Delta t, \tau \mid m_0, \tau_0) \\ - h_t(m) P(m, \tau \mid m_0, \tau_0)] \end{aligned} \quad (3.1)$$

In probability theory, this identifies the evolution as a continuous-time Markov chain (CTMC), with the integrated master equation obeying a Chapman-Kolmogorov equation [PP02]. This leads to the following joint density function for the two random variables *time to next firing* and *index of the next transition*:

$$\begin{aligned} P(\tau, t \mid m) d\tau \equiv \text{probability that, given } X(\tau) = m, \text{ transition } t \\ \text{will fire next in the interval } [\tau, \tau + d\tau) . \end{aligned} \quad (3.2)$$

When working with biological systems modelled as \mathcal{SPN} , it may be infeasible to set up the CTMC as the state space $\mathcal{R}_{\mathcal{SPN}}$ can be very large or even infinite. The largeness of CTMCs makes simulation an important analysis technique: instead of computing the CTMC directly, simulation aims at imitating the CTMC by generating different paths of it.

A path of the CTMC is generated in the following way. Starting from the initial marking m_0 , one has to repeatedly fire transitions. In order to fire a transition, one must answer two questions:

1. When will the next transition fire?
2. Which transition will fire next?

So, the enabled transitions in the net compete in a race condition. The fastest one determines the next marking and the simulation time elapsed. In the new marking, the race condition starts anew.

A path through the possibly infinite CTMC is a sequence of discrete random variables $X(\tau)$. The discrete random variable $X_p(\tau)$ describes the number of tokens on place $p \in P$ present at time τ . The system state (marking) at time τ is thus a discrete n -dimensional random vector $X(\tau) = (X_{p_1}(\tau), \dots, X_{p_n}(\tau)) \in \mathcal{X}$. The time $\Delta\tau$ to the next transition is an exponentially distributed random variable with mean $1/E(m)$; the probability density function (pdf) is

$$P(\Delta\tau \mid m) = E(m) \cdot e^{-E(m) \cdot \Delta\tau} . \quad (3.3)$$

The next transition to fire is a discrete random variable with probability mass function (pmf):

$$P(t \mid m) = \frac{h_t(m)}{E(m)} . \quad (3.4)$$

Given the system is in state $X(\tau)$, the probability that a transition $t \in T$ will occur in the time interval $[\tau, \tau + \Delta\tau)$ is given by:

$$\begin{aligned} P(\Delta\tau, t \mid m) &= E(m) \cdot e^{-E(m) \cdot \Delta\tau} \cdot \frac{h_t(m)}{E(m)} \\ &= h_t(m) \cdot e^{-E(m) \cdot \Delta\tau} . \end{aligned} \quad (3.5)$$

Although in principle known a long time before, Gillespie was the first who developed a supporting theory for stochastic simulation of chemical kinetics [Gil76; Gil77]. He presented the Stochastic Simulation Algorithm (SSA; often also called Gillespie's algorithm), which is a Monte Carlo procedure for numerically generating CTMC. Since Gillespie's seminal work, several variants and different implementations and optimisations of the SSA have been proposed. Basically, each variant performs the steps shown in Algorithm 1.

Determining the next time step $\Delta\tau$ (Algorithm 1 line 6) requires to generate a unit-interval uniform random number r_1 and let $\Delta\tau$ be

$$\Delta\tau = \frac{-\ln(r_1)}{E(m)} . \quad (3.6)$$

Selecting the next transition (Algorithm 1 line 7) requires to generate a unit-

Algorithm 1 Generic stochastic simulation algorithm

Require: \mathcal{SPN} with initial marking m_0 , time interval $[\tau_0, \tau_{max}]$
Ensure: marking m at time point τ_{max}

```

1: initRand(seed)
2: time  $\tau \leftarrow \tau_0$ 
3: marking  $m \leftarrow m_0$ 
4: while  $\tau < \tau_{max}$  do
5:   compute transition's rate function
6:   determine next time point  $\tau \leftarrow \tau + \Delta\tau$ 
7:   select transition  $t_j$  to fire depending on current marking  $m$ 
8:   perform firing of transition  $t_j$  and update marking  $m$ 
9: end while

```

interval uniform random number r_2 and find the first transition t_j for which

$$\sum_{i=1}^{j-1} h_{t_i}(m) < r_2 \cdot E(m) \leq \sum_{i=1}^j h_{t_i}(m). \quad (3.7)$$

The SSA simulates every state transition event, one at a time, and updates the system after each state transition. It is worth mentioning here that the generated sequence of state transitions is exact in the sense that the system remains in its current state until the end of the interval and then changes instantaneously. It is not a finite approximation of an infinitesimal time step, as in a standard differential equation solver.

Algorithm 1 generates one possible path through the CTMC, but reliable statements about the system behaviour (variance) can only be made based on many simulations runs. Different realisations of the stochastic process are obtained by different initialisations of the random number generator (Algorithm 1 line 1). After performing N simulation runs and recording the system state at predefined time points τ_{out} , the *average*, *mean* or *expected* number of tokens at τ_{out} is

$$\bar{X}(\tau_{out}) = (1/N) \cdot \sum_{n=1}^N X(n, \tau_{out}) \quad (3.8)$$

One can set the number of simulation runs manually and check whether the results withstand the needs. An alternative and more sophisticated approach to determine the required number of simulation runs is the confidence interval method as described in [SM08]. The confidence interval is specified by defining the confidence level $1 - \alpha$, usually 90%, 95% or 99%, the maximum relative

error β , e.g., 0.1, and the estimated accuracy γ of the results, e.g., 10^{-3} or 10^{-4} . The required number of simulation runs to achieve this confidence interval is calculated by

$$N \geq \frac{z_{1-\alpha/2}^2}{\beta^2} \cdot \frac{1-\gamma}{\gamma} . \quad (3.9)$$

To give an example, using a confidence level of 99%, a maximum relative error of 0.1, the required number of simulation runs to achieve the estimated accuracy of 10^{-5} is $N \geq 3.8 \times 10^7$. Please note that Equation (3.9) shows, the number of required simulation runs increases exponentially with the accuracy. Accelerating simulations is therefore desirable without changing the basic ideas of the algorithm.

Example 8. We demonstrate stochastic simulation on the \mathcal{SPN} in Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$. Figure 3.1 shows the averaged number of tokens on the places *buffer*, *consumer* and *producer* after 10 000 simulation runs.

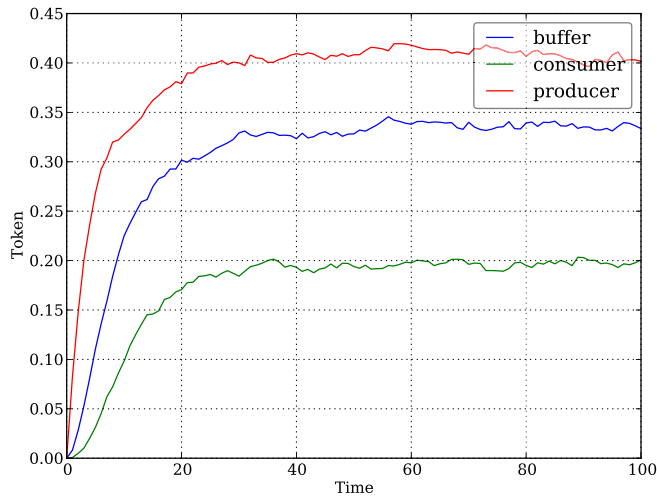


Figure 3.1: Averaged number of tokens of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.

3.2 Direct Method

The direct method, introduced by Gillespie in [Gil76], is an exact stochastic simulation algorithm, because it simulates every transition firing (basically by

using Equation (3.5)) one at a time, and keeps track of the current marking. Therefore every trace computed by the algorithm is an exact path through the corresponding CTMC. The algorithm is shown in Algorithm 2.

Algorithm 2 Direct method

Require: \mathcal{SPN} with initial marking m_0 , time interval $[\tau_0, \tau_{max}]$

Ensure: marking m at time point τ_{max}

```

1: initRand(seed)
2: time  $\tau \leftarrow \tau_0$ 
3: marking  $m \leftarrow m_0$ 
4: while  $\tau < \tau_{max}$  do
5:    $E(m) \leftarrow 0$ 
6:   for all transitions  $t \in T$  do
7:      $E_t(m) \leftarrow h_t(m)$ 
8:      $E(m) \leftarrow E(m) + E_t(m)$ 
9:   end for
10:   $r_1 \leftarrow \text{UNIFORMRANDOMREAL}((0, 1))$ 
11:   $\Delta\tau \leftarrow -\ln(r_1) / E(m)$ 
12:   $\tau \leftarrow \tau + \Delta\tau$ 
13:   $r_2 \leftarrow \text{UNIFORMRANDOMREAL}((0, 1])$ 
14:   $u \leftarrow E(m) \cdot r_2$ 
15:  for all transitions  $t \in T \wedge u > 0$  do
16:     $u \leftarrow u - E_t(m)$ 
17:  end for
18:   $m \leftarrow m + \Delta t$ 
19: end while

```

The idea of the direct method is to generate two random numbers $\{r_1, r_2\}$ uniformly distributed on $(0, 1)$. The first random number r_1 is used to determine an exponential distributed random variable that gives the time increment $\Delta\tau$. The time increment $\Delta\tau$ is computed according to Equation (3.3) using the inversion method (Algorithm 2 line 11). The second random number r_2 is used to select the next transition to fire (Equation (3.4)), this is done by a linear search over the transitions propensities. The random number is scaled by the sum of transition rates $E(m)$ (Algorithm 2 line 14). Now the scaled random number u is used to find the transition. Therefore u is subtracted by the rate of each transition and the transition, in which u becomes non-positive, is the requested one (Algorithm 2 line 15 - 17). This is called the chop-down search, because the value of u is chopped-down until it becomes non-positive.

Computational complexity

In stochastic Petri nets we allow only pre-places of a transition to be part of the transition's rate function. So assuming a loose coupling of the net reveals a complexity of $\mathcal{O}(1)$ for computing the rate function h_t . The computation of all transitions rate functions is $\mathcal{O}(n)$, $n = |T|$. The linear search requires iterating over all transitions and so the complexity is $\mathcal{O}(n)$ too. In summary the overall time complexity of the algorithm is $\mathcal{O}(n)$, because of the linear search used to find the next transition to fire and recomputing the propensities of all transitions in each step.

3.3 Optimised Direct Method

Gillespie roughly described a part to reduce the computational complexity of the direct method already [Gil76], i.e., storing the sum of all propensities and only updating the depended propensities and thus the sum of them while running the simulation. As he remained very vague in this topic, Gibson and Bruck [GB00] introduced a dependency graph for the transitions that declares which propensities have to be updated after a certain transition fires, whereas all others are reused. That reduces the average cost of recomputing the propensities from $\mathcal{O}(n)$ to $\mathcal{O}(1)$.

Many variants of the SSA aim at reducing the computational cost of selecting the next reaction that will occur. Cao et al. [CGP07] statically reordered the list of transitions to keep the transitions with larger propensities at the beginning of the list. The position of each transition in the list is thereby determined after some pre-simulations. McCollum et al. [McC+06] maintain a loosely sorted order of the reactions as the simulation proceeds. However, the time to manage the advanced data structures partially compensates the speed-up due to faster search [CLP04]. The logarithmic direct method was developed by Li and Petzold [LP06]. It maintains a list of partial sums of the propensities and uses binary search on this list to find the next transition to fire. Whereas the binary search has $\mathcal{O}(\log n)$, updating the partial sums has $\mathcal{O}(n)$. Therefore the computational complexity remains $\mathcal{O}(n)$.

Slepoy et al. [STP08] introduced the composition rejection method to achieve a computational complexity independent of the number of transitions. They

achieve this by grouping the transitions depending on the propensities. Let p_{min} be the minimum possible propensity greater than zero. The first group has an upper bound on the propensities of $2 \cdot p_{min}$ and contains all transitions having a propensity in the interval $(0, 2 \cdot p_{min}]$. The second group covers the interval $(2 \cdot p_{min}, 4 \cdot p_{min}]$. So the interval of the n th group is $(2^{n-1} \cdot p_{min}, 2^n \cdot p_{min}]$. The number of groups G can be determined using the maximum possible propensity p_{max} by $G = \log_2(p_{max}/p_{min})$. But it is not possible to compute p_{max} in advance in any case, e.g., having a unbounded \mathcal{SPN} leads to $p_{max} = \infty$ and even for bounded nets the computation of maximum number of tokens in a place is infeasible sometimes. In such cases the number of groups grows dynamically while the simulation takes place. The selection of the next transition to fire is done in two steps. First the group is selected using linear search as in the direct method. Therefore the sum of all propensities and the sums of each group are needed. After selecting the group, the rejection method is used to find the transition. That means, a transition in the group is chosen randomly and if its propensity is greater than a randomly selected portion of the group's upper bound, the transition will be selected. If not this is repeated until a transition is selected. On average, this is 2 times necessary, because the propensities in the groups are at least half their upper bound. While updating the transition rates they may be relocated in a different group and the group's sum have to be updated. The efficiency of the composition rejection method heavily depends on the fact that the number of groups is much less than the number of transitions. If that's the case the computational complexity of this method is $\mathcal{O}(G)$, $G \ll |T|$.

Mauch et al. [MS11] presented another transition selection approach the 2D search. The propensities are stored in a 2D array with $\sqrt{|T|}$ elements per row and an array containing the row sums is needed. The next transition to fire is determined with two linear searches. The first looks for the row and the second searches for the transition in the selected row. The update of the row sum is constant as it is for the total sum, thus the computational complexity of the 2D search is $\mathcal{O}(\sqrt{|T|})$.

An overview on the different transition selection approaches including comparison of the run-time behaviour is given in [MS11].

Algorithm 3 Optimised direct method using 2D - search

Require: \mathcal{SPN} with initial marking m_0 , time interval $[\tau_0, \tau_{max}]$, 2D array E

 with $l \leftarrow \sqrt{|T|}$ transitions per row

Ensure: marking m at time point τ_{max}

```

1: INITRAND(seed)
2: time  $\tau \leftarrow \tau_0$ 
3: marking  $m \leftarrow m_0$ 
4:  $R \leftarrow 0$  ▷ array of row sums
5:  $E(m) \leftarrow 0$ 
6: for all transitions  $t \in T$  do
7:    $E_t(m) \leftarrow h_t(m)$ 
8:    $E(m) \leftarrow E(m) + E_t(m)$ 
9:    $R_{t/l}(m) \leftarrow R_{t/l}(m) + E_t(m)$ 
10: end for
11: while  $\tau < \tau_{max}$  do
12:    $r_1 \leftarrow \text{UNIFORMRANDOMREAL}((0, 1))$ 
13:    $\Delta\tau \leftarrow -\ln(r_1) / E(m)$ 
14:    $\tau \leftarrow \tau + \Delta\tau$  ▷ next time point
15:    $r_2 \leftarrow \text{UNIFORMRANDOMREAL}((0, 1])$ 
16:    $u \leftarrow E(m) \cdot r_2, i \leftarrow 0$ 
17:   for  $i \leftarrow 0, |R| \wedge u > 0$  do
18:      $u \leftarrow u - R_i$ 
19:   end for
20:   for  $i \leftarrow i \cdot l, |T| \wedge u > 0$  do
21:      $u \leftarrow u - E_i(m)$ 
22:   end for
23:    $m \leftarrow m + \Delta t_i$  ▷ fire transition
24:   for all transitions  $t \in T$  affected by  $t_i$  do
25:      $E(m) \leftarrow E(m) - E_t(m) + h_t(m)$ 
26:      $R_{t/l}(m) \leftarrow R_{t/l}(m) - E_t(m) + h_t(m)$ 
27:      $E_t(m) \leftarrow h_t(m)$ 
28:   end for
29: end while

```

Computational complexity

From this set of optimizations we've incorporated the 2D search for transition selection, sparse arrays for transition firing and the dependency graph for transition updates in our optimized direct method (Algorithm 3). Thus the overall time complexity of the algorithm is $\mathcal{O}(\sqrt{|T|})$.

3.4 First Reaction Method

Gillespie introduced an alternate algorithm in [Gil76], the first reaction method. It is an exact stochastic simulation algorithm, like the direct method, because it differs only in the way of computing the random deviates. Therefore every trace computed by the algorithm is an exact path through the corresponding CTMC.

The time $\Delta\tau$ to the next transition (Equation (3.3)) can be rewritten for each transition. So that

$$P_t(\Delta\tau \mid m) = h_t(m) \cdot e^{-h_t(m) \cdot \Delta\tau} \quad (3.10)$$

is the probability that transition t occurs in the time interval $(\tau, \tau + \Delta\tau)$ under the assumption that no other transition fires in the time interval $(\tau, \tau + \Delta\tau)$. This provides us a preliminary transition firing time $\Delta\tau_t$ for each transition according to Equation (3.10). We use the inversion method to compute the preliminary times for all transitions (Algorithm 4 line 8). From these preliminary times, the smallest time will be the next time step (Algorithm 4 line 14) and the corresponding transition will fire next (Algorithm 4 line 15).

Algorithm 4 First reaction method

Require: SPN with initial marking m_0 , time interval $[\tau_0, \tau_{max}]$

Ensure: marking m at time point τ_{max}

```

1: initRand(seed)
2: time  $\tau \leftarrow \tau_0$ 
3: marking  $m \leftarrow m_0$ 
4: while  $\tau < \tau_{max}$  do
5:    $\Delta\tau_{min} \leftarrow \infty$ 
6:   for all transitions  $t \in T$  do
7:      $r \leftarrow \text{UNIFORMRANDOMREAL}((0, 1))$ 
8:      $\Delta\tau_t \leftarrow -\ln(r) / h_t(m)$ 
9:     if  $\Delta\tau_t < \Delta\tau_{min}$  then
10:       $\Delta\tau_{min} \leftarrow \Delta\tau_t$ 
11:       $t_{min} \leftarrow t$ 
12:     end if
13:   end for
14:    $\tau \leftarrow \tau + \Delta\tau_{min}$ 
15:    $m \leftarrow m + \Delta t_{min}$ 
16: end while
```

Computational complexity

The first reaction method has to compute $|T|$ random numbers in each step. The overall time complexity of the algorithm is $\mathcal{O}(n)$, $n = |T|$, because the propensities and the time steps for each transition are recomputed in each step.

3.5 Next Reaction Method

The next reaction method introduced by Gibson & Bruck in [GB00] is an adaptation of the first reaction method. It computes the time at which each transition will occur, just like the first reaction method. But in contrast to it, times are not computed anew at each time step, but re-used, if the transitions were not affected in the last step. Only these transitions are updated, where the amount of tokens on their pre-places had changed. This is achieved by the use of a dependency graph to determine the affected transitions. The transition times and indices are stored in an indexed priority queue, where transitions are ordered according to their time and the root element has always the minimal time. The next reaction method's algorithm is given in Algorithm 5.

Computational complexity

We assume a loose coupling of the net, this reveals a complexity of $\mathcal{O}(1)$ for computing the rate function h_t and the transition times τ_t . Inserting or changing a value in the priority queue has a time complexity of $\mathcal{O}(\log n)$. Getting the next time point is $\mathcal{O}(1)$, because it's the root of the queue. The index of the root element represents the transition, which means selecting the next transition is $\mathcal{O}(1)$, too. Therefore the overall time complexity of the next reaction method is $\mathcal{O}(\log n)$.

3.6 Tau-Leaping Method

An approximate speedup to the SSA is provided by τ -leaping [Gil01], in which time t is advanced by a preselected amount τ and the numbers of firings of the individual transitions during the time interval $[t, t + \tau)$ are approximated by

Algorithm 5 Next reaction method

Require: \mathcal{SPN} with initial marking m_0 , time interval $[\tau_0, \tau_{max}]$, indexed priority queue PQ , propensities vector \underline{a} , times vector $\underline{\tau}$

Ensure: marking m at time point τ_{max}

```

1: initRand(seed)
2: time  $\tau \leftarrow \tau_0$ 
3: marking  $m \leftarrow m_0$ 
4: for all transitions  $t \in T$  do
5:    $E_t(m) \leftarrow h_t(m)$ 
6:    $r \leftarrow \text{UNIFORMRANDOMREAL}((0, 1))$ 
7:    $\tau_t \leftarrow -\ln(r) / E_t(m)$ 
8:    $PQ.\text{insert}(t, \tau_t)$ 
9: end for
10: while  $\tau < \tau_{max}$  do
11:    $t, \tau_t \leftarrow PQ.\text{minimumElement}()$ 
12:    $\tau \leftarrow \tau_t$ 
13:    $m \leftarrow m + \Delta t$ 
14:   for all transitions  $k \in T \setminus t$  affected by  $t$  do
15:      $\tau_k \leftarrow (E_k(m) / h_k(m)) \cdot (\tau_k - \tau) + \tau$ 
16:      $E_k(m) \leftarrow h_k(m)$ 
17:      $PQ.\text{update}(k, \tau_k)$ 
18:   end for
19:    $E_t(m) \leftarrow h_t(m)$ 
20:    $r \leftarrow \text{UNIFORMRANDOMREAL}((0, 1))$ 
21:    $\tau_t \leftarrow -\ln(r) / E_t(m) + \tau$ 
22:    $PQ.\text{update}(t, \tau_t)$ 
23: end while

```

Poisson random numbers. Thus, instead of (sequentially) tracing every single state transition, several reactions are executed in parallel. With τ -leaping, it is assumed that all propensity functions are approximately constant in $[t, t + \tau)$, which is referred to as the leap condition. Approximating the number of transition firings by Poisson random numbers has to be done very carefully, because the Poisson distribution is infinite. So, it may compute transition firings greater than the enabledness of the transition, which result in incorrect markings. To ensure this, it is important to select τ sufficiently small, but also large enough to accelerate simulation. Several improvements were done in selecting an appropriate τ , see [CGP06; CGP07]. Approximative stochastic simulation is an ongoing research subject and besides τ -leaping, other *leaping* methods arose, e.g., binomial leap methods [TB04], R-leaping [ACK06] or K-

leaping [CX07].

Computational complexity

The computational complexity of the τ -leaping method is hardly comparable with the exact SSA's, because it *leaps* over certain steps, whereas the others take one step after the other. The single step complexity of τ -leaping is $\mathcal{O}(|T|)$, because for each transition the number of firings in the leap have to be computed. But the advantage of this method is reducing the number of steps needed to finish the simulation and thus it should take less run time than the exact methods.

3.7 Discrete-Time Leap Method

The discrete-time leap method developed by Christian Rohr [Roh16] (δ -leaping for short) aims at the simulation of stochastic Petri nets used for modelling biochemical reaction networks. In an attempt to decrease the time complexity of the simulation algorithm, we exploit the uniformization of the underlying CTMC in combination with the maximum firing rule.

The idea of converting a CTMC into a DTMC goes back to [Jen53]. Similar methods are known as *uniformization* or *randomization*, see [Ste94]. The DTMC is defined stochastically identical to the CTMC, i.e., the original CTMC is represented by a DTMC where the times are implicitly driven by a Poisson process. It can be shown that this DTMC behaves equivalently to the CTMC [San08].

Generating paths through the DTMC is as expensive as for the CTMC and we would not gain any efficiency by doing it in an exact way. That's why, we are leaping over several states. That means, all enabled transitions that are not mutually exclusive, are forced to fire within one leap. When the net is filled up with tokens, every transition will fire within every leap. Furthermore, we embody the maximum firing rule and let each transition fire concurrently to itself.

3.7.1 Transition firing

How often a transition is allowed to fire concurrently depends on its enabledness degree and is determined randomly at each step.

$$\text{firing rate} \cong \text{random}[0, \text{enableness degree}] \quad (3.11)$$

The construction of the DTMC induces that the times between transitions are all exponentially distributed. Hence, these times are randomized by a Poisson process. Since for the uniformized DTMC the number of transitions in any time interval of length δ has a Poisson distribution with rate λ . The Poisson distribution with rate λ is an approximation of the bounded discrete binomial distribution with two parameters k and pr according to the Poisson limit theorem:

$$\lambda = k \cdot pr . \quad (3.12)$$

The binomial distribution is used to model the number of successes in a sequence of k independent yes/no experiments with a probability pr to succeed. In our case, the enabledness degree corresponds to the sequence's length $k = \text{ed}_t$. The success probability pr is deduced from Equation (2.2), because of the exponentially distributed times between transitions. Given out of ed_t maximum firings and a firing rate h_t , we compute the probability pr for transition t in marking m for δ units of time as follows

$$pr = \begin{cases} 1 - e^{-\frac{h_t(m)}{\text{ed}_t(m)} \cdot \delta} & \text{ed}_t(m) > 0 \\ 0 & \text{otherwise.} \end{cases} \quad (3.13)$$

Thus the number of transition firings in the discrete-time leap method is a sample value of the binomial random variable $\mathcal{B}(\text{ed}_t, 1 - e^{-\frac{h_t(m)}{\text{ed}_t(m)} \cdot \delta})$ under the condition

$$0 \leq \frac{h_t(m)}{\text{ed}_t(m)} \cdot \delta \leq 1 . \quad (3.14)$$

This leads us to a good approximation of the exact stochastic simulation results. Even a violation of the condition in Equation (3.14) would not lead to negative values or incorrect markings (states), i.e., in any case a marking is reached that is part of the model's state space. But the temporal behaviour

of the model, simulated with δ -leaping, would not coincide anymore with the behaviour of exact stochastic simulation algorithms. This may be an indication for one of two situations. First, the model's time-scale is smaller than the chosen δ , i.e., reducing the δ would gain better approximation. Second, some transition's rate functions are not scaled correctly, i.e., stochastic reaction rates have to be scaled with respect to their reaction order, see [Wil06].

Comparing stochastic simulation results is difficult, because each run is somewhat different due to the inherent randomness. But we can apply the law of large numbers that states, the sample average converges to the expected value

$$\bar{X}_n \rightarrow \mu \text{ when } n \rightarrow \infty . \quad (3.15)$$

This holds for exact as well as approximate stochastic simulation algorithms and we can use the sample mean \bar{X} to calculate the approximation error. For that reason we computed the average of 1 000 000 simulation runs for the comparison. Now, the approximation error of δ -leaping compared to exact stochastic simulation algorithms is determined by the absolute error

$$\epsilon = |\bar{X}_{exact} - \bar{X}_{approx}| \quad (3.16)$$

and by the relative error

$$\eta = \frac{\epsilon}{\bar{X}_{exact}} \text{ for } \bar{X}_{exact} \neq 0 . \quad (3.17)$$

We compute the approximation error for each place at each observed time point and are able to observe the development of it over time. In most cases the relative error gives meaningful results, but if the average number of tokens is close to zero the absolute error is suited better. There is surely space for discussion on how significant the reported approximation error is and the interpretation may be subjective, but a relative error below 0.05 can be seen as sufficiently good.

We compare the simulation results of δ -leaping, with $\delta = 0.1$, against the direct method [Gil76] on the most common reaction types in biochemical reaction networks, i.e., first and second order reactions. The first order reaction $P1 \rightarrow P2$ is shown in Figure 3.2. It uses mass-action kinetics with rate con-

stant 1. The results of δ -leaping match the results of the direct method. The approximation error is very close to zero for place $P2$ and below 0.02 for place $P1$. The increase of the relative error for $P1$ is caused by $P1$ approaching zero. A place having an average number of tokens close to zero is meant to be part of rare events. Thus, an increase in the number of simulation runs, would reduce the approximation error even further.

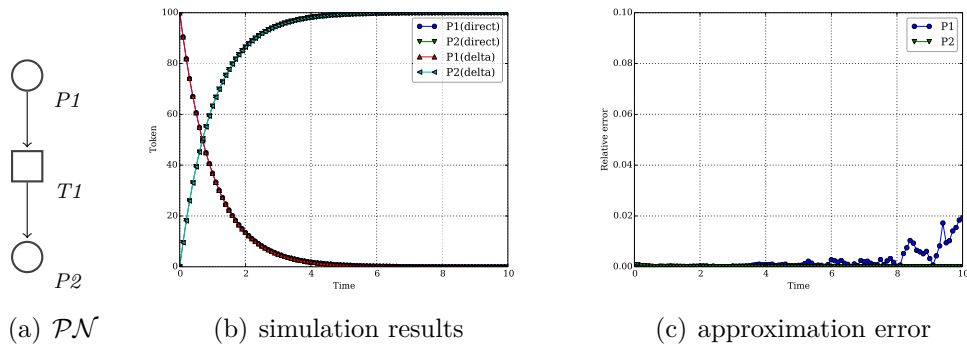


Figure 3.2: First order reaction: $P1 \rightarrow P2$

The second order reaction $P1 + P2 \rightarrow P3$ uses mass-action kinetics with rate constant 0.1. The results shown in Figure 3.3 are quite as good as in Figure 3.2. The approximation error for place $P1$ and $P3$ goes down to zero and it is slightly higher for place $P2$, but still very small.

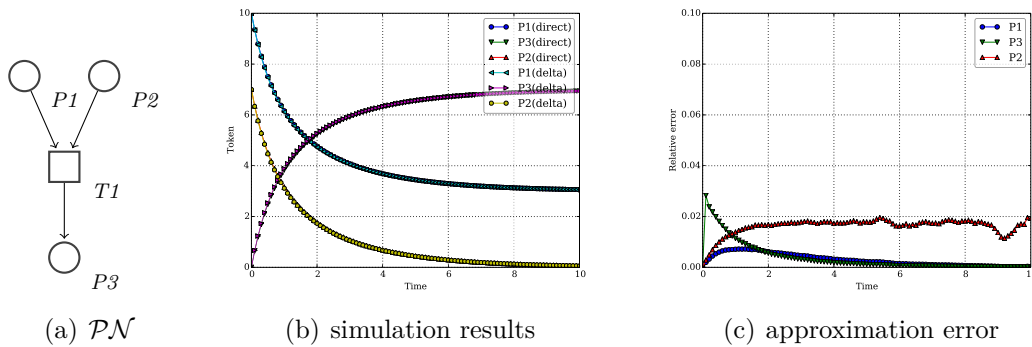


Figure 3.3: Second order reaction: $P1 + P2 \rightarrow P3$

3.7.2 Dependent Subnets

We discussed the firing of single and independent transitions, but a typical model consists of much more than that. There are two types of subnets that need some attention: *conflicts* and *sequences*.

A conflict exists inside a Petri net, if two or more transitions share a pre-place, see Figure 3.4. If the amount of tokens on this place is just as much as the arc weights, one has to determine, which transition will fire. This is not an issue in the stochastic simulation algorithm. In an exact stochastic simulation only one transition is selected at a time and there is no need for further conflict resolution. As in the standard conflict resolution for Petri nets, we have to choose a transition non-deterministically. The standard conflict resolution proposes a non-deterministic selection according to a uniform distribution on the number of affected transitions. But this is not the best solution for us, because this does not pay attention to the firing rates of the transitions. We are aiming at a *weighted* non-deterministic selection, which accounts for the firing rates of transitions as well.

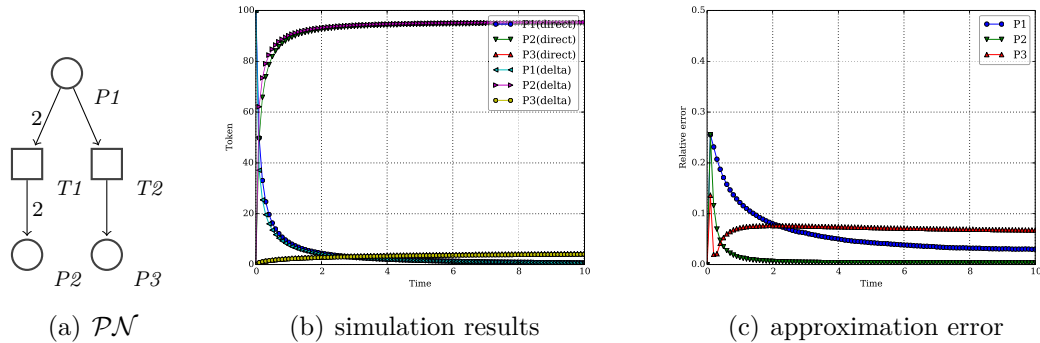
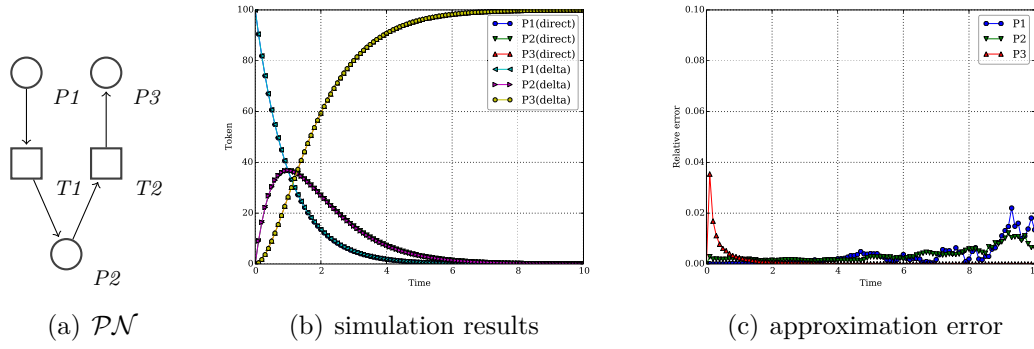


Figure 3.4: Conflict: $P1 \rightarrow P2$ and $P1 \rightarrow P3$

The handling of transition sequences, see Figure 3.5, is closely related to the conflict resolution. We embody the maximum firing rule and force every transition to fire (if enabled) in one time step. We shuffle the transitions and let them fire (if enabled) sequentially to approximate the stochastic behaviour.

Luckily, we can treat both issues, transitions in conflict or in sequence, in one solution. We generate a weighted random sequence of all transitions $t \in T$ in each step and let them fire (if enabled) sequentially. The serial firing

Figure 3.5: Sequence: $P1 \rightarrow P2 \rightarrow P3$

precludes additional attempts to solve conflicts. Our algorithm is based on the modern version of the Fisher–Yates shuffle [F+63] introduced in [Dur64]. Algorithm 6 incorporates Bernoulli sampling to realize a shuffling in accordance to transition weights. The weight computation in Equation (3.19) is based on a conflict marking, i.e., each place contains the minimal number of tokens so that each of its post-transitions becomes enabled. Then the transition weight is derived from the transition rate function evaluated on the conflict marking.

$$m_w(p) = \max_{t \in p^\bullet} (f(p, t)) \quad (3.18)$$

$$w_t = h_t(m_w) \quad (3.19)$$

Algorithm 6 Weighted random shuffle

Require: transition sequence T , transition weights W , $|T| = |W| = n$

Ensure: shuffled transition sequence

```

1: procedure WEIGHTEDRANDOMSHUFFLE(transitions  $T$ , weights  $W$ )
2:   for  $i = n; i > 1; i \leftarrow i - 1$  do
3:      $r \leftarrow \text{UNIFORMRANDOMINTEGER}([1, i - 1])$ 
4:      $t_1 \leftarrow T_{n-i}, t_2 \leftarrow T_{n-i+r}$ 
5:      $w \leftarrow W_{t_2} / (W_{t_1} + W_{t_2})$ 
6:     if  $\text{BERNOULLISAMPLING}(w) = 1$  then
7:        $\text{SWAP}(T_{n-i}, T_{n-i+r})$ 
8:     end if
9:   end for
10:  return  $T$ 
11: end procedure

```

We get quite meaningful results using our weighted random shuffle algorithm

for the simulation of conflicts (Figure 3.4) and sequences (Figure 3.5). The results for sequences are very close to exact stochastic simulation, but the approximation error for conflicts is higher than in any other case before, but still acceptable. This gives us the order to continue working on this subject and to refine it further.

3.7.3 Algorithm

So far, we discussed the essential steps of the discrete-time leap method. Here, we integrate the various steps in one algorithm, which is given in detail in Algorithm 7.

Algorithm 7 δ -leaping algorithm

Require: \mathcal{SPN} with initial marking m_0 , time interval $[\tau_0, \tau_{max}]$, time step δ , runs r_{max} , weights W

Ensure: marking m at time point τ_{max}

```

1: for  $r = 0$ ;  $r < r_{max}$ ;  $r \leftarrow r + 1$  do
2:    $time \tau \leftarrow \tau_0$ 
3:    $marking m \leftarrow m_0$ 
4:    $T_r \leftarrow T$ 
5:   while  $\tau \leq \tau_{max}$  do
6:      $T_r \leftarrow \text{WEIGHTEDRANDOMSHUFFLE}(T_r)$ 
7:     for all transitions  $t_j \in T_r$  do
8:        $k \leftarrow \text{ENABLEDNESSDEGREE}(t_j, m)$ 
9:        $h \leftarrow \text{FIRINGRATE}(t_j, m)$ 
10:      if  $k > 0$  then
11:         $f \leftarrow \text{BINOMIALSAMPLING}(a, (1 - e^{-\frac{h}{k} \cdot \delta}))$ 
12:         $m \leftarrow m + f \cdot \Delta t_j$ 
13:      end if
14:    end for
15:     $\text{GENERATERESULTPOINT}(\tau, m)$ 
16:     $\tau \leftarrow \tau + \delta$ 
17:  end while
18: end for
```

Each simulation run starts with the initialization phase (Algorithm 7, line 2–4), where the simulation time, the marking, and the transition sequence are set. The next step is the generation of the weighted random sequence of transitions using the *weightedRandomShuffle* algorithm (Algorithm 7, line 6). After that for each transition the following steps are done (Algorithm 7, line 7–14),

compute the enabledness degree k , the firing rate h , and pick a random number f according to the binomial distribution defined by k and Equation (3.13), finally the transition fires f -times. The simulation time is elapsed by δ time unit. All these steps are performed until the end of the simulation time interval τ_{max} is reached. Just like for stochastic simulation, one has to do several simulation runs, in order to get reasonable results. The overall result is the mean of all runs.

3.7.4 Caveat

The presented results of the discrete-time leap method approximate the stochastic simulation algorithm quite well. This might not be true for all kinds of biological mechanisms in a stochastic Petri net model, as it is the case for, e.g., a simplified birth-death process, shown in Figure 3.6. The result of the

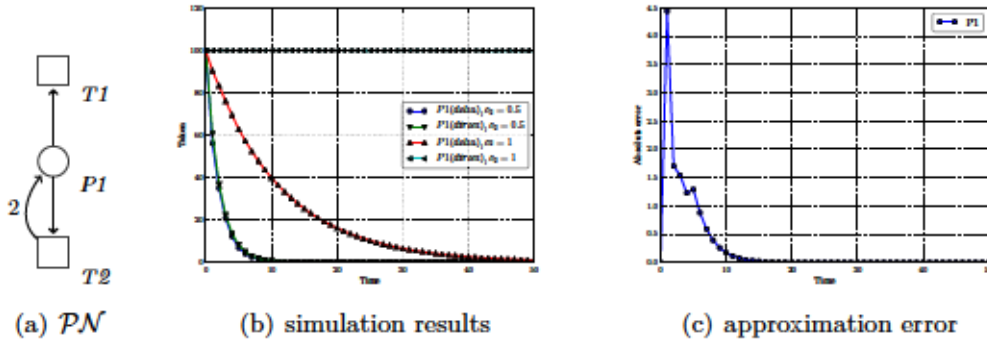


Figure 3.6: Simplified birth-death process, it shows the results for different rate constants of $T2$, i.e., $c_{T2} = 1$ (blue) and $c_{T2} = 0.5$ (green)

stochastic simulation using mass-action kinetics with a rate constant of 1 shows a steady state of $P1$ at the initial marking. Transitions $T1$ and $T2$ are both likely to fire and thus, fire alternately in average. The token consumed by $T1$ is produced by $T2$.

Due to the maximum firing rule, the result of the discrete-time leap method differs. Let us assume an initial marking of 100 tokens in $P1$ and each transition consumes 50% of the tokens in each firing for the purpose of demonstration. There exist two possible firing sequences $S_1 = [T1, T2]$ and $S_2 = [T2, T1]$. The execution of S_1 results in the following path $m_0(100) \xrightarrow{T1} m_1(50) \xrightarrow{T2}$

$m_2(75)$ and S_2 generates $m_0(100) \xrightarrow{T_2} m_1(150) \xrightarrow{T_1} m_2(75)$. Both sequences decrease the amount of tokens on $P1$. So $P1$ approaches zero tokens in the long term in any case. This contradicts the exact stochastic simulation results. Equal results can be obtained by adapting the rate constant of the stochastic transition, i.e., setting the rate constant of T_2 to 0.5.

Computational complexity

The computational complexity of the δ -leaping method is comparable with the τ -leaping method, because it *leaps* over certain steps, whereas the other takes one step after the other. The single step complexity of δ -leaping is $\mathcal{O}(|T|)$, because all transitions are shuffled in each step and for each transition the number of firings in the step has to be computed. But the advantage of this method is the reduction of the number of steps needed to finish the simulation and thus it should take less run time than the exact methods. The overall number of steps in δ -leaping is typically lower than in τ -leaping, because it can perform wider leaps without the risk of getting negative markings.

3.8 Extensions

The presented stochastic simulation algorithms so far are able to simulate SPN but not $GSPN$ or $XSPN$. The simulative processing of immediate, deterministic and scheduled transitions is rather straightforward, see [Ger01]. In short, the stochastic simulation algorithms need to be extended in two ways.

1. After every firing of a stochastic transition, it needs to be checked whether immediate transitions got enabled. If so, these have to be processed until no more immediate transition is enabled. This possibly leads to a time deadlock, if there exists a cyclic path of immediate transitions.
2. Having calculated the next time step, it needs to be checked whether a deterministic or scheduled transition is enabled in the time interval $[\tau, \tau + \Delta\tau]$. If yes, the one closest to τ is processed and the simulation time will be set to the value of this transition.

3.8.1 Immediate Transitions

We introduce the procedure *checkImmediateTransitions* (Algorithm 8), which checks for enabled immediate transitions and let them fire according to their weights. The procedure has to be used on the initial marking and each time after firing a stochastic transition, e.g., direct method Algorithm 2 line 18, optimized direct method Algorithm 3 line 23, first reaction method Algorithm 4 line 15 and next reaction method Algorithm 5 line 13.

Algorithm 8 Check immediate transitions

Require: \mathcal{GSPN} with vanishing marking m at time point τ

Ensure: tangible marking m at time point τ

```

1: procedure CHECKIMMEDIATETRANSITIONS(marking  $m$ )
2:   while  $\exists t \in T_i \wedge m \geq t^- \wedge m \geq t_r^- \wedge m < t_i^-$  do
3:      $W(m) \leftarrow 0$ 
4:     for all transitions  $t \in T_i$  do
5:        $W_t(m) \leftarrow h_t(m)$ 
6:        $W(m) \leftarrow W(m) + W_t(m)$ 
7:     end for
8:      $u \leftarrow W(m) \cdot \text{UNIFORMRANDOMREAL}((0, 1])$ 
9:     for all transitions  $t \in T_i \wedge u > 0$  do
10:       $u \leftarrow u - W_t$ 
11:    end for
12:     $m \leftarrow m + \Delta t$ 
13:  end while
14:  return  $m$ 
15: end procedure

```

The weights of immediate transitions define the ratio of the probability mass processed by each of them. The sum of weights of all enabled immediate transitions in marking m is denoted by

$$W(m) = \sum_{t \in T_i} h_t(m) . \quad (3.20)$$

The next immediate transition to fire is a discrete random variable [Ajm+95] with probability mass function (pmf):

$$P(t \mid m) = \frac{h_t(m)}{W(m)} . \quad (3.21)$$

The discrete random variable (Equation (3.21)) is equivalent to Equation (3.4),

thus Algorithm 8 applies a linear search for its computation as in Algorithm 2. This is an appropriate solution, because usually $|T_i| \ll |T_s|$. It may be useful to apply advanced methods like in Algorithm 3, if necessary.

The sojourn time of vanishing states is always zero, that's why immediate transitions can cause a time deadlock (Sec. 2.7), if there exists a cyclic patch of immediate transitions and one of these gets enabled. This is represented by Algorithm 8 line 2, resulting in an infinite loop without any progress in time. Such a modelling fault can be discovered by checking the net structure and looking for cycles of immediate transitions in the graph [Joh75].

3.8.2 Deterministic and Scheduled Transitions

The simulation of timed (deterministic and scheduled) transitions requires us to keep track of their timers. Each timed transition has its own timer, which is deactivated by default. Once a timed transition gets enabled, its timer becomes activated. The timer is initialized with the time point τ , when the timed transition got enabled. In case of a deterministic transition t_d , the timer runs up τ_{t_d} , derived from the enabling time point τ and the deterministic time delay $h_{t_d}(m)$:

$$\tau_{t_d} = \tau + h_{t_d}(m) . \quad (3.22)$$

If t_d loses its enabledness, the timer is reset and deactivated until it gets enabled again. The timer of the scheduled transition t_p runs up to the absolute time point τ_{t_p} defined in its schedule. If t_p loses its enabledness, the timer is reset and deactivated. The last used time point τ_{t_p} is removed from t_p 's schedule and the next time point greater than τ' in the schedule is used, if t_p gets enabled at τ' . When the timer runs off, the timed transitions are forced to fire and if they are still enabled after firing, their timers are restarted, otherwise reset and deactivated. A conflict, between two or more enabled timed transitions at the same time point, is solved non-deterministic. If a timed transition t_d is in conflict with an immediate transition t_i , e.g., $h_{t_d} = 0$, t_i is going to fire, because immediate transitions have a higher priority than timed transitions. Deterministic transitions having a time delay of zero are able to cause a time deadlock in the same way as immediate transitions, and should be used only with special care. Stochastic transitions have a lower

priority than timed transitions, i.e., the timed transition will fire in case of a conflict at time point τ .

Algorithm 9 Check deterministic and scheduled transitions

Require: \mathcal{XSPN} marking m at time point τ

Ensure: marking m after firing timed transitions at time point τ'

```

1: procedure CHECKTIMEDTRANSITIONS(marking  $m$ , time  $\tau$ )
2:    $\Lambda \leftarrow \emptyset$ 
3:    $\tau' \leftarrow \tau$ 
4:   for all transitions  $t \in T_d \cup T_p$  do
5:     if FINISHED( $t_{timer}$ ) then
6:       if TIME( $t_{timer}$ ) <  $\tau$  then
7:          $\tau' \leftarrow \text{TIME}(t_{timer})$ 
8:          $\Lambda \leftarrow t$ 
9:       else if TIME( $t_{timer}$ ) =  $\tau$  then
10:         $\Lambda \leftarrow \Lambda \cup t$ 
11:      end if
12:    end if
13:  end for
14:  if  $\Lambda \neq \emptyset$  then
15:     $\tau \leftarrow \tau'$ 
16:    repeat
17:       $u \leftarrow \text{UNIFORMRANDOMINTEGER}([1, |\Lambda|])$ 
18:       $t \leftarrow \Lambda_u$ 
19:      if  $m \geq t^- \wedge m \geq t_r^- \wedge m < t_i^-$  then
20:         $m \leftarrow m + \Delta t$ 
21:         $m \leftarrow \text{CHECKIMMEDIATETRANSITIONS}(m)$ 
22:      end if
23:       $\Lambda \leftarrow \Lambda \setminus t$ 
24:    until  $\Lambda = \emptyset$ 
25:  end if
26:  return  $m, \tau$ 
27: end procedure

```

We show the procedure *checkTimedTransitions* in Algorithm 9. It needs to be used after computing the next time step in the stochastic simulation algorithms, e.g., direct method Algorithm 2 line 11, optimized direct method Algorithm 3 line 13, first reaction method Algorithm 4 line 14 and next reaction method Algorithm 5 line 11.

3.9 Random Number Generation

Each of the presented simulation algorithms uses random deviates at some points, e.g., determining the next time step, selecting the next transition to fire, shuffling the transition sequence and so on. The results of simulation algorithms highly depend on the computation of random deviates, both in terms of correctness and speed.

The random deviates used here are not considered to be *truly* random, but are created using Pseudo-Random Number Generators (PRNGs). These are deterministic algorithms creating a sequence of *random* numbers without an external source of entropy. They operate on an internal state (*seed*), whose initialization is often crucial to its results. Initializing a PRNG with the same value leads to the same sequence of random numbers. This is a helpful property for validating and debugging simulation algorithms. The internal state of a PRNG is finite and thus the sequence of random deviates is periodic and repeats at some point. The period of the PRNG should be several orders of magnitude greater than the expected number of necessary random deviates.

The most widely used algorithms for PRNGs are *linear congruential generators* (LCG). They are computed with

$$X_{n+1} = (a \cdot X_n + b) \bmod m \quad (3.23)$$

for constants a and b . An efficient implementation can be achieved by using modulus m as a power of 2. The constants a and b have to be chosen carefully to avoid problematic cases and achieve maximum period [Knu97]. LCGs are fast and require only small amounts of memory. That is why they are widely supported, e.g., `rand()` in C/C++ with $m = 2^{31}$ and JAVA's `java.util.Random` with $m = 2^{48}$. But they are not suitable for stochastic simulation algorithms, because of their rather short period and the serial correlation among other things. In summary, much research work prior to the 21st century that relied on random deviates computed with LCGs, is much less reliable than it might have been as a result of using poor-quality PRNGs [Pre+07, chap. 7].

Generators based on *linear recurrences* were a major advance. In 1997 Matsumoto and Nishimura developed the Mersenne Twister algorithm [MN98]. It became the most widely used general-purpose PRNG, not only in scientific

research. It avoids most of the problems of earlier generators. They published two versions of the Mersenne Twister, MT11213 and MT19937. We use MT19937, which is the most common used. It has a very long period of $2^{19937} - 1$ (approximately $4.3 \cdot 10^{6001}$), which is close to impossible to reach. It is proven to be equidistributed in up to 623 dimensions. Thereby it is surprisingly fast (faster than most LCGs) [Mar15], but needs about 2.5 KiB of memory to hold its internal state. This might be an issue for some systems having limited memory. The Mersenne Twister algorithm is available in many programming languages, e.g., C/C++, Java, FORTRAN, Lisp and so on. In 2006 Panneton, L’Ecuyer, and Matsumoto introduced the well-equidistributed long-period linear algorithms (WELL) [PLM06] that is based on linear recurrences modulo 2. They come in different period sizes for 2^n and $n = 512, 521, 607, 800, 1024, 19937, 21701, 23209$, and 44497. These algorithms produce higher-quality random deviates with better equidistribution than MT19937 and improve upon “bit-mixing” properties. It is supposed to be a replacement of the Mersenne Twister and we are investigating their use in our simulation algorithms.

3.10 Closing Remarks

In this chapter we presented the most widely used stochastic simulation algorithms, i.e., direct method, first reaction method, next reaction method and τ -leaping. We presented optimizations for the direct method by including the dependency graph, updating continuously the sum of transitions rates and adding the 2D search, so that there is only little distance to the next reaction method in terms of run-time performance. Thus it is a matter of taste, which one to use for simulating stochastic systems. Moreover, we introduced the discrete-time leap method, a new approximate stochastic simulation algorithm developed by Christian Rohr. We showed that it computes reasonable results on typical biochemical reactions. In Chapter 6, we will use δ -leaping on real biochemical models and compare it to the direct method. To put it directly, the results of δ -leaping are reasonable and it outperforms the direct method on large and dense networks.

In the literature, stochastic simulation algorithms typically process stochastic

models, which correspond to stochastic Petri nets. Christian Rohr’s work built on previous work by Sebastian Lehrack’s master thesis [Leh07] and adapted the standard algorithms as described in Section 3.3 and 3.5 to support full \mathcal{XSPN} , which specifically required to incorporate deterministically timed transitions (delayed, scheduled). In contrast, the approximative simulation algorithms of Section 3.6 and 3.7 support only standard \mathcal{SPN} ; the idea of jumps over time steps obviously contradicts the embedding of deterministically timed transitions between the occurrences of stochastic transitions.

In the discussion, but not in the implementation, we neglected one specific optimization technique for stochastic simulation that is the parallelisation of the simulation algorithms. It is straightforward to parallelise all algorithms in the same way, because each one of them computes a single path through the CTMC and this is a completely independent procedure. Thus, letting several instances of a stochastic simulation algorithm run in parallel and collect the results afterwards leads to a speed-up close to the number of instances. There is only one issue that needs to be taken care of, each instance has to initialise its pseudo random number generator with a different seed, otherwise they would generate the same trace. We implemented the parallelisation for two different scenarios. The first implementation uses multi-threading and is meant for shared memory machines. The second implementation using multi-processing is intended for distributed memory environments.

The following stochastic simulation algorithms were implemented by the thesis’ author in the advanced analysis tool MARCIE [SRH11]: optimized direct method with 2D search, next reaction method, τ -leaping and discrete-time leap method. All simulation algorithms are parallelised for multi-threading and multi-processing.

Chapter 4

Simulative Analysis

We presented some methods for the simulation of stochastic Petri nets in the previous chapter. Now, we want to analyse the output of these methods, i.e., the generated traces. In a first step, we have to observe the running simulation and save the marking of the net at desired time points. That is equivalent to an experimentalist taking notes of the state of the experiment at certain time points. As we are in a stochastic setting, we have to repeat the simulation and the observation several times to get reliable results. After that, we can apply several techniques to the observed data to get better insights in the stochastic behaviour of the simulated model.

In this chapter, we present some analysis methods for simulation of stochastic Petri nets, ranging from trace generation and distribution generation to observer computation and steady state simulation. Trace generation is the direct outcome of stochastic simulation and thus the most common analysis technique we present. Probability distributions are typically computed by numerical methods [Sch14] and not by stochastic simulation. We developed a method for the approximation of probability distributions by means of stochastic simulation and optimized this method for any time interval. Steady state simulation is an ongoing area of research, but most of it is related to the computation of single variate [Paw90; AG07]. We introduce a method for the approximation of the steady state distribution by use of stochastic simulation. Last but not least, we apply the above methods to the simulation of derived measures, i.e., observers (sometimes called rewards, or costs).

4.1 Trace Generation

The first simulative analysis we discuss is trace generation, which suggests itself. We have to apply statistics to our observed data, so we have to declare some terminology first. When simulating stochastic Petri nets, we are generating paths through the continuous-time Markov chain. This means, we are visiting only a subset of all states in general. In terms of statistics, the CTMCs of the stochastic Petri net at the observed time points represent the statistical populations. Thus sampling statistics have to be applied to the observed data. The observed number of tokens of a place p at a certain time point τ is a random variable $x(p)$. After having done N independent simulation runs, we get a vector $\underline{x}(p)$ of independently drawn observations. The sample mean of the number of tokens of place p can be computed by

$$\bar{x}(p) = \frac{1}{N} \sum_{i=1}^N x_i(p) . \quad (4.1)$$

This can be generalised to all places of the net in the random variable vector \underline{x} . Thus, the sample mean vector contains the average of the observations for each place, and is written

$$\bar{\underline{x}} = \frac{1}{N} \sum_{i=1}^N \underline{x}_i . \quad (4.2)$$

The sample mean $\bar{x}(p)$ is a good estimator of the population mean $\mu(p)$, because it can be computed efficiently and it is unbiased. The population mean is also referred to as expected value or expectation $E(X)$. It is computed over all possible values n of the discrete probability distribution by

$$E(X) = \sum_{i=1}^n x_i p_i , \quad (4.3)$$

where x_i is a value of the random variable X and p_i is its probability. In contrast to the population mean, the sample mean is a random variable, because its value is different each time we take new observations from stochastic simulation. It is distributed with the same distribution as the CTMC at the observed time point τ .

The weak law of large numbers states that the sample mean \bar{x} converges in

probability towards the population mean μ [Wil06]. That is to say that for any positive number ϵ ,

$$\lim_{n \rightarrow \infty} Pr(|\bar{x} - \mu| < \epsilon) = 1 . \quad (4.4)$$

The strong law of large numbers states that the sample mean \bar{x} converges almost surely to the population mean μ [Wil06]. That is,

$$Pr(\lim_{n \rightarrow \infty} \bar{x} = \mu) = 1 . \quad (4.5)$$

In the end, the sample mean $\bar{x}(p)$ gives us a good estimator of the average stochastic behaviour of a place p . Putting several sample means, computed at different time points, together results in a good overview on the behaviour of the observed places over time.

In addition to the mean, the variance of the observed data may be of interest, i.e., we want to know how far the values are spread around the mean. Thus we want to estimate the population variance σ^2 . As the population mean is unknown, we have to take special care of the variance, because the (uncorrected) sample variance is a biased estimator and underestimates the population variance by a factor of N^{-1}/N . For that reason, we apply Bessel's correction to get an unbiased sample variance s^2 by

$$s^2 = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2 . \quad (4.6)$$

The computation of the sample variance with Equation 4.6 would require to compute the sample mean first, while keeping all observation data, and compute the sample variance afterwards. This is inefficient both in terms of runtime and memory consumption. Thus, we apply the squared deviations from the mean and use the following short cut

$$\sum_{i=1}^N (x_i - \bar{x})^2 = \left[\sum_{i=1}^N x_i^2 \right] - \frac{1}{N} \left[\sum_{i=1}^N x_i \right]^2 \quad (4.7)$$

to compute the sample variance more efficiently by

$$s^2 = \frac{1}{N-1} \left(\left[\sum_{i=1}^N x_i^2 \right] - \frac{1}{N} \left[\sum_{i=1}^N x_i \right]^2 \right) . \quad (4.8)$$

Now, we can compute the sample variance using Equation (4.8) by just taking the observed data and in one pass.

The standard deviation can be used to quantify the amount of variation of a set of values, too. The advantage of having the standard deviation instead of the variance is that it is in the same unit as the observed data, whereas the unit of the variance is squared. The standard deviation σ is usually computed as the square root of the variance σ^2 . We compute the sample standard deviation s from the corrected sample variance in the same way

$$s = \sqrt{\frac{1}{N-1} \left(\left[\sum_{i=1}^N x_i^2 \right] - \frac{1}{N} \left[\sum_{i=1}^N x_i \right]^2 \right)}. \quad (4.9)$$

This yields to a biased estimator of the standard deviation σ , because the square root is a non-linear function. The reintroduced bias is small, but significant for values of $N \leq 10$ and drops of at $1/N$ with N increasing. In contrast to mean and variance, there is no formula for a unbiased sample standard deviation that works for all distributions. Therefore, we use Equation (4.9) and disregard the bias by increasing the number of simulation runs.

Example 9. We show the evolution of mean and standard deviation over time on Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$. Figure 4.1 shows the results after 10 000 simulation runs.

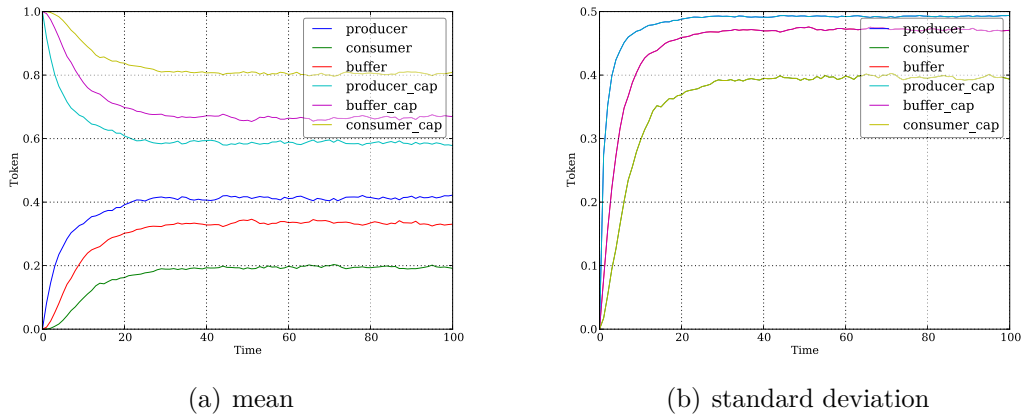


Figure 4.1: Mean and standard deviation over time of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.

Up to now, we discussed the observation of the number of tokens on places

at distinct time points. Another measure of interest is the activity of a transition in a certain time interval, e.g., the interval between the observations of the number of tokens on places. The activity of a transition is expressed by the number of firings. This number has to be accumulated, each time the transition fires. When the observation of the transition's activity takes place, the accumulated number of firings is preserved in the random variable $x(t)$ and the accumulated number of firings of the transition is reset to zero. As the observed activity of the transition is a random variable from independent observations, we can apply the sampling statistics to it presented before.

Example 10. We demonstrate the evolution of transition firings over time on Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$. Figure 4.2 shows the results after 10 000 simulation runs.

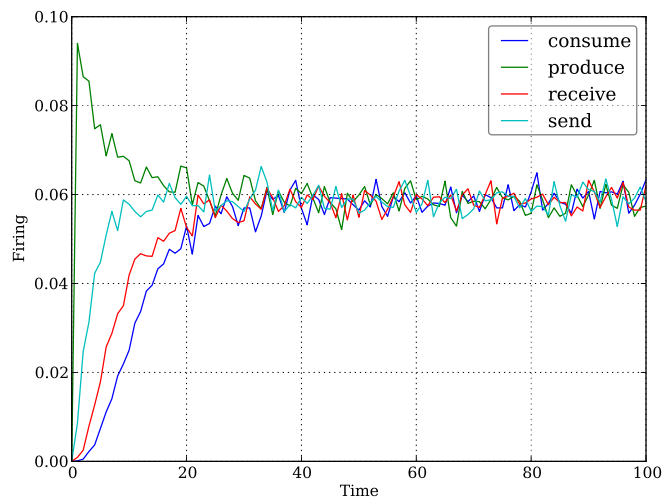


Figure 4.2: Averaged number of transition firings of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.

4.2 Transient Solutions

In the previous section, we discussed the analysis of the generated simulation traces by means of sampling statistics. So we can get an overview of the averaged stochastic behaviour of the model over time. Now, we want to compute transient solutions of the continuous-time Markov chain. There exist several

numeric approaches for this problem, e.g., uniformization [Sch14], matrix decomposition and Krylov subspace methods [Ste94]. These methods yield very good results, but rely on finite state spaces with at most 1×10^9 states [Sch14]. There are approximative numerical methods that are able to compute transient solutions of CTMCs with larger or infinite state spaces, e.g., fast adaptive uniformization [Did+09]. But the application of this method is still limited by the size of the constructed subgraph. We want to approximate the probability distribution of the CTMC at some time point τ using stochastic simulation.

Let $X(\tau)$ be the random variable describing the CTMC at time point τ . Then $\pi_i(\tau)$ is the probability that the system is in state i , i.e.,

$$\pi_i(\tau) = Pr(X(\tau) = i) . \quad (4.10)$$

The probability to be in state i after $\Delta\tau$ time units is

$$\pi_i(\tau + \Delta\tau) = \pi_i(\tau) \left(\sum_{k \in \mathcal{R}} q_{ki}(\tau) \pi_k(\tau) \right) \Delta\tau + o(\Delta\tau) \quad (4.11)$$

using little-o notation. Then

$$\lim_{\Delta\tau \rightarrow 0} \left(\frac{\pi_i(\tau + \Delta\tau) - \pi_i(\tau)}{\Delta\tau} \right) = \lim_{\Delta\tau \rightarrow 0} \left(\sum_{k \in \mathcal{R}} q_{ki}(\tau) \pi_k(\tau) + \frac{o(\Delta\tau)}{\Delta\tau} \right) , \quad (4.12)$$

i.e.,

$$\frac{d\pi_i(\tau)}{d\tau} = \sum_{k \in \mathcal{R}} q_{ki}(\tau) \pi_k(\tau) . \quad (4.13)$$

In matrix notation, this is

$$\frac{d\pi(\tau)}{d\tau} = \pi(\tau) \mathbf{Q}(\tau) . \quad (4.14)$$

The solution $\pi(\tau)$ is given by

$$\pi(\tau) = \pi(0) e^{\mathbf{Q}\tau} , \quad (4.15)$$

where $\pi(0)$ is the initial probability vector.

We can approximate $\pi(\tau)$ by applying Borel's law of large numbers. It states that for every experiment, which is repeated N times under identical condi-

tions, the occurrence of any specified event approximates the probability of the event's occurrence in general. The more repetitions take place, the better the approximation tends to be. Let any specified event E occur with probability pr , and $N(E)$ is the number of occurrence of event E throughout N experiments, then with probability one [Wen91]

$$\frac{N(E)}{N} \rightarrow pr \text{ when } N \rightarrow \infty , \quad (4.16)$$

i.e.,

$$Pr \left(\lim_{N \rightarrow \infty} \frac{N(E)}{N} = pr \right) = 1 . \quad (4.17)$$

In addition, this gives us the lower bound of the probability any event can have that occurs in an experiment. Let $N(E) = 1$, then the probability for event E is

$$pr = \frac{1}{N} . \quad (4.18)$$

Referring back to stochastic simulation, we are able to approximate the probability of any visited state i at time point τ by

$$\pi_i(\tau) \approx \sum_{n=1}^N \begin{cases} N^{-1} & X(\tau) = i \\ 0 & otherwise . \end{cases} \quad (4.19)$$

The computation of the probability distribution is straightforward from an algorithmic point of view. The current marking has to be saved after reaching time point τ and the probability of this marking has to be increased by $1/N$ in each simulation run. After N simulation runs, we have the approximate probability distribution at time point τ of at most N different states and with a lower bound on the probability of N^{-1} .

Example 11. We approximate the probability distribution at time point $\tau = 10$ of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs

$$\pi(100) \approx \{0.4363, 0.1532, 0.0592, 0.0263, 0.0084, 0.2633, 0.0522, 0.0011\}.$$

This is quite close to the probability distribution computed by the numerical

algorithm

$$\pi(100) = \{0.4310, 0.1593, 0.0649, 0.0248, 0.0078, 0.2611, 0.0499, 0.0012\}.$$

After being able to approximate the probability distribution at time τ , it seem likely to compute the probability distributions at several time points $\{\tau_1, \tau_2, \dots, \tau_j\}$. In a naive approach, we could use repeatedly the described method for each of the n time points. This would yield correct results, but would be very inefficient, because we would have to perform $n \cdot N$ simulation runs and we would repeat the simulation for each τ_{j+1} with $j > 1$ up to τ_j that was already done before.

But we can do better. We want to avoid the repeated simulations for the prior time intervals and want to keep the number of simulation runs close to N . Therefore, Equation (4.15) can be adapted for any time interval (τ_j, τ_{j+1}) with $j > 0$ and $\tau_0 = 0$ to

$$\pi(\tau_{j+1}) = \pi(\tau_j) e^{Q(\tau_{j+1} - \tau_j)}, \quad (4.20)$$

because of the memoryless property of a stochastic process, i.e., Markov property.

We adapt our approach for Equation (4.20). For the first time interval (τ_0, τ_1) , we start the stochastic simulation from the initial marking and the probability vector $\pi(0)$ consists of one entry for m_0 , which has probability one. So we can compute the probabilities of the visited states at τ_1 with Equation (4.19). In the next step, we start with probability vector $\pi(\tau_1)$, which almost surely has more than one entry. We perform simulations for each state in $\pi(\tau_1)$ up to time point τ_2 . The question is now, how many simulation runs do we perform for each state. We could do N simulation runs like in the first step. This would enable us to capture states with a probability of at least $(N \cdot |\pi(\tau_1)|)^{-1}$, but we would have to perform $N \cdot |\pi(\tau_1)|$ simulation runs. Thus, we decided to perform $\lceil N \cdot \pi_i(\tau_1) \rceil$ simulation runs for each state i , which results in slightly more simulation runs, but this is not significant for the overall run-time. The probability of the reached state at τ_2 is then increased by the quotient $\pi_i(\tau_1) / \lceil N \cdot \pi_i(\tau_1) \rceil$ of the probability of the starting state i and the number of simulation runs, calculated before. We can apply these steps to the subsequent time intervals and get the probability distributions without

any unnecessary simulations.

In general, the probability to be in state i at the end of the time interval (τ_j, τ_{j+1}) is approximately

$$\pi_i(\tau_{j+1}) \approx \sum_{k=1}^{|\pi(\tau_j)|} \sum_{n=1}^{\lceil N \cdot \pi_k(\tau_j) \rceil} \begin{cases} \pi_k(\tau_j) / \lceil N \cdot \pi_k(\tau_j) \rceil & X(\tau_{j+1}) = i \\ 0 & \text{otherwise} \end{cases} \quad (4.21)$$

Example 12. We compute the transient solutions up to $\tau = 100$ of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$. Figure 4.3 shows the results after 10 000 simulation runs.

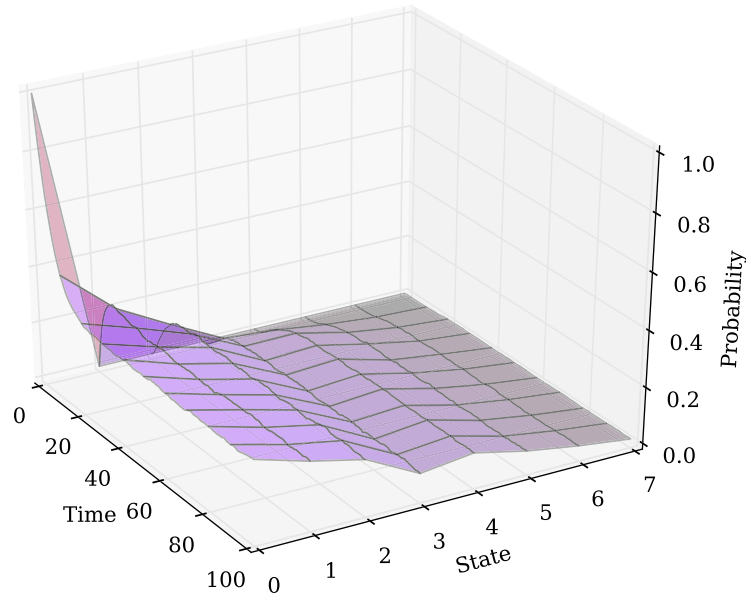


Figure 4.3: Transient solutions up to $\tau = 100$ of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.

4.3 Steady State Distribution

As we are able to compute the transient probability distributions at any time interval, now we are interested in the long run behaviour. A given continuous-time Markov chain may converge to a limiting stationary distribution, i.e., steady state probability distribution.

A distribution is called stationary, if $\forall i \in \mathcal{R}$ it satisfies

$$0 \leq \pi_i \leq 1, \sum_{i \in \mathcal{R}} \pi_i = 1, \pi_i = \sum_{k \in \mathcal{R}} \pi_k q_{ki}. \quad (4.22)$$

In matrix notation, this is

$$\pi \mathbf{Q} = \pi. \quad (4.23)$$

A distribution is called limiting, if there exists

$$\lim_{\tau \rightarrow \infty} \pi(\tau) \quad (4.24)$$

and we write

$$\pi = \lim_{\tau \rightarrow \infty} \pi(\tau) \quad (4.25)$$

as the limiting distribution. The stationary distribution must not necessarily be the limiting distribution of a Markov chain [Ste94]. Thus, the existence of a steady state probability distribution π for a given CTMC depends on the structure of the generator matrix \mathbf{Q} and can be computed with

$$\pi \mathbf{Q} = 0. \quad (4.26)$$

There is no assumption on the starting distribution $\pi(0)$ in Equation (4.26), i.e., the CTMC converges to the steady state distribution no matter where it begins.

In our simulative approach, we are not able to encounter the generator matrix \mathbf{Q} , but we can take advantage of the fact, that if there exists a steady state distribution, then the rate of change of $\pi(\tau)$ at steady state is zero, i.e.,

$$\frac{d\pi(\tau)}{d\tau} = 0. \quad (4.27)$$

The steady state distribution π is a property over the entire state space, but it is also an ergodic property [GS01]. That means we can show how the spatial property π_i of state i relates to the time average of the system. Let $N_i(\tau)$ be the times of being in state i during the time interval $(0, \tau)$. The time average is denoted by

$$\frac{N_i(\tau)}{\tau}. \quad (4.28)$$

We apply the ergodic theorem [Bir31] that assert the existence of a time average on each trajectory and show that, for any initial state $X(0) = k$ and for every state i , Equation (4.28) converges almost surely to π_i when τ goes to infinity, i.e.,

$$\lim_{\tau \rightarrow \infty} \frac{N_i(\tau)}{\tau} = \pi_i . \quad (4.29)$$

Let us consider first the case that i is a transient state. A state i is called transient if there exists a state k which is reachable from i , i.e., $i \rightarrow k$, but not in the opposite direction, i.e., $k \not\rightarrow i$. Otherwise, state i is called recurrent. That means, there is a probability greater than zero to reach state k starting from i , but it is not possible to return back to state i . Thus, after some finite time we will almost surely never reach state i , i.e.,

$$\lim_{\tau \rightarrow \infty} \frac{N_i(\tau)}{\tau} = 0 = \pi_i . \quad (4.30)$$

In the second case that state i is recurrent, let the sequence of successive visits $\tau_1, \tau_2, \tau_3, \dots$ of state i be independent and identically distributed. By the definition of $N_i(\tau)$ we have

$$\sum_{m=1}^{N_i(\tau)} \tau_m \leq \tau < \sum_{m=1}^{N_i(\tau)+1} \tau_m \quad (4.31)$$

from which we get

$$\frac{\sum_{m=1}^{N_i(\tau)} \tau_m}{N_i(\tau)} \leq \frac{\tau}{N_i(\tau)} < \frac{\sum_{m=1}^{N_i(\tau)+1} \tau_m}{N_i(\tau) + 1} . \quad (4.32)$$

Applying the strong law of large numbers we have that almost surely

$$\lim_{n \rightarrow \infty} \frac{\sum_{m=1}^n \tau_m}{n} = \lim_{n \rightarrow \infty} \frac{\sum_{m=2}^n \tau_m}{n} + \lim_{n \rightarrow \infty} \frac{\tau_1}{n} . \quad (4.33)$$

Combining Equation (4.32) with Equation (4.33) get us

$$\lim_{\tau \rightarrow \infty} \frac{\tau}{N_i(\tau)} = \frac{1}{\pi_i} , \quad (4.34)$$

from which we obtain Equation (4.28) almost surely.

Referring back to stochastic simulation, the last issue to consider is when to

stop the simulation run? We adapt Equation (4.27) and introduce an upper bound ϵ on the change over time, i.e.,

$$\frac{d\pi(\tau)}{d\tau} \leq \epsilon . \quad (4.35)$$

The simulation stops if for all visited states i the change in the probability is below or equal to ϵ . Thus, we are able to approximate the steady state probability of any visited state i with only 1 simulation run by

$$\pi_i \approx \frac{N_i(\tau_\epsilon)}{\tau_\epsilon} . \quad (4.36)$$

This works well if the reachability graph has only one terminal strongly connected component, i.e., the Markov chain has only one recurrence class. In case that there are multiple terminal strongly connected components, i.e., multiple recurrence classes R_1, R_2, \dots, R_r , each recurrence class has its own steady state distribution $\pi^i, 1 \leq i \leq r$. These distributions have $\pi_k^i = 0$ for all states $k \notin R_i$ and $0 < \pi_k^i \leq 1$ for all states $k \in R_i$. The family of all the steady state distributions can be obtained by taking all possible convex combinations of $\pi^i, 1 \leq i \leq r$. The convex combination X of several distributions Y_i is a weighted sum of its component distributions, this is called a finite mixture distribution. The probability density function of X is

$$p_X(x) = \sum_{i=1}^n w_i \cdot p_{Y_i}(x) \quad (4.37)$$

with $0 < w_i \leq 1$ and $\sum_{i=1}^n w_i = 1$. Intuitively speaking, the finite mixture distribution combines the steady state probabilities of each recurrence class (terminal strongly connected component) with the probability of reaching each one of them. In our simulative setting, we can achieve this by computing several simulation runs N and combine the steady state probabilities in the following way

$$\pi_i \approx \frac{1}{N} \sum_{n=1}^N \frac{N_i(\tau_\epsilon)}{\tau_\epsilon} . \quad (4.38)$$

Example 13. We approximate the steady state distribution of Example 6

with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 100 simulation runs

$$\pi(100) \approx \{0.0309, 0.1857, 0.0926, 0.2774, 0.0104, 0.1085, 0.0620, 0.2322\}.$$

This is quite close to the steady state distribution computed by the numerical algorithm

$$\pi(100) = \{0.0309, 0.1856, 0.0928, 0.2784, 0.0103, 0.1082, 0.0619, 0.2320\}.$$

4.4 Observers

Sometimes it may be valuable to observe derived measures. This can be achieved by enriching the CTMC with an observer, also called reward, cost, gain or bonus. This adds an extra dimension to the CTMC, while moving on in time, it accumulates an output. In order to realize this, a reward structure $(\underline{\rho}, \underline{\iota})$ is added to the CTMC. The state reward function $\underline{\rho}: \mathcal{R} \rightarrow \mathbb{R}_0^+$ defines the rate at which reward $\underline{\rho}(s)$ is obtained in a state s . That means a reward of $\tau \cdot \underline{\rho}(s)$ is earned, if the CTMC stays in state s for τ time units. Thus the accumulated state reward at time point τ in a path $\omega = ((s_0, \tau_0)(s_1, \tau_1), \dots)$ is defined by

$$Y_s(\tau) = \sum_{i=0}^{n-1} \tau_i \cdot \underline{\rho}(s_i) + \left(\tau - \sum_{i=0}^{n-1} \tau_i \right) \cdot \underline{\rho}(s_n), \quad (4.39)$$

if and only if $\sum_{i=0}^{n-1} \tau_i \leq \tau \wedge \sum_{i=0}^n \tau_i > \tau$. For every state s_i the state reward is the product of the state reward function $\underline{\rho}(s_i)$ and the sojourn time τ_i . But for state n the state reward function is multiplied by the residual time until time point τ .

The impulse reward function $\underline{\iota}: \mathcal{R} \times \mathcal{R} \rightarrow \mathbb{R}_0^+$ assigns to each transition t from state s to s' a reward $\underline{\iota}(s, s')$, i.e., a reward $\underline{\iota}(s, s')$ is acquired, if transition t fires. Thus the accumulated impulse reward at time point τ in a path $\omega = ((s_0, \tau_0)(s_1, \tau_1), \dots)$ is defined by

$$Y_i(\tau) = \sum_{i=0}^{n-1} \underline{\iota}(s_i, s_{i+1}). \quad (4.40)$$

if and only if $\sum_{i=0}^{n-1} \tau_i \leq \tau \wedge \sum_{i=0}^n \tau_i > \tau$.

The observer defines a stochastic process $Y(\tau), \tau \in \mathbb{R}_0^+$, which is the state and impulse rewards accumulated from time point 0 to τ , i.e.,

$$Y(\tau) = Y_s(\tau) + Y_i(\tau) . \quad (4.41)$$

Note that $Y(\tau)$ is non Markovian, because its value depends solely on the past. But $Y(\tau)$ is a function of the CTMC, because it is defined over the same probability space. If the CTMC has reached state n of the path ω at time point τ , the accumulated reward $Y(\tau)$ is the sum of the state reward accumulated in the preceding states and the impulse reward accumulated on the firing of each transition in path ω reaching state s_n .

A CTMC enriched with a state reward function $\underline{\rho}$ and an impulse reward function ι is called a *Markov reward model* (MRM).

In simulative analysis, we can evaluate directly the reward functions while the trace ω is generated. So we can efficiently apply the analysis techniques presented in this chapter to observer as well with just a small adaptation.

The state reward function and the impulse reward function are discrete weight functions. Generally speaking, a discrete weight function $w: A \rightarrow \mathbb{R}_0^+$ is a positive function defined on a discrete set A , which is finite or at least countable. A weight function $w(a) := 1$ refers to the unweighted case, where all elements have equal weights. Let $f: A \rightarrow \mathbb{R}$ be a real valued function, then the unweighted sum is

$$\sum_{a \in A} f(a) . \quad (4.42)$$

Adding the weight function w , the weighted sum is defined as

$$\sum_{a \in A} f(a)w(a) . \quad (4.43)$$

In addition, the unweighted mean of f

$$\frac{1}{|A|} \sum_{a \in A} f(a) , \quad (4.44)$$

can be replaced with the weighted mean

$$\frac{\sum_{a \in A} f(a)w(a)}{\sum_{a \in A} w(a)} . \quad (4.45)$$

Which can be simplified to

$$\sum_{a \in A} f(a)w'(a) , \quad (4.46)$$

after normalizing the weights, i.e., $\sum_{a \in A} w'(a) = 1$.

Referring back to simulative analysis, the weighted sample mean \bar{Y} of the observer, with normalized weights (weights sum to one) is itself a random variable, and its expected value and variance are related to the expected value and variance of the observations in the following way.

Let the observations have an expected value from Equation (4.3) then the weighted sample mean has expectation

$$E(\bar{Y}) = \sum_{i=1}^n x_i p_i y_i . \quad (4.47)$$

The weighted sample variance $s_{\bar{Y}}^2$ of the observer is then

$$s_{\bar{Y}}^2 = \sum_{i=1}^n s_i^2 y_i^2 . \quad (4.48)$$

It is easy to see that a state reward function $\rho := 1$ or an impulse reward function $\iota := 1$ would lead to the unweighted sample mean and unweighted sample variance presented in Section 4.1. By having Equation (4.47), it is straightforward to compute transient solutions and the steady state distribution of an observer using stochastic simulation.

Example 14. We demonstrate the computation of an observer for the \mathcal{SPN} in Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ up to $\tau = 100$. Figure 4.4 shows the averaged waiting time for *consumer* to receive a token after 10 000 simulation runs. We achieve this by evaluating a state reward function that earns a reward of 1 in each state where *consumer* has zero tokens.

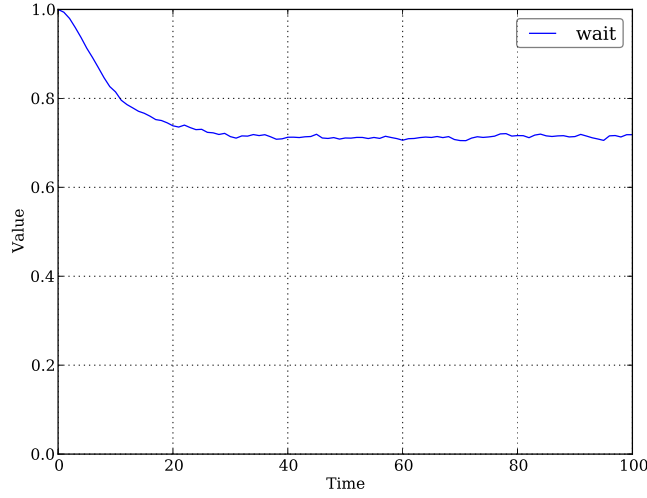


Figure 4.4: Averaged waiting time for *consumer* to receive a token up to $\tau = 100$ of Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$ after 10 000 simulation runs.

4.5 Closing Remarks

We presented various simulative analysis techniques in this chapter, ranging from trace generation to the computation of the transient solutions and the steady state distribution, and the evaluation of observers. These analysis techniques provide us with a good insight into the behaviour of the network under investigation.

Trace generation is the most common analysis technique and usually available in most of the simulation tools. Probability distributions are typically computed by numerical methods and not by stochastic simulation. We developed a method for the approximation of probability distributions by means of stochastic simulation and optimized this method for any time interval. Furthermore, we introduced a method for the approximation of the steady state distribution by use of stochastic simulation. Last but not least, we applied the above methods to the simulation of derived measures, i.e., observers. We implemented all analysis techniques presented above in the analysis tool MARCIE [HRS13].

In the next chapter, we present a more advanced analysis technique based on stochastic simulation, known as simulative model checking.

Chapter 5

Simulative Model Checking

Model checking is an advanced analysis technique to check whether a model of a system satisfies some properties or specifications. Therefore, temporal logics are used to specify the properties of interest. Networks modelled as \mathcal{PN} or \mathcal{XPN} can be checked with the Computational Tree Logic (CTL) [CE81] and the Linear-time Temporal Logic (LTL) [Pnu77]. An efficient approach to verify bounded \mathcal{PN} and \mathcal{XPN} using CTL and LTL was presented in [Tov08].

Their probabilistic extensions, Probabilistic CTL (PCTL) [HJ94], Continuous Stochastic Logic (CSL) [Azi+00], Continuous Stochastic Reward Logic (CSRL) [Hav+02] and Probabilistic LTL (PLTL) [Bai98], are used for stochastic models. For the specification of temporal formulas we use PLTLc [DG08b], because it reasons over paths through the state space of the model and stochastic simulation produces traces through the state space of the model. We extend PLTLc with time-unbounded temporal operators and incorporate the steady state operator [Roh13]. Nevertheless, we introduce a simulative model checking algorithm for time-unbounded CSL as well as the steady state operator with one restriction, the formula must not have nested operators. Furthermore, we present simulative reward computation, which enables us to verify CSRL formulas with the same restriction as for CSL. Thus, we introduce the first simulative model checking algorithm for CSRL, to the best of our knowledge.

5.1 Simulative PLTLc Model Checking

For the specification of temporal formulas we define the probabilistic extension of the Linear-time Temporal Logic (LTL) [Pnu77] with numerical constraints [FR07], which is called Probabilistic Linear-time Temporal Logic with numerical constraints (PLTLc) [DG08b]. The grammar of all PLTLc formulas is given in Definition 25.

Definition 25. Syntax of the Probabilistic Linear-time Temporal Logic with Constraints:

$$\begin{aligned}
\psi &:= \mathcal{P}_{\bowtie x} [\phi] \mid \mathcal{P}_{=?} [\phi] \\
\bowtie &\in \{<, \leq, \geq, >\}, \quad x \in [0, 1] \\
\phi &:= X^I \phi \mid F^I \phi \mid G^I \phi \mid \phi \mid U^I \phi \mid \neg \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \sigma \\
I &:= [x_1, x_2] = \{x \in \mathbb{R}_0^+ \mid x_1 \leq x \leq x_2\}, \text{ omit } I = [0, \infty) \\
\sigma &:= \neg \sigma \mid \sigma \wedge \sigma \mid \sigma \vee \sigma \mid \text{value} \trianglelefteq \text{value} \mid \text{true} \mid \text{false} \\
\trianglelefteq &\in \{<, \leq, \geq, >, =, \neq\} \\
\text{value} &:= \text{value} \sim \text{value} \mid \text{Place} \mid \$\text{Variable} \mid \text{Int} \mid \text{Real} \mid \text{function} \\
\sim &\in \{+, -, *, /\}
\end{aligned}
\tag*{«}$$

A trace ω fulfils a linear-time temporal logic formula ϕ if the following \models relations hold.

$$\begin{aligned}
\omega &\models X^I \phi \iff \omega^{(1)} \models \phi \text{ and } \tau_0 \in I \\
\omega &\models F^I \phi \iff \exists \tau \in I : \omega @ \tau \models \phi \\
\omega &\models G^I \phi \iff \forall \tau \in I : \omega @ \tau \models \phi \\
\omega &\models \phi_1 U^I \phi_2 \iff \exists \tau \in I : \omega @ \tau \models \phi_2 \text{ and } \forall \tau' < \tau : \omega^{(\tau')} \models \phi_1 \\
\omega &\models \neg \phi \iff \omega \not\models \phi \\
\omega &\models \phi_1 \wedge \phi_2 \iff \omega \models \phi_1 \wedge \omega \models \phi_2 \\
\omega &\models \phi_1 \vee \phi_2 \iff \omega \models \phi_1 \vee \omega \models \phi_2 \\
\omega &\models v_1 \trianglelefteq v_2 \iff \text{evalState}(v_1, \omega^{(i)}) \trianglelefteq \text{evalState}(v_2, \omega^{(i)}) \\
\omega &\models \mathcal{P}_{\bowtie x} [\phi] \iff \text{Pr}(\omega \in \text{Path}(s) \mid \omega \models \phi) \bowtie x
\end{aligned}$$

The probability operator \mathcal{P} has two different modes. If it is used with the question mark as $\mathcal{P}_{=?}[\phi]$, then it will return the expected probability $Pr(\phi)$ that ϕ is true. We apply sample statistics as presented in Section 4.1, because $Pr(\phi)$ is the expected value of a random variable. We approximate the probability by computing the sample mean of the outcomes of N evaluations with

$$Pr(\phi) \approx \frac{1}{N} \sum_{n=1}^N \begin{cases} 1 & \text{if } \phi \text{ is true} \\ 0 & \text{otherwise.} \end{cases} \quad (5.1)$$

In simulative model checking, we compute a confidence interval (*c.i.*), rather than a single value. For simplicity, we assume the *c.i.* to have a lower and an upper bound $B_l, B_u \in \mathbb{R}_{\geq 0}$, such that the probability $Pr(\phi)$, which is not known in our case, is $B_l \leq Pr(\phi) \leq B_u$. A common method for the computation of a confidence interval for the probability of success is the Wald interval [Wal92]. It is used in hypothesis testing as well [You+06], but it was shown in [BCD02] that the Wald interval has unstable coverage characteristics even for large N . Thus, we decided to incorporate the Wilson score confidence interval [Wil22], which had shown good coverage characteristics even for small N and extreme probabilities. It is computed from the expected probability Pr , the number of simulation runs N and the $Z = 1 - \alpha/2$ percentile of the normal distribution with the confidence level α by

$$[B_l, B_u] = \frac{1}{1 + \frac{Z^2}{N}} \left(Pr + \frac{Z^2}{2N} \pm Z \sqrt{\frac{Pr(1 - Pr)}{N} + \frac{Z^2}{4N^2}} \right). \quad (5.2)$$

In the second case, $\mathcal{P}_{\bowtie x}[\phi]$ returns *true*, if $Pr(\phi) \bowtie x$ is fulfilled, *false* otherwise. We have to introduce an additional return value *unknown*, because of the indifference region defined by the confidence interval. It is returned, if the computed confidence interval covers the probability x , because in that case we can not decide whether ϕ is true or not. The return value of $\mathcal{P}_{\bowtie x}[\phi]$ is defined

as

$$\mathcal{P}_{\bowtie x}[\phi] = \begin{cases} true & \text{if } x \bowtie [B_l, B_u] \wedge x \notin [B_l, B_u] \\ false & \text{if } x \not\bowtie [B_l, B_u] \wedge x \notin [B_l, B_u] \\ unknown & \text{if } x \in [B_l, B_u] . \end{cases}$$

The operators \neg , \wedge , \vee are the standard boolean operators *not*, *and*, *or*. Whereas X, F, G, U denote the temporal operators *NEXT*, *FINALLY*, *GLOBALLY* and *UNTIL*. The *NEXT* operator ($X^I \phi$) refers to *true* in the next state and within the time interval I . The *UNTIL* operator ($\phi_1 U^I \phi_2$) indicates that a state where ϕ_2 holds is eventually reached within the time interval I , while ϕ_1 continuously holds. The *FINALLY* operator ($F^I \phi$) means that at some point within the time interval I a state where ϕ holds is eventually reached. Whereas the *GLOBALLY* operator ($G^I \phi$) refers to the condition ϕ continuously holding true within the time interval I . The latter two are syntactic sugar, as they rely on the following equivalences [BK08]:

$$F \phi \equiv true \ U \ \phi \tag{5.3}$$

$$G \phi \equiv \neg F \neg \phi . \tag{5.4}$$

State formulas are denoted by σ . They are evaluated for the given marking and return *true* or *false*. The function $evalState(v, \omega^{(i)})$ assigns a numerical value to the expression v by looking up the tokens that each place $x \in P(v)$ has in state $\omega^{(i)}$ of trace ω .

In the next sections we present algorithms to compute time-bounded and time-unbounded formulas, and afterwards an algorithm to compute steady state formulas in a simulatively manner.

5.1.1 Time-bounded Formula

Formulas are marked out as time-bounded, if all their temporal operators are decorated with an interval $I = (n, m)$ with $0 \leq n \leq m < \infty$. Hence it requires only a finite simulation trace to verify any time-bounded formula. Algorithms for simulative model checking of time-bounded formulas are well known, see [FR07; Bal+09]. A simulation trace, whether computed beforehand or on-the-fly, approximate or exact, constitutes a linear Kripke structure in

which PLTLc formulas can be verified [FR07].

The procedure to evaluate time-bounded formulas is given in Algorithm 10. We omit the evaluation of F^I and G^I , because they can be substituted using Equations (5.3) and (5.4). Algorithm 10 is a forward model checking method starting from the initial marking and adding states on-the-fly to the trace, if necessary. This has the advantage of being able to stop the trace generation as soon as the property is satisfied or falsified. On the other hand backward model checking algorithms, relying on precomputed traces, can check G by just looking at the last state of the trace [DG08a].

The verification of a formula ϕ is performed by the function *evalFormula*. It takes as arguments a formula ϕ , the current position *pos* in the trace and the generated trace ω . The return value of the function is a tuple of the actual result (*true/false*) and the position in the trace, where the function finished. The trace is timed, i.e., an element of the trace consists of the marking and the entry time. The trace is initialised with the initial marking and the starting time τ_0 . Each time the algorithm reaches the end of ω the next state is generated and is appended to the trace.

State formulas σ are evaluated by the function *evalState*, taking the formula σ and the marking of the current position in the trace as parameters. In case of boolean expressions, *evalFormula* is called on the sub expression(s) $\phi_1(\phi_2)$ and the current position in the trace. The result(s) of the sub expression(s) are then assessed with their boolean operation. Special care has to be taken of the returned positions pos_1 and pos_2 for the binary operators \wedge and \vee . For the \wedge -operator we take the minimum of the two positions, because we know that up to $\min(pos_1, pos_2)$ both sub expressions ϕ_1 and ϕ_2 must be *true* or it is the first position, where at least one sub expression is *false*. For the \vee -operator we take the maximum of the two positions, because we ensure that up to $\max(pos_1, pos_2)$ at least on sub expression ϕ_1 or ϕ_2 must be *true* or it is the first position, where both sub expressions are *false*.

The temporal operator $X^I \phi_1$ is evaluated in the time interval I . Therefore, we move to the next element in the trace and check if the time τ_{pos} is in the interval I specified in X^I . If that is the case, the result of the evaluation of the sub expression ϕ_1 is returned. Verifying time-bounded until formulas $\phi_1 U^I \phi_2$ is done by creating a trace ω and checking if $\omega \models \phi_1 U^I \phi_2$. Therefore we

extend ω until a state is reached where $\omega^{(pos)} \not\models \phi_1$, so that trace ω does not fulfil our formula, or $\omega^{(pos)} \models \phi_2$, that means trace ω satisfies our formula.

This approach works fine for time-bounded formulas, because it is guaranteed to terminate with a probability of 1. It either terminates on a positive or negative observation of our formula or at the end of the greatest time interval associated with the temporal operators.

It is crucial for simulative PLTLc model checking to hold the complete trace ω for two reasons. First for the sake of efficiency. Second to avoid branching that may occur, due to roll-back to a previous state, after the evaluation of sub formulas.

Example 15. We verify the following PLTLc formula on the \mathcal{SPN} in Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$. We want to know the probabilities of the number of tokens on place *buffer* at time point $\tau = 10$. This can be computed by using a free variable in the following formula

$$\mathcal{P}_{=?} [true \text{ U}^{10,10} buffer = \$x] .$$

The simulative model checking computes a confidence interval of $[1, 1]$ after 10 000 simulation runs, which is to be expected, because the free variable x is set to a value so that the formula becomes *true*. The probabilities for the number of tokens on x are $\{(0, 0.777), (1, 0.223)\}$.

5.1.2 Time-unbounded Formula

The model checking algorithm presented in the previous section works well for time-bounded formulas, but it is not applicable in the case of time-unbounded until formulas, because there is no finite time bound, so the algorithm does not eventually terminate in some cases.

The stochastic Petri net in Figure 5.1 demonstrates the problem of not terminating while verifying time-unbounded until formulas [Roh13].

Consider, for example, the formula

$$\mathcal{P}_{=?} [true \text{ U } p3 = 1] .$$

The corresponding CTMC of the \mathcal{SPN} in Figure 5.1 has only one state that

Algorithm 10 Evaluate time-bounded formula

Require: $\omega \leftarrow (m_0, \tau_0), \tau_{max}$

```

1: procedure EVALFORMULA( $\phi, pos, \omega$ )
2:   repeat
3:     switch  $\phi$ 
4:       case  $\sigma$  :
5:         return ( $pos, \text{EVALATOMIC}(\sigma, \omega^{(pos)})$ )
6:       case  $\neg\phi_1$  :
7:         ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \omega$ )
8:         return ( $pos_1, \neg res_1$ )
9:       case  $\phi_1 \wedge \phi_2$  :
10:        ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \omega$ )
11:        ( $pos_2, res_2$ )  $\leftarrow$  EVALFORMULA( $\phi_2, pos, \omega$ )
12:        return ( $\min(pos_1, pos_2), res_1 \wedge res_2$ )
13:       case  $\phi_1 \vee \phi_2$  :
14:        ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \omega$ )
15:        ( $pos_2, res_2$ )  $\leftarrow$  EVALFORMULA( $\phi_2, pos, \omega$ )
16:        return ( $\max(pos_1, pos_2), res_1 \vee res_2$ )
17:       case  $X^I\phi_1$  :
18:         if  $pos = |\omega|$  then
19:            $\omega \leftarrow \omega + \text{NEXTSTATE}(\omega^{(pos)})$ 
20:         end if
21:          $pos \leftarrow pos + 1$ 
22:         if  $\tau_{pos} \in I$  then
23:           return EVALFORMULA( $\phi_1, pos, \omega$ )
24:         end if
25:       case  $\phi_1 U^I \phi_2$  :
26:         if  $\tau_{pos} \in I$  then
27:           ( $pos_2, res_2$ )  $\leftarrow$  EVALFORMULA( $\phi_2, pos, \omega$ )
28:           if  $res_2 = \text{true}$  then
29:             return ( $\text{true}, res_2$ )
30:           end if
31:         end if
32:         ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \omega$ )
33:         if  $res_1 = \text{false}$  then
34:           return ( $\text{false}, res_1$ )
35:         end if
36:          $pos \leftarrow pos_1$ 
37:     end switch
38:     if  $pos = |\omega|$  then
39:        $\omega \leftarrow \omega + \text{NEXTSTATE}(\omega^{(pos)})$ 
40:     end if
41:      $pos \leftarrow pos + 1$ 
42:   until  $\tau_{pos} > \tau_{max}$ 
43:   return ( $pos, \text{false}$ )
44: end procedure

```

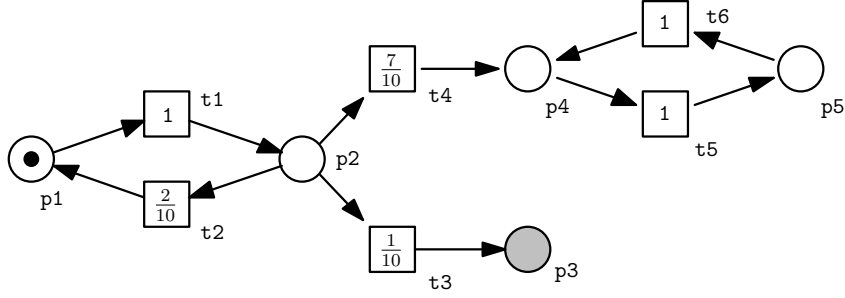


Figure 5.1: Stochastic Petri net demonstrating the issue of not terminating verification of time-unbounded until formulas.

satisfies the formula. But any trace ω starting in s_0 (Figure 5.1), that does not fulfil the formula is of infinite length. For this net, the probability of reaching the satisfying state starting from state s_0 and time 0 can be computed as follows:

$$Pr(true \text{ U } p3 = 1) = \frac{1}{10} \cdot \sum_{n=0}^{\infty} \left(\frac{2}{10}\right)^n = \frac{1}{8}.$$

Using the algorithm for time-bounded formulas does not work, because it will not terminate with a probability of $7/8$. We need another stopping criteria to solve this problem.

In Algorithm 11 we present the algorithm for checking time-unbounded until formulas. It is nearly the same as for time-bounded formulas except that one has to stop the simulation trace at some time point not known in advance. The decision of doing that is the crucial part of the algorithm. We assume that reaching the steady state is a reasonable stopping criteria (line 36). A system in a steady state has numerous properties that are unchanging in time. This implies that for any property p of the system, the partial derivative with respect to time is zero:

$$\frac{\partial p}{\partial t} = 0$$

If a system is in steady state, then the recently observed behaviour of the system will continue into the future. In stochastic systems, the probabilities that various states will be repeated will remain constant. But, if the system does not have a steady state, e.g., it oscillates, the algorithm will run forever or until the user stops the program. In the following section, we present the approach for evaluating whether the current simulation run reached already a steady state or not.

Algorithm 11 Evaluate time-unbounded formula

Require: $\text{trace} \leftarrow (m_0, \tau_0)$

```

1: procedure EVALFORMULA( $\phi, pos, \text{trace}$ )
2:    $\text{steadyStateReached} \leftarrow \text{false}$ 
3:   repeat
4:     switch  $\phi$ 
5:       case  $\sigma$  :
6:         return ( $pos, \text{EVALATOMIC}(\sigma, \text{trace}^{(pos)})$ )
7:       case  $\neg\phi_1$  :
8:         ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \text{trace}$ )
9:         return ( $pos_1, \neg res_1$ )
10:      case  $\phi_1 \wedge \phi_2$  :
11:        ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \text{trace}$ )
12:        ( $pos_2, res_2$ )  $\leftarrow$  EVALFORMULA( $\phi_2, pos, \text{trace}$ )
13:        return ( $\min(pos_1, pos_2), res_1 \wedge res_2$ )
14:      case  $\phi_1 \vee \phi_2$  :
15:        ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \text{trace}$ )
16:        ( $pos_2, res_2$ )  $\leftarrow$  EVALFORMULA( $\phi_2, pos, \text{trace}$ )
17:        return ( $\max(pos_1, pos_2), res_1 \vee res_2$ )
18:      case  $X\phi_1$  :
19:        if  $pos = |\text{trace}|$  then
20:           $\text{trace} \leftarrow \text{trace} + \text{NEXTSTATE}(\text{trace}^{(pos)})$ 
21:        end if
22:         $pos \leftarrow pos + 1$ 
23:        ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \text{trace}$ )
24:        return ( $pos_1, res_1$ )
25:      case  $\phi_1 U \phi_2$  :
26:        ( $pos_2, res_2$ )  $\leftarrow$  EVALFORMULA( $\phi_2, pos, \text{trace}$ )
27:        if  $res_2 = \text{true}$  then
28:          return ( $pos_2, res_2$ )
29:        end if
30:        ( $pos_1, res_1$ )  $\leftarrow$  EVALFORMULA( $\phi_1, pos, \text{trace}$ )
31:        if  $res_1 = \text{false}$  then
32:          return ( $pos_1, res_1$ )
33:        end if
34:         $pos \leftarrow pos_1$ 
35:    end switch
36:     $\text{steadyStateReached} \leftarrow \text{CHECKSTEADYSTATE}(\text{trace})$ 
37:    if  $pos = |\text{trace}|$  then
38:       $\text{trace} \leftarrow \text{trace} + \text{NEXTSTATE}(\text{trace}^{(pos)})$ 
39:    end if
40:     $pos \leftarrow pos + 1$ 
41:  until  $\text{steadyStateReached} = \text{true}$ 
42:  return ( $pos, \text{false}$ )
43: end procedure

```

Example 16. We adapt Example 15 and remove the time interval.

$$\mathcal{P}_{=?} [true \text{ U } buffer = \$x] \text{ .}$$

The simulative model checking computes a confidence interval of $[1, 1]$ after 10 000 simulation runs, which is to be expected, because the free variable x is set to a value so that the formula becomes *true*. In the time-unbounded setting the two possible number of tokens on place *buffer* are reached eventually. Thus the probabilities are $\{(0, 1.0), (1, 1.0)\}$.

5.1.3 Steady State Operator

In steady state simulation, the measures of interest are defined as limits, as the length of the simulation goes to infinity. There is no natural event to terminate the simulation, so the length of the simulation is made large enough to get “good” estimates of the quantities of interest. Steady state simulation generally poses two problems:

1. The existence of a transient phase may cause the estimate to be biased.
2. The simulation runs are long, and usually one cannot afford to carry out many independent simulations.

There are several methods that allow to cope with these problems to some extent. Among them are: the batch means method, the method of independent replicas, and the regeneration method. Each of these methods have its advantages and disadvantages. Additionally, we presented an approach to approximate the steady state distribution in Section 4.3, but this approach is not suitable best in the case of linear-time temporal logic, because we have to have the complete trace instead of the distribution. That is why we use in our implementation a sample batch means algorithm to compute the steady state. We choose Skart [Taf+08], which is an automated sequential procedure for on-the-fly steady state simulation output analysis, because it is specifically designed to handle observation-based statistics and usually requires a smaller initial sample size compared with other well-known simulation analysis procedures [TW10]. This algorithm partitions a long simulation run into batches,

computes an average statistics for each batch and constructs an interval estimate using the batch means. Based on this interval estimate Skart decides whether a steady state is reached or more samples were needed. A detailed description of the algorithm is given in [Taf+08].

We extend PLTLc with the steady state operator \mathcal{S} in [Roh12] and the syntax is defined in Definition 26. The return values are the same as for the probability operator \mathcal{P} . But inside of \mathcal{S} only state formulas are allowed, i.e., no temporal operators.

Definition 26. Extension of PLTLc with steady state operator \mathcal{S} .

$$\begin{aligned} \psi &:= \mathcal{P}_{\bowtie x} [\phi] \mid \mathcal{P}_{=?} [\phi] \mid \mathcal{S}_{\bowtie x} [\sigma] \mid \mathcal{S}_{=?} [\sigma] \\ \bowtie &\in \{<, \leq, \geq, >\}, \quad x \in [0, 1] \end{aligned} \quad \ll$$

Steady-state formulas are computed with Algorithm 12. At first the simulation trace is created until the steady state is reached (line 4 – 8). To get an unbiased result, we cut off the first n states, which bias the steady state (line 9). The remaining states are now checked whether the steady state property holds or not and the occupation time τ_o of the fulfilling states and the simulation time τ_s are summed up (line 11 – 19). The steady state probability is now the ratio τ_o/τ_s (line 19). But this gives correct results only for those Petri nets, where the reachability graph consists of only one strongly connected component (SCC). The complexity of this decision is the same as for constructing the reachability graph. In symbolic model checking the strongly connected components and the probabilities of reaching them are computed first. After that the probabilities within each SCC are computed and these are weighted with the probability of reaching the SCC. In that way the correct steady state probability is calculated. To solve this problem in simulative model checking, one has to make several simulation runs (steady state computations) and average the results. In that way the individual steady state estimates are weighted according to the strongly connected components.

Example 17. We verify the following PLTLc formula on the \mathcal{SPN} in Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$. We want to know the probabilities in the steady state for the number of tokens on place *buffer*. This can be

Algorithm 12 Steady state computation for one simulation run

Require: $\omega \leftarrow (m_0, \tau_0)$

```

1: procedure EVALSTEADYSTATE( $\sigma$ )
2:   steadyStateReached  $\leftarrow$  false
3:   pos  $\leftarrow$  0
4:   repeat
5:      $\omega \leftarrow \omega + \text{NEXTSTATE}(\omega^{(pos)})$ 
6:     pos  $\leftarrow$  pos + 1
7:     steadyStateReached  $\leftarrow$  CHECKSTEADYSTATE( $\omega$ )
8:   until steadyStateReached = true
9:   cutOff  $\leftarrow$  GETSTEADYSTATECUTOFF
10:   $\tau_o \leftarrow 0, \tau_s \leftarrow 0$ 
11:  for  $i \leftarrow \text{cutOff}, |\omega|$  do
12:     $(s_i, \tau_i) \leftarrow \omega^{(i)}$   $\triangleright$  state  $s_i$ , sojourn time  $\tau_i$  in  $s_i$ 
13:     $\tau_s \leftarrow \tau_s + \tau_i$ 
14:    res  $\leftarrow$  EVALSTATE( $\sigma, s_i$ )
15:    if res = true then
16:       $\tau_o \leftarrow \tau_o + \tau_i$ 
17:    end if
18:  end for
19:  return  $\tau_o / \tau_s$ 
20: end procedure

```

computed by using a free variable in the following formula

$$\mathcal{S}_{=?} [buffer = \$x] .$$

The simulative model checking computes a confidence interval of $[1, 1]$ after 1000 simulation runs, which is to be expected, because the free variable x is set to a value so that the formula becomes *true*. The probabilities for the number of tokens on place *buffer* in the steady state are $\{(0, 0.665), (1, 0.335)\}$.

5.2 Simulative CSL Model Checking

In Section 5.1, we presented simulative PLTLc model checking for verifying properties on stochastic Petri nets. Now, we discuss an simulative approach for model checking CSL. The Continuous Stochastic Logic (CSL) introduced in [Azi+00] and extended in [Bai+00a] is a stochastic adaptation of the Computation Tree Logic (CTL) [CGP01] to formulate properties of

CTMCs. We adapted the definition of CSL in [Bai+00a] by removing nested operator $\mathcal{P}_{\bowtie x}[\cdot]$. We discuss this issue in Section 5.2.1.

The syntax of our adapted CSL is given in Definition 27.

Definition 27. Syntax of the Continuous Stochastic Logic:

$$\begin{aligned}
\psi &:= \mathcal{P}_{\bowtie x}[\phi] \mid \mathcal{P}_{=?}[\phi] \mid \mathcal{S}_{\bowtie x}[\sigma] \mid \mathcal{S}_{=?}[\sigma] \\
\bowtie &\in \{<, \leq, \geq, >\}, \quad x \in [0, 1] \\
\phi &:= X^I \sigma \mid F^I \sigma \mid G^I \sigma \mid \sigma U^I \sigma \mid \sigma \\
I &:= [x_1, x_2] = \{x \in \mathbb{R}_0^+ \mid x_1 \leq x \leq x_2\}, \text{ omit } I = [0, \infty) \\
\sigma &:= \neg \sigma \mid \sigma \wedge \sigma \mid \sigma \vee \sigma \mid a \mid \text{true} \mid \text{false} \\
a &:= \text{value} \trianglelefteq \text{value} \\
\trianglelefteq &\in \{<, \leq, \geq, >, =, \neq\} \\
\text{value} &:= \text{value} \sim \text{value} \mid \text{Place} \mid \text{Int} \mid \text{Real} \mid \text{function} \\
\sim &\in \{+, -, *, /\}
\end{aligned}$$

«

The probability operator \mathcal{P} has the same characteristic as in PLTLc. It either returns the probability $Pr(\phi)$ that ϕ is true. Or it returns *true*, *false* or *unknown*, if $Pr(\phi) \bowtie x$ is satisfied, falsified or not decidable, because x falls into the computed confidence interval. We use again the Wilson score confidence interval, described in Section 5.1. The probability operator \mathcal{P} reasons over path formulas ϕ that contain temporal operators $\{X, F, G, U\}$. We apply again the equivalences in Equation (5.3) and Equation (5.4), and do not discuss F and G. In contrast to [Bai+00a], our temporal operators assert only state formulas and no probability operators. The operator \mathcal{S} asserts the steady state probability for a state formula. A state formula is denoted by σ . It evaluates atomic propositions a with boolean operators $\{\neg, \wedge, \vee\}$ in the given state. Atomic propositions compare *values* derived from the number of tokens on the places in the given state by comparison operators $\{<, \leq, \geq, >, =, \neq\}$. These *values* can be combined by arithmetic operators $\{+, -, *, /\}$.

The subset of CSL formulas, we defined, can be expressed equivalently by a subset of PLTLc. So the question may arise, what are the reasons for having it. The first answer is for the sake of efficiency. PLTLc verifies over paths and it is necessary to keep track of the path in general. Our adapted CSL evaluates

paths too, but it is not necessary to keep track of the whole path. We can verify CSL formulas by evaluating the current state of the path and can disregard the already passed states. This reduces the memory requirements for model checking time-bounded CSL formulas to $\mathcal{O}(1)$ instead of $\mathcal{O}(|\omega|)$ for PLTLc formulas. For model checking time-unbounded CSL formulas, the memory requirement is $\mathcal{O}(S)$, where S is the number of distinct states visited, which is typically much smaller than the length of the generated path. The second answer is, because it is a by-product of the continuous stochastic reward logic (CSRL) presented in Section 5.4.

A CSL state formula is satisfied in state s , if the following \models relations hold.

$$\begin{aligned}
s \models a &\iff s \models \text{evalAtomic}(a, s) \\
s \models \neg \sigma &\iff s \not\models \sigma \\
s \models \sigma_1 \wedge \sigma_2 &\iff s \models \sigma_1 \wedge s \models \sigma_2 \\
s \models \sigma_1 \vee \sigma_2 &\iff s \models \sigma_1 \vee s \models \sigma_2 \\
s \models \mathcal{P}_{\bowtie x}[\phi] &\iff \Pr(\omega \in \text{Path}(s) \mid \omega \models \phi) \bowtie x \\
s \models \mathcal{S}_{\bowtie x}[\sigma] &\iff \sum_{s' \models \sigma} \pi_s(s') \bowtie x
\end{aligned}$$

The function $\text{evalAtomic}(a, s)$ evaluates the atomic proposition a by looking up the tokens that each place $x \in P(a)$ has in state s .

A path formula is satisfied for any path $\omega \in \text{Path}(s)$, if the following \models relations hold.

$$\begin{aligned}
\omega \models X^I \sigma &\iff \omega^{(1)} \models \sigma \text{ and } \tau_0 \in I \\
\omega \models F^I \sigma &\iff \exists \tau \in I : \omega @ \tau \models \sigma \\
\omega \models G^I \sigma &\iff \forall \tau \in I : \omega @ \tau \models \sigma \\
\omega \models \sigma_1 U^I \sigma_2 &\iff \exists \tau \in I : \omega @ \tau \models \sigma_2 \text{ and } \forall \tau' < \tau : \omega^{(\tau')} \models \sigma_1
\end{aligned}$$

We discuss nesting of probabilistic operators in the context of simulative model checking and describe the algorithms used to verify CSL formulas in the following sections.

5.2.1 Nested Probabilistic Operator

The continuous stochastic logic is a branching-time logic like the computation tree logic. In the setting of simulative model checking it is possible to verify efficiently properties of states and paths, because the simulator generates paths through the state space of the model starting from an initial state. It is much more expensive in terms of run-time and memory consumption to verify properties of paths starting from a sets of initial states, as it happens to be in the case of nested probabilistic operators. This branching is theoretically possible in simulation context, see [YS02], but it becomes infeasible in terms of run-time. Let the expected number of simulation runs (samples) for each probabilistic operator be n_i with $0 \leq i < N$, N is the number of nested operators, $i = 0$ represents the outer probabilistic operator and $i > 0$ any nested probabilistic operator. The expected total number of simulation runs n is

$$n = \prod_{i=0}^N n_i , \quad (5.5)$$

which grows rapidly with the number of nested probabilistic operators. That is why we leave it to future work to find a simulative method with feasible run-time behaviour. Nevertheless, the statistical model checker Ymer supports nested probabilistic operator, but due to the already mentioned run-time issue, they suggest to use numerical methods for sub formulas and to use simulation only for the top formula [You+06].

5.2.2 Time-bounded Formula

A CSL formula is meant to be time-bounded if the time interval associated with any temporal logic operator is (n, m) with $0 \leq n \leq m < \infty$. Thus, the formula can be verified within finite time. The algorithm for evaluating time-bounded formulas is a forward model checking algorithm. This way it is able to stop the trace generation as soon as a decision was made. It is divided in two procedures, namely Algorithm 13 for the evaluation of path formulas and Algorithm 14 for the evaluation of state formulas.

Algorithm 13 starts from the initial state and evaluates the path formula on the current state. In case of operator $X^I \sigma$, the successor state s' of the current state s is generated, if the transition from state s to s' takes place in the time

Algorithm 13 Evaluate time-bounded path formula

```

1: procedure EVALPATHFORMULA( $\phi, m_0, \tau_0, \tau_{max}$ )
2:   ( $s, [\tau_s, \tau_{s+1})$ )  $\leftarrow$  INITIALSTATE( $m_0, \tau_0$ )
3:   repeat
4:     switch  $\phi$ 
5:       case  $X^I \sigma_1$  :
6:         if  $\tau_{s+1} \notin I$  then
7:           return false
8:         end if
9:         ( $s, [\tau_s, \tau_{s+1})$ )  $\leftarrow$  NEXTSTATE( $s, \tau_{s+1}$ )
10:        return EVALSTATEFORMULA( $\sigma_1, s$ )
11:       case  $\sigma_1 U^I \sigma_2$  :
12:        if  $\tau_s \in I$  then
13:           $res_2 \leftarrow$  EVALSTATEFORMULA( $\sigma_2, s$ )
14:          if  $res_2 = true$  then
15:            return true
16:          end if
17:        end if
18:         $res_1 \leftarrow$  EVALSTATEFORMULA( $\sigma_1, s$ )
19:        if  $res_1 = false$  then
20:          return false
21:        end if
22:      end switch
23:      ( $s, [\tau_s, \tau_{s+1})$ )  $\leftarrow$  NEXTSTATE( $s, \tau_{s+1}$ )
24:    until  $\tau_{s+1} > \tau_{max}$ 
25:    return false
26: end procedure

```

interval I , i.e., $\tau_{s+1} \in I$. The state formula σ is going to be evaluated by Algorithm 14. Verifying operator $\sigma_1 U^I \sigma_2$ is done by evaluating σ_1 until a state s is reached such that $s \not\models \sigma_1$ or $s \models \sigma_2$ and state s is reached within the time interval I . If there was no decision made, the algorithm generates the next state s and the corresponding sojourn time interval $[\tau_s, \tau_{s+1})$ until a state is reached those exit time τ_{s+1} is greater than τ_{max} . In this case the path formula is returned to be *false*, because it was not possible to decide whether a state satisfied or falsified the path formula in the given interval I .

The evaluation of state formulas in Algorithm 14 is straight forward. The algorithm gets the state formula ϕ and the state s that the formula is to be evaluated on. Therefore, the procedure *evalStateFormula* is called recursively on all sub formulas. The function *evalAtomic*(a, s) evaluates the atomic

Algorithm 14 Evaluate state formula

```

1: procedure EVALSTATEFORMULA( $\sigma, s$ )
2:   switch  $\sigma$ 
3:     case  $a$  :
4:       return EVALATOMIC( $a, s$ )
5:     case  $\neg\sigma_1$  :
6:       return  $\neg$ EVALSTATEFORMULA( $\sigma_1, s$ )
7:     case  $\sigma_1 \wedge \sigma_2$  :
8:        $res_1 \leftarrow$  EVALSTATEFORMULA( $\sigma_1, s$ )
9:        $res_2 \leftarrow$  EVALSTATEFORMULA( $\sigma_2, s$ )
10:      return  $res_1 \wedge res_2$ 
11:     case  $\sigma_1 \vee \sigma_2$  :
12:        $res_1 \leftarrow$  EVALSTATEFORMULA( $\sigma_1, s$ )
13:        $res_2 \leftarrow$  EVALSTATEFORMULA( $\sigma_2, s$ )
14:      return  $res_1 \vee res_2$ 
15:   end switch
16:   return false
17: end procedure

```

proposition a by looking up the tokens that each place $x \in P(a)$ has in state s .

Example 18. We verify the following CSL formula on the \mathcal{SPN} in Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$. We want to know how likely it is that *producer* and *consumer* have one token at time point $\tau = 10$

$$\mathcal{P}_{=?} \left[true \text{ U}^{10,10} producer = 1 \wedge consumer = 1 \right] .$$

The expected probability computed by the numerical engine is 2.6036e-02. The simulative model checking computes a confidence interval of [2.4862e-02, 2.7461e-02] after 100 000 simulation runs, which is covering well the expected value.

5.2.3 Time-unbounded Formula

Simulative model checking of time-unbounded CSL formulas suffers from the same issues as described in Section 5.1. In the context of statistical CSL model checking, there were several approaches to tackle this issue. A method based on perfect simulation was presented in [RP09], but this method is only

applicable if the model under study is monotone. In addition, the authors do not discuss how to decide the monotonicity of a model. Another interesting approach may be the computation of termination-probability using rare-event simulation [LDT06], but there is still some work to do to make this applicable to simulative model checking. A method suitable for models of finite state space was presented in [YCZ11]. It incorporates reachability analysis using symbolic data structures and the results are very promising, but it is restricted to finite state spaces. Thus, we apply the approach, presented in Section 5.1, of detecting the steady state as a sufficient stopping criteria.

Algorithm 15 Evaluate time-unbounded path formula

```

1: procedure EVALUNBOUNDEDPATHFORMULA( $\phi$ )
2:   steadyStateReached  $\leftarrow$  false
3:    $(s, [\tau_s, \tau_{s+1})) \leftarrow \text{INITIALSTATE}(m_0)$ 
4:   repeat
5:     switch  $\phi$ 
6:       case  $X \sigma_1$  :
7:          $(s, [\tau_s, \tau_{s+1})) \leftarrow \text{NEXTSTATE}(s, \tau_{s+1})$ 
8:         return EVALSTATEFORMULA( $\sigma_1, s$ )
9:       case  $\sigma_1 \cup \sigma_2$  :
10:         $res_2 \leftarrow \text{EVALSTATEFORMULA}(\sigma_2, s)$ 
11:        if  $res_2 = \text{true}$  then
12:          return true
13:        end if
14:         $res_1 \leftarrow \text{EVALSTATEFORMULA}(\sigma_1, s)$ 
15:        if  $res_1 = \text{false}$  then
16:          return false
17:        end if
18:      end switch
19:      steadyStateReached  $\leftarrow \text{CHECKSTEADYSTATE}(s)$ 
20:       $(s, [\tau_s, \tau_{s+1})) \leftarrow \text{NEXTSTATE}(s, \tau_{s+1})$ 
21:    until steadyStateReached = true
22:    return false
23: end procedure

```

Algorithm 15 is mostly identically to Algorithm 13, but we omitted the check for the time intervals and added the steady state detection. The steady state detection is internally somewhat different to PLTLc, this is described in the next section.

Example 19. We adapt Example 18 and remove the time interval.

$$\mathcal{P}_{=?} [true \cup producer = 1 \wedge consumer = 1] \text{ .}$$

In this case a state that satisfies $producer = 1 \wedge consumer = 1$ is eventually reached at some time. The expected probability computed by the numerical engine is 1. The simulative model checking computes a confidence interval of $[1, 1]$ after 100 000 simulation runs, which is covering well the expected value.

5.2.4 Steady State Operator

The steady state operator \mathcal{S} was introduced in CSL in [Bai+00a] and describes the steady state behaviour of the CTMC. The formula $\mathcal{S}_{=?} [\sigma]$ is used to compute the probability to be in a state satisfying σ in a long-run. For the evaluation of steady state formulas, we want to keep the memory consumption as low as possible, i.e., we do not want to keep the complete trace in memory as in Section 5.1.3. That is why we apply the approach described in Section 4.3 for computing the steady state distribution using stochastic simulation. This approach has two advantages. First, the number of unique states in the steady state distribution is usually smaller than the length of the path generated, because states are usually visited not only once. Thus the memory consumption is lower. Second, there is no need to cut off some states, because the transient states are neglected in the steady state distribution.

Algorithm 16 shows the computation of the probability for steady state formula $\mathcal{S}_{=?} [\sigma]$. At first the steady state distribution is computed by generating a path and adding the visited states s and their sojourn time $\tau_{s+1} - \tau_s$ to the distribution π . While the path is generated, the sojourn times of each state are accumulated. The accumulated sojourn times are used to compute the steady state probability by Equation (4.36) and the steady state is reached if Equation (4.35) holds for all visited states. After reaching the steady state we iterate over all states and accumulate the probability of all states in which σ holds. In this way we get the steady state probability for irreducible CTMCs, if the CTMC is reducible we got the probability for only one recurrence class. So we have to repeat Algorithm 16 several times and average the resulting probabilities to take the recurrence classes into account.

Algorithm 16 Steady state computation for one simulation run

```

1: procedure EVALSTEADYSTATE( $\sigma$ )
2:    $steadyStateReached \leftarrow false$ 
3:    $(s, [\tau_s, \tau_{s+1}]) \leftarrow \text{INITIALSTATE}(m_0)$ 
4:    $\pi \leftarrow (s, \tau_{s+1} - \tau_s)$ 
5:   repeat
6:      $(s, [\tau_s, \tau_{s+1}]) \leftarrow \text{NEXTSTATE}(s, \tau_{s+1})$ 
7:      $\pi \leftarrow \pi + (s, \tau_{s+1} - \tau_s)$ 
8:      $steadyStateReached \leftarrow \text{CHECKSTEADYSTATE}(\pi, \epsilon)$ 
9:   until  $steadyStateReached = true$ 
10:   $pr \leftarrow 0$ 
11:  for  $i \leftarrow 0, |\pi|$  do
12:     $(s_i, pr_i) \leftarrow \pi_i$   $\triangleright$  state  $s_i$ , probability  $pr_i$ 
13:     $res \leftarrow \text{EVALSTATEFORMULA}(\sigma, s_i)$ 
14:    if  $res = true$  then
15:       $pr \leftarrow pr + pr_i$ 
16:    end if
17:  end for
18:  return  $pr$ 
19: end procedure

```

Example 20. We verify the following CSL formula on the \mathcal{SPN} in Example 6 with $B = 1$ and $m_0 = (0, 1, 0, 1, 0, 1)$. We want to know how likely it is in the long run that *producer* and *consumer* have one token.

$$\mathcal{S}_{=?} [producer = 1 \wedge consumer = 1] \text{ .}$$

The expected steady state probability computed by the numerical engine is 7.2165e-02. The simulative model checking computes a confidence interval of [6.5817e-02, 7.9159e-02] after 10 000 simulation runs, which is covering well the expected value.

5.3 Simulative Reward Computation

The Continuous Stochastic Logic has been extended in [KNP07] by special operators to compute the expectations of instantaneous and cumulative rewards. In order to realize this, a reward structure $(\underline{\rho}, \underline{\iota})$ is added to the CTMC, see Section 4.4.

The syntax of the reward extension of CSL is given in Definition 28.

Definition 28. Syntax of the reward extension of CSL:

$$\begin{aligned}
& \mathcal{R}_{\bowtie r} [C^{\leq \tau}] \mid \mathcal{R}_{=?} [C^{\leq \tau}] \mid \mathcal{R}_{\bowtie r} [I^{\tau}] \mid \mathcal{R}_{=?} [I^{\tau}] \mid \\
& \mathcal{R}_{\bowtie r} [F \sigma] \mid \mathcal{R}_{=?} [F \sigma] \mid \mathcal{R}_{\bowtie r} [\mathcal{S}] \mid \mathcal{R}_{=?} [\mathcal{S}] \\
& \bowtie \in \{<, \leq, \geq, >\}, \quad r, \tau \in \mathbb{R}_0^+
\end{aligned}
\tag{5.5}$$

The reward operator \mathcal{R} has two different modes. If it is used with the question mark as $\mathcal{R}_{=?}[\psi]$, then it will return the expected reward $E(\psi)$. We apply sample statistics as presented in Section 4.1, because $E(\psi)$ is the expected value of a random variable. We approximate the probability by computing the sample mean of the outcomes of N evaluations with

$$E(\psi) \approx \bar{Y} = \frac{1}{N} \sum_{n=1}^N Y_n . \tag{5.6}$$

As well as the probabilistic operator, the reward operator returns a confidence interval instead of a single value. In this case it is not possible to use the Wilson score confidence interval, Equation (5.2), or the Wald confidence interval, because both are defined for values in the interval $(0, 1)$. According to the central limit theorem [SM08], the confidence interval $[B_l, B_u]$ is computed from the sample mean \bar{Y} and the sample variance s^2 of the expected reward $E(\psi)$, the number of simulation runs N and the $Z = 1 - \alpha/2$ percentile of the normal distribution with the confidence level α by

$$[B_l, B_u] = \bar{Y} \pm Z \sqrt{\frac{s^2}{N}} . \tag{5.7}$$

The second case is mostly common with the probabilistic operator, $\mathcal{R}_{\bowtie x}[\psi]$ returns *true*, if $E(\psi) \bowtie x$ is fulfilled, *false* otherwise. We have to introduce an additional return value *unknown*, because of the indifference region defined by the confidence interval. It is returned, if the computed confidence interval covers the value x , because in that case we can not decide whether $E(\psi) \bowtie x$

is true or not. The return value of $\mathcal{R}_{\bowtie x}[\psi]$ is defined as

$$\mathcal{R}_{\bowtie x}[\psi] = \begin{cases} true & \text{if } x \bowtie [B_l, B_u] \wedge x \notin [B_l, B_u] \\ false & \text{if } x \not\bowtie [B_l, B_u] \wedge x \notin [B_l, B_u] \\ unknown & \text{if } x \in [B_l, B_u] . \end{cases}$$

A formula, defined in Definition 28, is satisfied in state s , if the following \models relations hold.

$$\begin{aligned} s \models \mathcal{R}_{\bowtie x} [C^{\leq \tau}] &\iff E(s, Y_{C^{\leq \tau}}) \bowtie x \\ s \models \mathcal{R}_{\bowtie x} [I^{\tau}] &\iff E(s, Y_{I^{\tau}}) \bowtie x \\ s \models \mathcal{R}_{\bowtie x} [F \sigma] &\iff E(s, Y_{F \sigma}) \bowtie x \\ s \models \mathcal{R}_{\bowtie x} [\mathcal{S}] &\iff \lim_{\tau \rightarrow \infty} \tau^{-1} \cdot E(s, Y_{C^{\leq \tau}}) \bowtie x \end{aligned}$$

The expected value of the random variable Y for any path $\omega \in Path(s)$ is denoted by $E(s, Y)$.

The random variable $Y_{C^{\leq \tau}}$ denotes the cumulated reward of state and impulse reward functions up to time point τ and is computed by Equation (4.41), i.e., Algorithm 17 line 8.

Let the random variable $Y_{I^{\tau}}$ be the expected state reward at time point τ , it is defined as

$$Y_{I^{\tau}} = \underline{\rho}(\omega @ \tau) , \quad (5.8)$$

where $\omega @ \tau$ is the state s_n in path ω at time point τ , i.e., Algorithm 17 line 15.

The random variable $Y_{F \sigma}$ denotes the expected reward to be accumulated before reaching a state satisfying σ . Let state s_n be the first state satisfying σ on path ω , then

$$Y_{F \sigma} = \begin{cases} 0 & \text{if } \omega(0) \models \sigma \\ \infty & \text{if } \forall s_i \in \omega, s_i \not\models \sigma \\ \sum_{i=0}^{n-1} \tau_i \cdot \underline{\rho}(s_i) + \iota(s_i, s_{i+1}) & \text{otherwise.} \end{cases} \quad (5.9)$$

The evaluation of $Y_{F \sigma}$ is divided in 3 parts. First, if the first state satisfies

σ , then the expected value is zero, because no reward could be accumulated. Second, if no state in path ω satisfies *sigma*, then the expectation is infinity, because the cumulated reward is monotonically increasing. In our simulative settings, we have to stop the trace generation at some point and we apply again the steady state property as in Section 5.2, i.e., Algorithm 17 line 18–25. The random variable Y_S denotes the expected reward in the long-run and is defined as

$$Y_S = \lim_{\tau \rightarrow \infty} \tau^{-1} \cdot \sum_{i \in \mathbb{N}} \tau_i \cdot \underline{\rho}(s_i) + \iota(s_i, s_{i+1}) . \quad (5.10)$$

Thus, we use the same approach as described in Section 4.3, but rather than the probability we compute the cumulated reward. Equation (5.10) matches the case that the CTMC is irreducible. In case the CTMC is reducible, we have to perform several simulation runs and average the resulting rewards.

Example 21. We verify the following reward formulas on the \mathcal{SPN} in Example 6 with $B = 1$, $m_0 = (0, 1, 0, 1, 0, 1)$ and the state reward function of Example 14. We want to know

1. the accumulated reward until time point $\tau = 10$

$$\mathcal{R}_{=?} [C^{\leq 10}] = 9.6257 \approx [9.6042, 9.6561],$$

2. the expected state reward at time point $\tau = 10$

$$\mathcal{R}_{=?} [I^{=10}] = 0.9013 \approx [0.8953, 0.9106],$$

3. the accumulated reward until a state is reached satisfying $producer = 1 \wedge consumer = 1$

$$\mathcal{R}_{=?} [F producer = 1 \wedge consumer = 1] = 4.0042 \approx [3.9239, 4.0866],$$

4. the accumulated reward in the long-run

$$\mathcal{R}_{=?} [S] = 0.8041 \approx [0.7936, 0.8141].$$

The expected reward values computed by the numerical engine are all covered by the confidence intervals computed by the simulative model checking

procedure after 10 000 simulation runs.

Algorithm 17 Evaluate reward formula

```

1: procedure EVALREWARDFORMULA( $\phi, m_0, \tau_0, \tau_{max}$ )
2:   ( $s, [\tau_s, \tau_{s+1})$ )  $\leftarrow$  INITIALSTATE( $m_0, \tau_0$ )
3:    $res \leftarrow 0$ 
4:   repeat
5:     switch  $\phi$ 
6:       case  $C^{\leq \tau}$  :
7:         if  $\tau_s \leq \tau$  then
8:            $res \leftarrow res + \text{EVALREWARD}(s, \tau_s, \tau_{s+1})$ 
9:         end if
10:        if  $\tau_{s+1} > \tau_{max}$  then
11:          return  $res$ 
12:        end if
13:       case  $I^{\tau}$  :
14:        if  $\tau \in [\tau_s, \tau_{s+1})$  then
15:          return  $\text{EVALREWARD}(s, \tau, \tau)$ 
16:        end if
17:       case  $F \sigma$  :
18:        if  $\text{EVALSTATEFORMULA}(\sigma, s) = \text{true}$  then
19:          return  $res$ 
20:        end if
21:         $\pi \leftarrow \pi + (s, \tau_{s+1} - \tau_s)$ 
22:        if  $\text{CHECKSTEADYSTATE}(\pi, \epsilon) = \text{true}$  then
23:          return  $\infty$ 
24:        end if
25:         $res \leftarrow res + \text{EVALREWARD}(s, \tau_s, \tau_{s+1})$ 
26:       case  $\mathcal{S}$  :
27:         $\pi \leftarrow \pi + (s, \tau_{s+1} - \tau_s)$ 
28:         $res \leftarrow res + \text{EVALREWARD}(s, \tau_s, \tau_{s+1})$ 
29:        if  $\text{CHECKSTEADYSTATE}(\pi, \epsilon) = \text{true}$  then
30:          return  $res / \tau_{s+1}$ 
31:        end if
32:     end switch
33:     ( $s, [\tau_s, \tau_{s+1})$ )  $\leftarrow$  NEXTSTATE( $s, \tau_{s+1}$ )
34:   until  $\text{true}$ 
35: end procedure

```

5.4 Simulative CSRL Model Checking

The Continuous Stochastic Reward Logic (CSRL), introduced in [Bai+00b; Hav+02], is a superset of the continuous stochastic logic, presented in Section 5.2 and is a specification formalism for performability measures. Now, the temporal operators are decorated additionally with a reward interval. Model checking CSRL requires to compute the joint distribution of the CTMC and the stochastic process representing the evolution of the accumulated rewards. This is much more involving than computing transient probabilities, since the latter does not feature the Markov property. However, there are various algorithms to compute numerically the performability measures [Clo+05; CH06], but their applicability was restricted to systems of a few thousand states only. This limit has been moved by several orders of magnitude to systems with billion states in [Sch14], due to symbolic on-the-fly analysis. Nevertheless, the size of the models is still growing, especially in systems biology, and exceeds quite fast this limit. By using simulation analysis, we can overcome this limit and analyse even larger systems at the expense of accuracy. The syntax of our adapted CSRL is given in Definition 29.

Definition 29. Syntax of the Continuous Stochastic Reward Logic:

$$\begin{aligned}
\psi &:= \mathcal{P}_{\bowtie x} [\phi] \mid \mathcal{P}_{=?} [\phi] \mid \mathcal{S}_{\bowtie x} [\sigma] \mid \mathcal{S}_{=?} [\sigma] \\
\bowtie &\in \{<, \leq, \geq, >\}, \quad x \in [0, 1] \\
\phi &:= X_J^I \sigma \mid F_J^I \sigma \mid G_J^I \sigma \mid \sigma U_J^I \sigma \mid \sigma \\
I &:= [x_1, x_2] = \{x \in \mathbb{R}_0^+ \mid x_1 \leq x \leq x_2\}, \text{ omit } I = [0, \infty) \\
J &:= [r_1, r_2] = \{r \in \mathbb{R}_0^+ \mid r_1 \leq r \leq r_2\}, \text{ omit } J = [0, \infty) \\
\sigma &:= \neg \sigma \mid \sigma \wedge \sigma \mid \sigma \vee \sigma \mid a \mid \text{true} \mid \text{false} \\
a &:= \text{value} \trianglelefteq \text{value} \\
\trianglelefteq &\in \{<, \leq, \geq, >, =, \neq\} \\
\text{value} &:= \text{value} \sim \text{value} \mid \text{Place} \mid \text{Int} \mid \text{Real} \mid \text{function} \\
\sim &\in \{+, -, *, /\}
\end{aligned}$$

«

It is quite similar to Definition 27, but extends the path operators with a reward interval. The reward interval J can be omitted, if it is defined as $J = [0, \infty)$,

and the CSRL formula becomes in fact a CSL formula. In the same way, the time interval I can be omitted, if it is defined as $I = [0, \infty)$. Now the CSRL formula becomes a continuous reward logic (CRL) formula. CSL and CRL are complementary [Bai+00b], i.e., CRL-properties over Markov reward models can be interpreted as CSL-properties over the derived continuous-time Markov chain. So we can check CRL formulas with the same algorithms as CSL formulas.

A CSRL state formula is satisfied in state s , if the following \models relations hold.

$$\begin{aligned}
s \models a &\iff s \models \text{evalAtomic}(a, s) \\
s \models \neg \sigma &\iff s \not\models \sigma \\
s \models \sigma_1 \wedge \sigma_2 &\iff s \models \sigma_1 \wedge s \models \sigma_2 \\
s \models \sigma_1 \vee \sigma_2 &\iff s \models \sigma_1 \vee s \models \sigma_2 \\
s \models \mathcal{P}_{\bowtie x}[\phi] &\iff \Pr(\omega \in \text{Path}(s) \mid \omega \models \phi) \bowtie x \\
s \models \mathcal{S}_{\bowtie x}[\sigma] &\iff \sum_{s' \models \sigma} \pi_s(s') \bowtie x
\end{aligned}$$

The function $\text{evalAtomic}(a, s)$ evaluates the atomic proposition a by looking up the tokens that each place $x \in P(a)$ has in state s .

A path formula is satisfied for any path $\omega \in \text{Path}(s)$, if the following \models relations hold.

$$\begin{aligned}
\omega \models X_J^I \sigma &\iff \omega^{(1)} \models \sigma \text{ and } \tau_0 \in I \text{ and } y_0 \in J \\
\omega \models F_J^I \sigma &\iff \exists \tau \in I : \omega @ \tau \models \sigma \text{ and } Y_\tau \in J \\
\omega \models G_J^I \sigma &\iff \forall \tau \in I : \omega @ \tau \models \sigma \text{ and } Y_\tau \in J \\
\omega \models \sigma_1 U_J^I \sigma_2 &\iff \exists \tau \in I : \omega @ \tau \models \sigma_2 \text{ and } \forall \tau' < \tau : \omega^{(\tau')} \models \sigma_1 \text{ and } Y_\tau \in J
\end{aligned}$$

The random variable Y_τ denotes the cumulated reward up to time point τ . In particular, the path ω satisfies $X_J^I \sigma$, if the sojourn time of the first state is $\tau_0 \in I$, the cumulated reward in the first state is $y_0 \in J$ and the successor state satisfies σ . Verifying operator $\sigma_1 U_J^I \sigma_2$ is done by evaluating σ_1 until a state s is reached such that $s \not\models \sigma_1$ or $s \models \sigma_2$, state s is reached within the

time interval I and the accumulated reward is in the reward interval J .

The model checking procedure for CSRL formulas is essentially the same as for CSL formulas, see Algorithm 18. We just have to accumulate the reward using Equation (4.41), while a new state is reached. Having done this, we can check the cumulated reward against the reward bounds on the path operators. Such a straightforward adaptation of the model checking procedure is possible in our simulative setting, because we operate directly on a path and in continuous-time. Additionally, there is no need to distinguish between different until formulas, as it is the case when using numerical techniques [Clo06; Sch14].

Example 22. We verify the following CSRL formula on the \mathcal{SPN} in Example 6 with $B = 1$, $m_0 = (0, 1, 0, 1, 0, 1)$ and the state reward function of

Algorithm 18 Evaluate time-bounded reward path formula

```

1: procedure EVALREWARDPATHFORMULA( $\phi, m_0, \tau_0, \tau_{max}$ )
2:    $(s, [\tau_s, \tau_{s+1}]) \leftarrow \text{INITIALSTATE}(m_0, \tau_0)$ 
3:    $y \leftarrow y + \text{EVALREWARD}(s, \tau_s, \tau_{s+1})$ 
4:   repeat
5:     switch  $\phi$ 
6:       case  $X_J^I \sigma_1$  :
7:         if  $\tau_{s+1} \notin I \vee y \notin J$  then
8:           end if
9:            $(s, [\tau_s, \tau_{s+1}]) \leftarrow \text{NEXTSTATE}(s, \tau_{s+1})$ 
10:          return EVALSTATEFORMULA( $\sigma_1, s$ )
11:       case  $\sigma_1 U_J^I \sigma_2$  :
12:         if  $\tau_s \in I \wedge y \in J$  then
13:            $res_2 \leftarrow \text{EVALSTATEFORMULA}(\sigma_2, s)$ 
14:           if  $res_2 = \text{true}$  then
15:             return true
16:           end if
17:         end if
18:          $res_1 \leftarrow \text{EVALSTATEFORMULA}(\sigma_1, s)$ 
19:         if  $res_1 = \text{false}$  then
20:           return false
21:         end if
22:       end switch
23:        $(s, [\tau_s, \tau_{s+1}]) \leftarrow \text{NEXTSTATE}(s, \tau_{s+1})$ 
24:        $y \leftarrow y + \text{EVALREWARD}(s, \tau_s, \tau_{s+1})$ 
25:   until  $\tau_{s+1} > \tau_{max}$ 
26:   return false
27: end procedure

```

Example 14. We want to know how likely it is that *producer* and *consumer* have one token at time point $\tau = 10$ and the accumulated reward is at most 8

$$\mathcal{P}_{=?} \left[true \, U_{0,8}^{10,10} \, producer = 1 \wedge consumer = 1 \right] .$$

Such kind of formula is used for performability analysis. The expected probability computed by the numerical engine is 1.1722e-02. The simulative model checking computes a confidence interval of [9.9442e-03, 1.5702e-02] after 100 000 simulation runs, which is covering well the expected value.

5.5 Closing Remarks

In this chapter we have shown how to verify certain properties, expressed in temporal logics, by the application of simulative model checking. We extended PLTLc with time-unbounded temporal operators and the steady state operator [Roh13]. Therefore, we exploited the steady state property to find an appropriate truncation point for the generated path. This was achieved by two different methods. First, the sample bath means algorithm *Skart* [Taf+11] was used for the verification of PLTLc formulas, because it operates directly on the generated path. Second, we approximated the steady state distribution as described in Section 4.3 for the verification of CS(R)L formulas, because this is more efficient in terms of memory than keeping the whole trace, which is not necessary for verifying CS(R)L formulas. The simulative model checking algorithm for CSRL formulas is the first to known and it supports state rewards as well as impulse rewards. We implemented all algorithms for simulative model checking of PLTLc, CSL and CSRL formulas in our model checker MARCIE [HRS13].

Chapter 6

Case Studies

In this chapter we provide some case studies to demonstrate the applicability of the different analysis methods presented in Chapter 4 and to verify certain properties expressed in temporal logics presented in Chapter 5. Furthermore, we compare the run-time performance of the introduced δ -leaping simulation method with the well known and widely used direct method on models of different size. We use models ranging from just a few nodes and arcs up to some thousand nodes and tens of thousand arcs.

The models originate from different areas, i.e., systems biology and technical systems. All Petri nets were modelled with Snoopy [RMH10; Hei+12] and analysed with MARCIE [SRH11; HRS13]. We start with five biochemical case studies of different sizes, presented from small to large. These models can be separated into two categories signal transduction networks and metabolic networks. The signal transduction networks are the RKIP inhibited ERK pathway (Section 6.1), the Mitogen-activated Protein Kinase (Section 6.2), the angiogenetic process (Section 6.3) and a simplified repressilator (Section 6.4). The metabolic networks are a reduced *E.coli* K-12 Metabolic model and a genome scale *E.coli* K-12 Metabolic model.

Afterwards we present two technical case studies, i.e., the flexible manufacturing systems (Section 6.6) and a cyclic server polling system (Section 6.7).

The experiments were carried out on two different machines. The first one is a MacPro with 2×Intel® Xeon® E5520 with 2.26GHz and 32GB RAM running Mac OSX 10.11, from now on we refer to it as *machine 1*. This machine is used to demonstrate the scalability of the shared memory, multi-threading

Table 6.1: The size of the state space for different initial markings of \mathcal{SPN}_{ERK} computed with MARCIE’s symbolic state space generation.

N	states	N	states	N	states	N	states
5	1,974	20	1,696,618	40	79,414,335	100	1.591×10^{10}
10	47,047	25	5,723,991	50	2.834×10^8	250	3.582×10^{12}
15	368,220	30	15,721,464	60	8.114×10^8	500	2.231×10^{14}

implementation. The second one has 4×AMD Opteron™ 6276 with 2.3 GHz and 256GB RAM running CentOS 6, from now on we refer to it as *machine 2*. It is used to demonstrate the scalability of the distributed memory, multi-processing implementation.

6.1 RKIP inhibited ERK pathway

This model shows the influence of the Raf Kinase Inhibitor Protein (RKIP) on the Extracellular signal Regulated Kinase (ERK) signalling pathway. A model of non-linear ordinary differential equations was originally published in [Cho+03]. Later on, it was discussed as qualitative and continuous Petri nets in [GH06], and as three related Petri net models in [HDG10]. The stochastic Petri net \mathcal{SPN}_{ERK} comprises 11 places and 11 transitions connected by 34 arcs and is shown in Figure 6.1.

All transition rate functions use mass action kinetics with the original parameter values from [HDG10]. There is one exception, the constant, used in scaling second order reactions, with a value of 2.5 violates the condition in Equation 3.14. Thus, we set this constant to 1, so that our condition is fulfilled. The model is scalable by the initial amount of tokens in the places *ERK*, *MEKpp*, *Raf1Star*, *RKIP* and *RP*. The more initial tokens N on each of these places, the bigger the state space of the Petri net. The number of reachable states for different initial markings is shown in Table 6.1.

We compared the well known and widely used direct method [Gil76] with the *delta*-leaping method [Roh16] and performed experiments with different values of $N = \{10, 100, 1000, 10\,000\}$ on *machine 1*. We present simulation results for four randomly chosen places, i.e., the results concerning the approximation error are comparable for all places of the model. They are given in Fig. 6.2

and the run-times are provided in Table 6.2. For all instances of N the results of direct method and δ -leaping match quite well. For $N = 10$ the simulation run-time is lower for the direct method than for δ -leaping. The simulation run-time increases for both algorithms with $N = 100$, but the discrete-time leap method is a little faster now. For $N = 1000$ and $N = 10\,000$ the run-times for the direct method are far away from the run-times of δ -leaping. Overall, the simulation run-time of the direct method increases by a factor of 10, whereas the run-time for δ -leaping increases by only 15%.

In order to verify the correctness of our simulative model checking approach, we check the same properties as in [Hei+10]. We use *machine 2* for this purpose. We first check the reachability of a state at some time in the future, such that

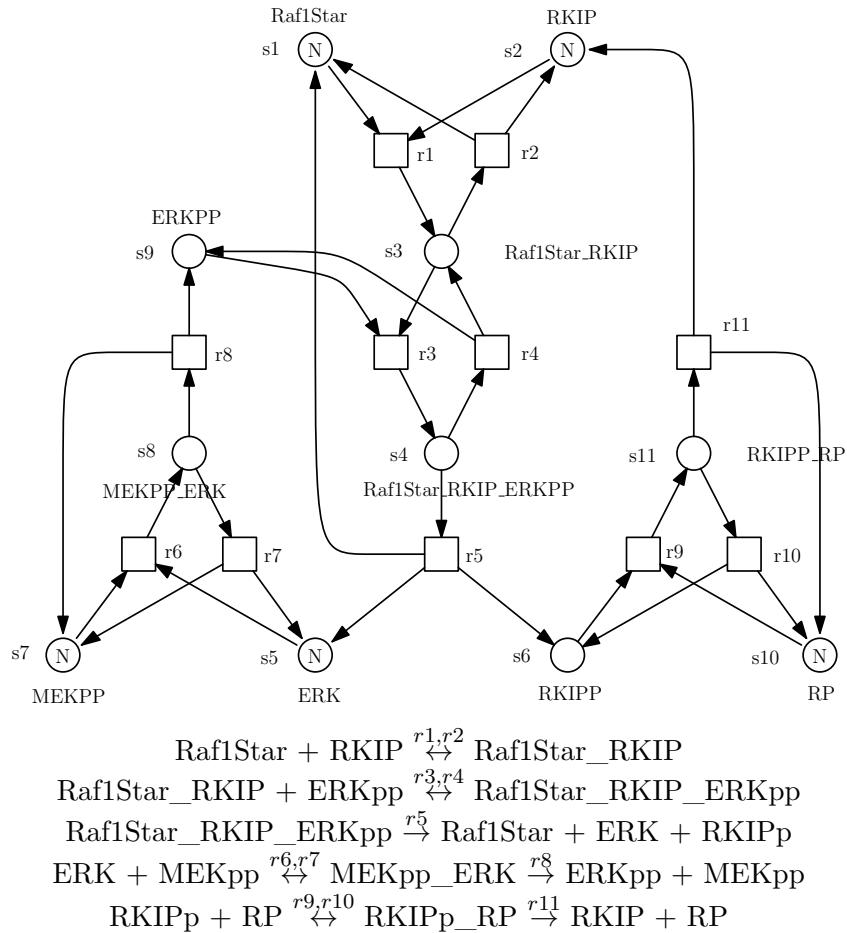


Figure 6.1: Stochastic Petri net of the RKIP inhibited ERK pathway, including textual representation of the chemical reactions [HDG10].

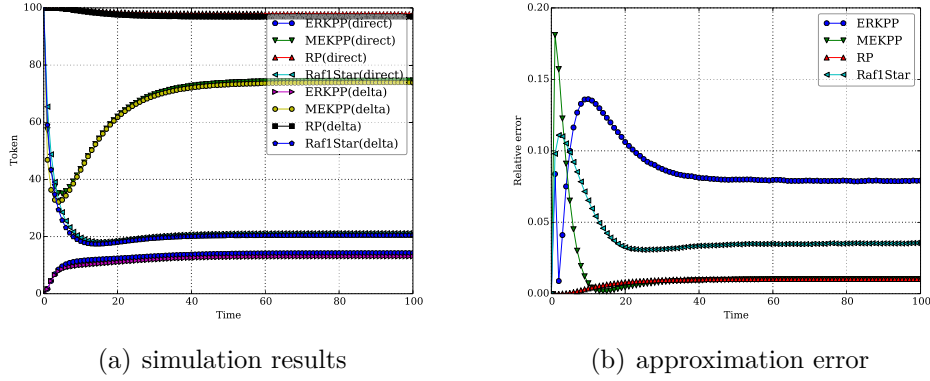


Figure 6.2: \mathcal{SPN}_{ERK} with $N = 100$ and 1 000 000 simulation runs

Table 6.2: Comparison of run-times for the direct method and δ -leaping. \mathcal{SPN}_{ERK} was parametrised with N and simulated with 1 000 000 simulation runs.

N	direct method	δ -leaping
10	2m8s	9m10s
100	17m48s	13m24s
1000	2h50m	15m28s
10 000	1d4h	17m10s

the number of tokens on place $MEKpp$ is between 60% and 80% of N :

$$\mathcal{P}_{=?} [\mathbf{F} [MEKpp \geq N \cdot 0.6 \wedge MEKpp \leq N \cdot 0.8]].$$

In any case such a state was reached, therefore the probability of the formula is 1. Figure 6.3 shows the run-time of the reachability analysis for different initial markings N and for several number of workers.

Since we know now that such a state is eventually reached, we want to compute the steady state probability of being in such a state, where the number of tokens on place $MEKpp$ is between 60% and 80% of N :

$$\mathcal{S}_{=?} [MEKpp \geq N \cdot 0.6 \wedge MEKpp \leq N \cdot 0.8].$$

First the results in Table 6.3 show that the resulting confidence interval covers the probability computed by the Jacobi method in [Hei+10]. Second the algorithm scales nearly linear with the number of worker processes, see Figure 6.4.

Table 6.3: Steady state analysis for different initial markings N of \mathcal{SPN}_{ERK} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine.

N	Pr	CI
20	0.77508	[0.77482, 0.77534]
30	0.83297	[0.83277, 0.83325]
40	0.87452	[0.87416, 0.87470]
50	0.90465	[0.90437, 0.90486]
60	0.92682	[0.92641, 0.92696]

A very interesting behaviour regards the relationship between the state space size and the total run-time of the computation. One could expect an increase of the run-time, but it stays the same. This is a result of the level semantics described in [Cal+06], i.e., the rate functions are scaled by the initial number of tokens N . Therefore, the sojourn time of the transition remains the same, while the initial amount of tokens is increasing. In contrast to the numerical engine, the stochastic rate functions are decisive for the run-time of the stochastic simulation and not the size of the state space.

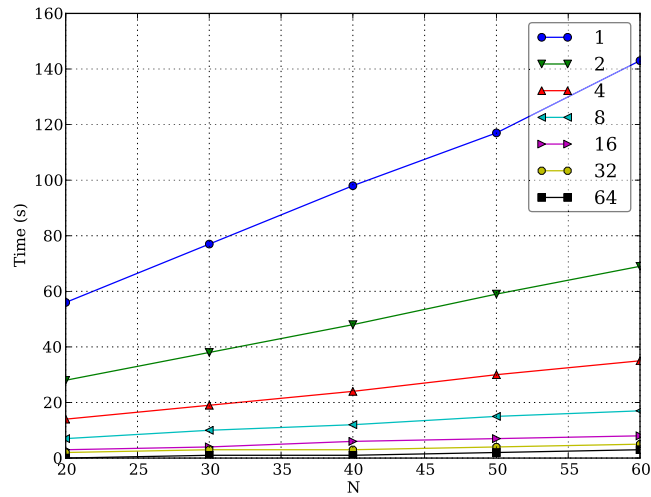


Figure 6.3: Transient analysis for different initial markings N of \mathcal{SPN}_{ERK} . The total run-time is given for several number of workers.

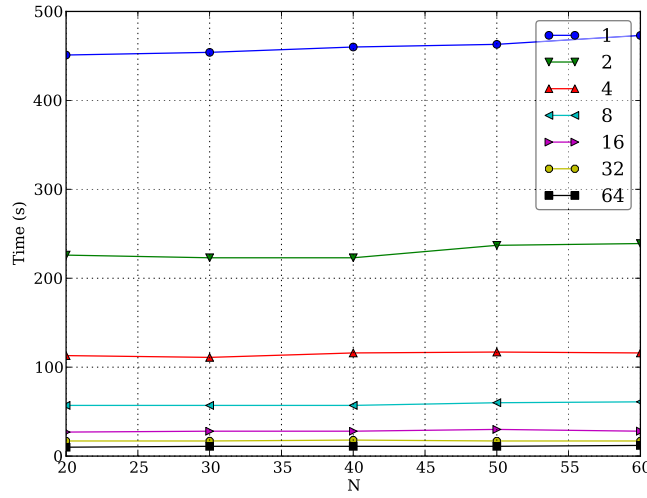


Figure 6.4: Steady state analysis for different initial markings N of \mathcal{SPN}_{ERK} . The total run-time is given for different numbers of workers.

6.2 Mitogen-activated Protein Kinase

The mitogen-activated protein kinase (MAPK) is the core of the ERK/MAPK pathway that can, for example, carry cell division and differentiation signals from the cell membrane to the nucleus. The model was published in [LBS00] and later on, discussed as stochastic Petri net in [HGD08].

The Petri net \mathcal{SPN}_{MAPK} comprises 22 places and 30 transitions connected by 90 arcs. The model is scalable by the initial amount of tokens in six places. All transition rate functions of \mathcal{SPN}_{MAPK} use mass action kinetics with rate constants taken from [HGD08].

The number of reachable states for different initial markings are shown in Table 6.4.

Now, let us compare the well known and widely used direct method [Gil76]

Table 6.4: The size of the state space for different initial markings of \mathcal{SPN}_{MAPK} computed with MARCIE's symbolic state space generation.

N	states	N	states	N	states	N	states
2	6.110×10^6	8	2.712×10^{13}	14	4.197×10^{16}	20	5.635×10^{18}
4	6.920×10^9	10	4.783×10^{14}	16	2.584×10^{17}	40	1.064×10^{23}
6	7.694×10^{11}	12	5.296×10^{15}	18	1.306×10^{18}	80	2.616×10^{27}

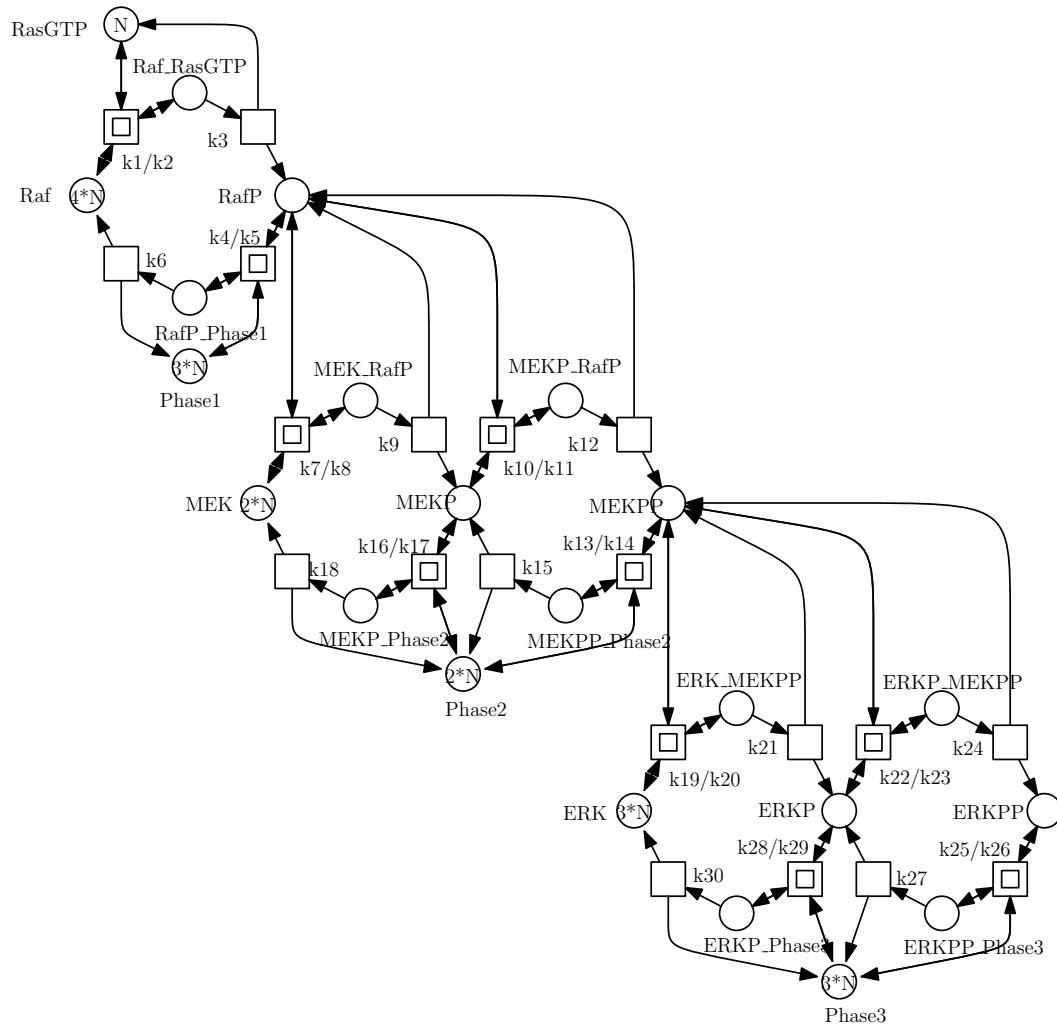


Figure 6.5: Stochastic Petri net of the mitogen-activated protein kinase [HGD08].

Table 6.5: Comparison of run-times for the direct method and δ -leaping. \mathcal{SPN}_{MAPK} was parametrised with N and simulated with 1 000 000 simulation runs.

N	direct method	δ -leaping
10	19m36s	17m20s
100	3h13m	25m52s
1000	1d6h	31m52s
10 000	12d19h	39m55s

with the *delta*-leaping method [Roh16]. We performed experiments with the following different values of $N = \{10, 100, 1000, 10\,000\}$ and all were carried out on *machine 1*.

Exemplary simulation results of four places for $N = 100$ are given in Fig. 6.6. They match quite well for the given places, as well as for the others. The run-time behaviour for this model develops in a comparable way to \mathcal{SPN}_{ERK} , see Table 6.5. The run-time of the direct method in the first instance $N = 10$ is lower than for δ -leaping, but it increases much faster in the other instances of N .

We investigate the behaviour of place *RafP*. At first we want to know how likely it is that *RafP* contains no tokens from the beginning up to time point $\tau = 1$. We check this property using the following formula:

$$\mathcal{P}_{=?} \left[G^{0,\tau} [RafP = 0] \right].$$

The results in Table 6.6 show that it is very likely for *RafP* to contain no token

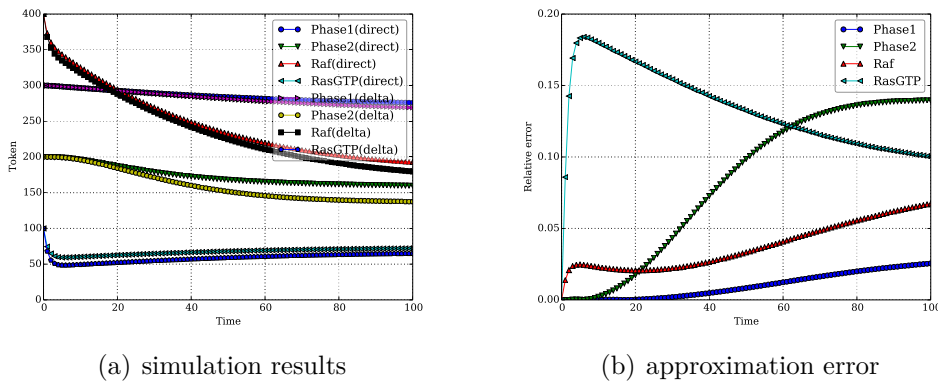


Figure 6.6: \mathcal{SPN}_{MAPK} with $N = 100$ and 1 000 000 simulation runs

Table 6.6: Transient analysis up to time point $\tau = 1$ for different number of stations N of \mathcal{SPN}_{MAPK} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.

N	Pr	CI
1	0.98491	[0.98475, 0.98499]
2	0.97039	[0.97026, 0.97059]
3	0.95609	[0.95597, 0.95638]
4	–	[0.94171, 0.94218]
5	–	[0.92774, 0.92826]
6	–	[0.91444, 0.91500]

up to time point 1 and the probability is just slightly decreasing. This is not a surprise but a consequence of the scaled transition rates taken from [HGD08]. The confidence intervals computed by simulative model checking cover the expected probability very well, but the numerical engine could not compute results for $N > 3$, because of the large state space and thus insufficient memory. Figure 6.7 shows the total run-times of the transient analysis up to time point $\tau = 1$ for different initial markings N . The run-times are given for 1 to 8 worker threads after 6 634 234 simulation runs. They increase as N increases and for each N the run-time is cut nearly into halves as the number of worker threads doubles.

As we now know that in the beginning of the transient phase $RafP$ does not hold a token most of the time, we are now interested in the long run behaviour. Therefore we want to evaluate the probability that $RafP$ contains no token in the steady state. We achieve this by verifying the following formula:

$$\mathcal{S}_{=?} [RafP = 0].$$

The results in Table 6.7 show that it is much more likely for place $RafP$ to carry at least one token in the steady state, than in the beginning of the transient phase. The probability decreases, as N increases.

The coverage of the confidence intervals is as good as in the previous case. The run-times, shown in Figure 6.8, exhibit the same characteristics as in the transient case, i.e., the simulative model checking scales quite well with the number of worker threads.

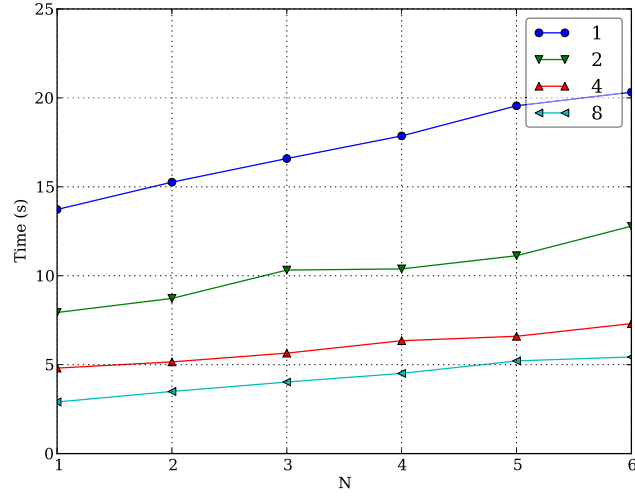


Figure 6.7: Transient analysis up to time point $\tau = 1$ for different initial markings N of \mathcal{SPN}_{MAPK} . The total run-time is given for several number of worker threads after 6 634 234 simulation runs.

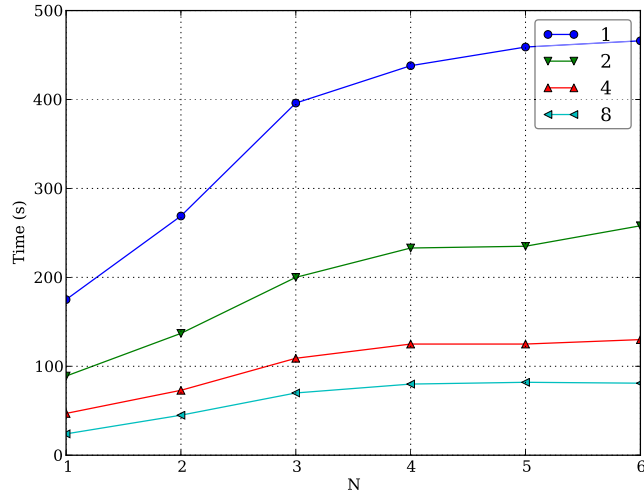


Figure 6.8: Steady state analysis for different initial markings N of \mathcal{SPN}_{MAPK} . The total run-time is given for different numbers of workers after 128 simulation runs.

Table 6.7: Steady state analysis for different number of stations N of \mathcal{SPN}_{MAPK} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 128 simulation runs.

N	Pr	CI
1	2.609×10^{-1}	$[2.595 \times 10^{-1}, 2.610 \times 10^{-1}]$
2	7.196×10^{-2}	$[7.183 \times 10^{-2}, 7.211 \times 10^{-2}]$
3	—	$[2.000 \times 10^{-2}, 2.011 \times 10^{-2}]$
4	—	$[5.742 \times 10^{-3}, 5.793 \times 10^{-3}]$
5	—	$[1.753 \times 10^{-3}, 1.780 \times 10^{-3}]$
6	—	$[6.982 \times 10^{-4}, 7.149 \times 10^{-4}]$

6.3 Angiogenesis

Angiogenesis is a complex phenomenon that goes from a molecular level to macroscopic events. This Petri net models a part of the signal transduction pathway involved in the angiogenetic process and was originally published in [Nap+09]. The stochastic Petri net \mathcal{SPN}_{ANG} comprises 39 places and 64 transitions connected by 185 arcs.

The model is scalable by the initial amount of tokens in the places *Akt*, *DAG*, *Gab1*, *KdStar*, *Pip2*, *P3k*, *Pg* and *Pten*. The more initial tokens on each of these places, the bigger the state space of the Petri net. The number of reachable states for different initial markings are shown in Table 6.8.

As in Section 6.1, we check for reachability first. We use *machine 2* for this purpose. Now we want to know the probability of eventually reaching a state where no tokens reside on place *Akt*:

$$\mathcal{P}_{=?} [\text{F } [Akt = 0]].$$

Table 6.8: The size of the state space for different initial markings of \mathcal{SPN}_{ANG} computed with MARCIE's symbolic state space generation. The places *Akt*, *DAG*, *Gab1*, *KdStar*, *Pip2*, *P3k*, *Pg* and *Pten* carry initially N tokens.

N	states	N	states	N	states	N	states
1	96	4	2, 413, 480	7	2.181×10^9	10	4.537×10^{11}
2	5, 384	5	29, 224, 050	8	1.464×10^{10}	15	5.207×10^{14}
3	144, 188	6	277, 789, 578	9	8.623×10^{10}	20	1.428×10^{17}

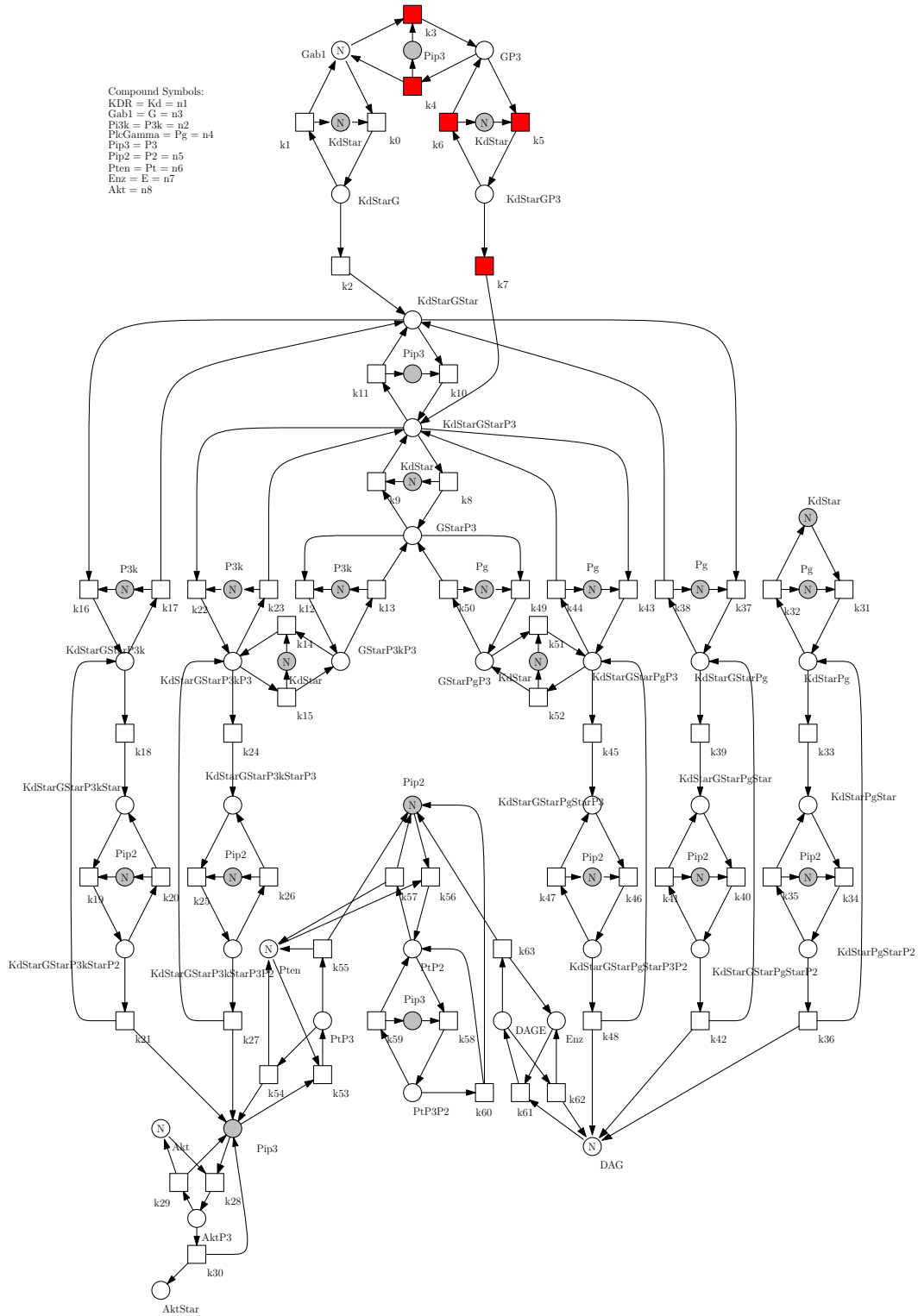


Figure 6.9: Stochastic Petri net of the angiogenesis process [Nap+09].

Table 6.9: Transient analysis for different initial markings N of \mathcal{SPN}_{ANG} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine.

N	Pr	CI
1	0.44141	[0.43542, 0.44642]
2	0.80836	[0.80292, 0.81370]
3	0.92899	[0.92319, 0.93365]
4	0.97950	[0.97189, 0.98216]
5	—	[0.99380, 0.99396]
6	—	[0.99760, 0.99770]

In contrast to \mathcal{SPN}_{ERK} , Table 6.9 shows that the probability ranges from about 0.44 ($N = 1$) to 0.9 ($N = 6$). That means a state where no tokens lay on place *Akt* is not always reached, because the CTMC consists of several strongly connected components and in some of them such a state does not exist.

Figure 6.10 shows the run-time of the reachability analysis for different initial markings N and for several number of workers.

Second we compute the steady state probability of being in a state that has no tokens on place *Akt*:

$$\mathcal{S}_{=?} [Akt = 0] .$$

The results in Table 6.10 show that the steady state probability is nearly the

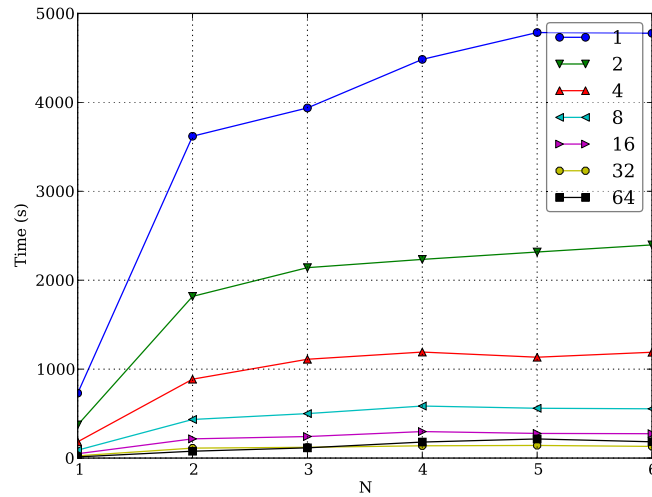


Figure 6.10: Transient analysis for different initial markings N of \mathcal{SPN}_{ANG} . The total run-time is given for several number of workers.

Table 6.10: Steady state analysis for different initial markings N of \mathcal{SPN}_{ANG} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine.

N	Pr	CI
1	0.44141	[0.43773, 0.44771]
2	0.80836	[0.80446, 0.81237]
3	0.92899	[0.92772, 0.93284]
4	0.97950	[0.97859, 0.98140]
5	—	[0.98923, 0.99121]
6	—	[0.99649, 0.99758]

same as in the reachability case as the overall steady state probability consists of two parts, first the probability of reaching a strongly connected component and second the steady state probability inside these component. The result means the steady state probability inside a strongly connected component, where a state exists with $Akt = 0$, is almost 1. That is why the overall steady state probability almost coincides with the reachability probability.

Figure 6.11 shows the run-time of the steady state analysis for different initial markings N and for several number of workers.

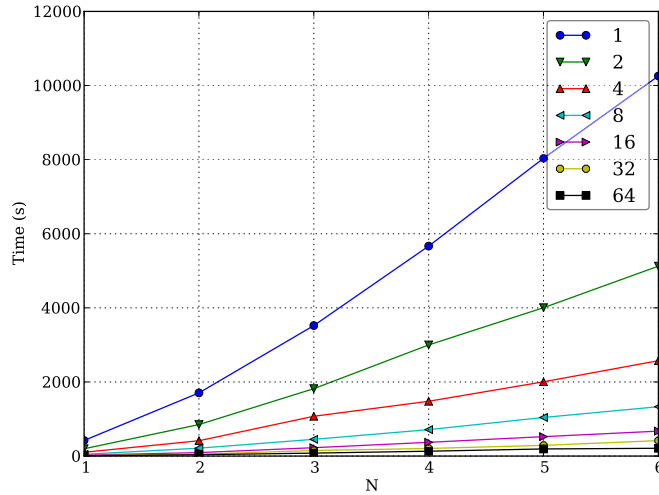


Figure 6.11: Steady state analysis for different initial markings N of \mathcal{SPN}_{ANG} . The total run-time is given for different numbers of workers.

of models.

Figure 6.13 shows stochastic simulation of \mathcal{SPN}_{SR}^c for 1 copy, 1000 copies per gene and for 1, 1000 simulation runs. Performing one simulation with only a single copy of each gene results in an oscillation superimposed by random fluctuations. The random fluctuations diminish if the number of copies per gene are increased. Averaging the results over several simulation runs reduces the amplitude of the oscillation as random fluctuations superimpose.

In a next step we reproduce a property from [LH14]: We want to explore the value range of the proteins up to time point τ . Therefore we use the *free variables* of PLTLc. A free variable is specified by a leading \$. Let $\$x$ be a free variable, the value range of the proteins is computed by the following formula:

$$\mathcal{P}_{=?} \left[F^{0,\tau} [p_1 > \$x \vee p_2 > \$x \vee p_3 > \$x] \right] .$$

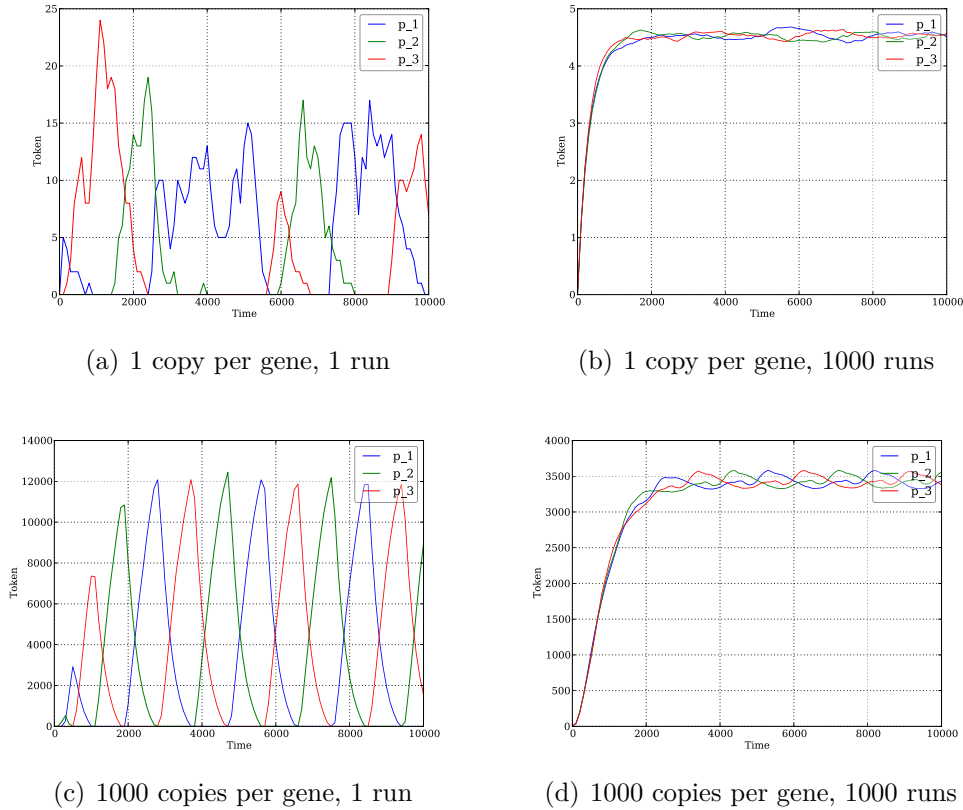


Figure 6.13: Stochastic simulations of the simplified repressilator for 1 copy, 1000 copies per gene and for 1, 1000 simulation runs.

The domain of the free variable x , i.e., the value range of the proteins, is shown in Figure 6.14. It shows not only the appeared value ranges, but the probability for each of them. Our results differ from [LH14], but this was expected, because of the different rate constants. In our case it is very unlikely for each protein to have more than 40 tokens.

In the next step, we apply simulative steady state computation to \mathcal{SPN}^c_{SR} , because we want to know the probability distribution of protein p_1 in the steady state. The computed distribution is given in Figure 6.15 and it sustains our previous assumption that it is very unlikely for a protein to carry more than 40 tokens.

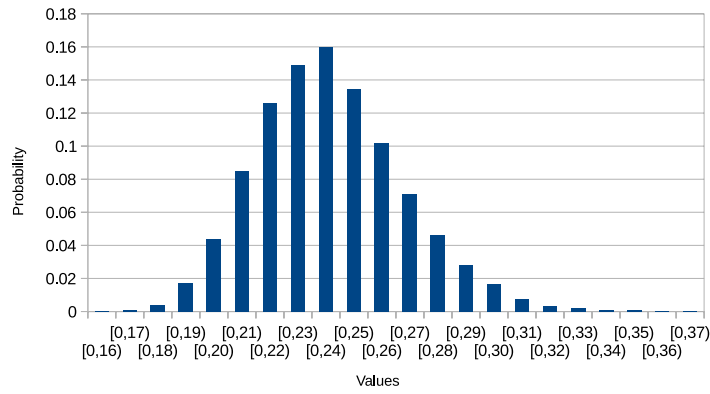


Figure 6.14: Probabilities of the value ranges on the places p_i up to time point $\tau = 10\,000$.

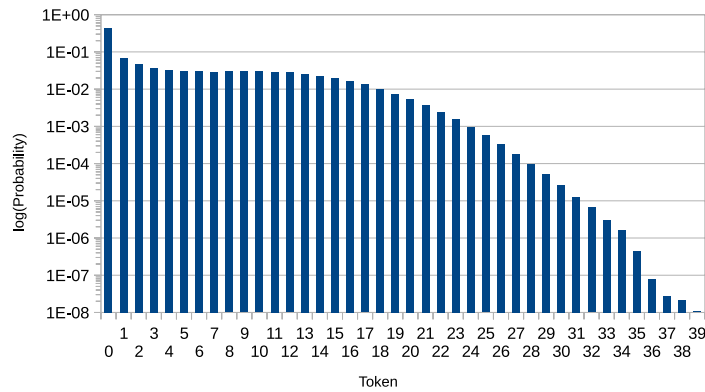


Figure 6.15: Steady state probability distribution of the number of tokens on place p_1 .

6.5 *E.coli* K-12 Metabolic model

This model describes the whole genome metabolism of *E.coli* K-12 iJO1366 substrain MG1655, and is one of the 55 GEM models published by [Mon+13]. We have chosen this strain as an example, because it was the first strain of *E.coli* to be sequenced; it is considered to be the best curated model and thus it has been used as the basis for reconstructing the models of the other strains of *E.coli*. The used versions of the model are currently part of investigations concerning structural issues and were provided by [GH16]. All experiments in this section were carried out on *machine 1*.

6.5.1 Reduced *E.coli* K-12 Metabolic model

This reduced model has been developed to illustrate the basic structure of the whole genome metabolism of *E.coli* K-12. The reduction was originally done by hand [OFP10] and subsequently used for comparison with the results of an automated procedure [ESK15]. The Petri net comprises 93 places and 208 transitions connected by 649 arcs. The model is scalable by the initial amount of tokens in 12 places belonging to a place invariant. It has some places with a high connectivity (Fig. 6.16(b)), e.g., M_h_c (53), M_h2o_c (28) and M_h_e (27), there are 18 places with a connectivity of ≥ 10 . These places are involved in many reactions and changing their values leads to many updates of transition rates. This has a strong influence on the overall speed of the stochastic simulation.

The model is assumed to have mass action kinetics, but it is not parametrized. We decided to set all kinetic constants to 1 and applied the scaling of second and higher order reactions as suggested in [Wil06].

We performed experiments with the following values of $N=\{100, 1000, 10\,000, 100\,000\}$. The simulation results in Fig. 6.17 and the run-times in Table 6.11 are quite interesting. The trajectories correlate quite well in the direct method and δ -leaping. The run-time variations differing in several orders of magnitude, speak for themselves. In case of *delta*-leaping, the run-time increases moderately with N increasing, but the run-time of the direct method increases drastically, showing the limited use of exact stochastic simulation for higher values of N .

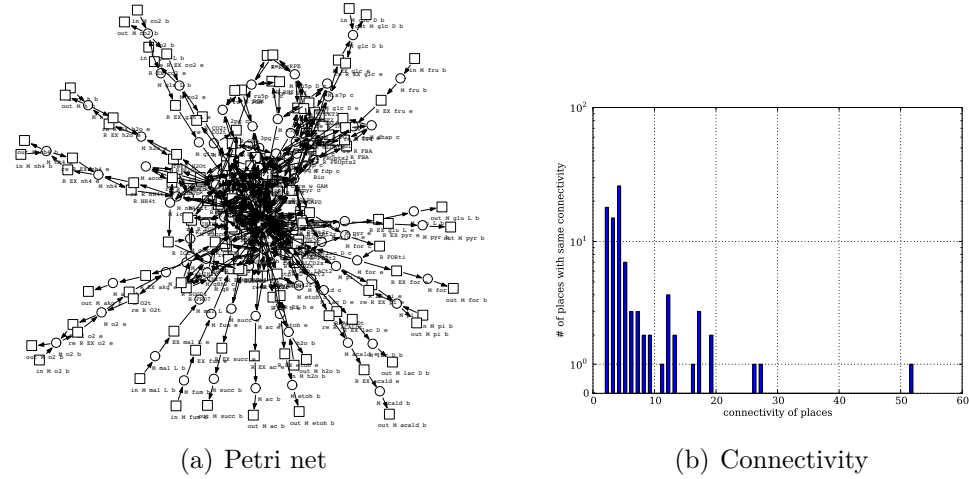


Figure 6.16: Petri net (a) and connectivity (b) of the reduced *E.coli* K-12 core model.

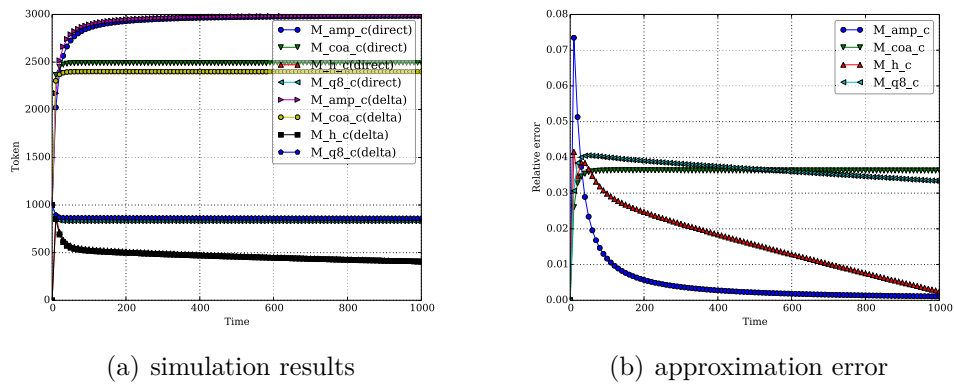


Figure 6.17: *E.coli* core with $N = 1000$ and 1 000 000 simulation runs

Table 6.11: Comparison of run-times for the direct method (a) and δ -leaping (b). \mathcal{SPN}_{CORE} was parametrised with N and simulated with several number of simulation runs. \dagger is placed, if the simulation did not finish in reasonable time (>40 days).

N	1 run		100 runs		100 000 runs		1 000 000 runs	
	a	b	a	b	a	b	a	b
100	<1s	<1s	8s	6s	1h58m	1h38m	21h4m	15h44m
1000	<1s	<1s	24s	7s	6h15m	1h41m	2d15h	16h16m
10 000	4s	<1s	3m28s	8s	2d3h	1h49m	20d5h	17h52m
100 000	15s	<1s	34m16s	9s	21d14h	2h6m	\dagger	20h40m

The model is still under investigation for its structural correctness and there are no kinetic rate constants available; so the simulation results may not correspond to wet lab experiments.

6.5.2 *E.coli* K-12 Genome Scale Metabolic model

This model describes the whole genome metabolism of *E.coli* K-12 iJO1366 substrain MG1655. The Petri net comprises 2130 places and 4162 transitions connected by 13571 arcs. The model is scalable by the initial amount of tokens in 101 of 2046 places. The model is supposed to be rather dense. This is confirmed by looking at the place connectivity in Fig. 6.18(b). The top six places at the connectivity ranking are M_h_c with 1198 arcs, M_h2o_c with 617 arcs, M_atp_c with 402 arcs, M_h_p with 369 arcs, M_pi_c with 339 arcs and M_adp_c with 314 arcs. Places with such high connectivity have a large impact on the performance of stochastic simulation, because the greater connectivity of a place, the more transitions change the number of tokens on this place and the more transition rates have to be evaluated each time this happens.

The model is assumed to have mass action kinetics, but it is not parametrized. We decided to set all kinetic constants to 1 and applied the scaling of second and higher order reactions as suggested in [Wil06].

We performed experiments with the following values of $N=\{100, 1000, 10\,000,$

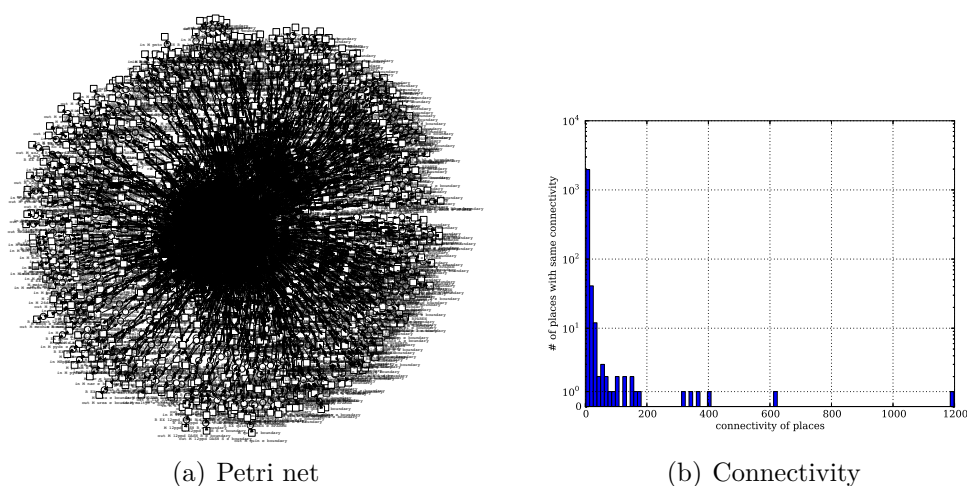


Figure 6.18: Petri net (a) and connectivity (b) of the *E.coli* K-12 genome scale metabolic model.

Table 6.12: Comparison of run-times for the direct method (a) and δ -leaping (b). \mathcal{SPN}_{ECOLI} was parametrised with N and simulated with several number of simulation runs. \dagger is placed, if the simulation did not finish in reasonable time (>40 days).

N	1 run		100 runs		100 000 runs		1 000 000 runs	
	a	b	a	b	a	b	a	b
100	17s	1s	38m56s	3m20s	21d15h	1d21h	\dagger	19d7h
1000	2m28s	2s	5h17m	4m53s	\dagger	2d16h	\dagger	26d16h
10 000	24m28s	2s	1d6h	5m20s	\dagger	2d21h	\dagger	28d18h
100 000	3h43m	3s	20d22h	5m52s	\dagger	3d2h	\dagger	30d20h

100 000}.

The simulation results given in Fig. 6.19, as well as the run-times in Table 6.12 are quite interesting. The plots of the randomly chosen places are comparable and the approximation error is moderate. We were not able to finish a higher number of exact stochastic simulations within 40 days. The simulation run-times for δ -leaping are acceptable for such a big and dense model. The scaling parameter N has small influence on the simulation run-time of δ -leaping. The simulation run-times for δ -leaping are moderate for such a big and dense model.

6.6 Flexible Manufacturing System

The Flexible Manufacturing System with three machines has been published in [CT93]. The original model contains immediate transitions; thus it is a

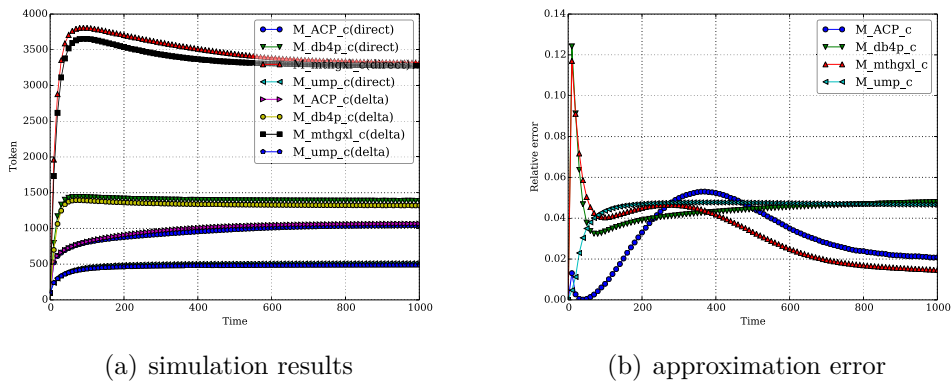


Figure 6.19: *E.coli* K-12 with $N = 100$ and 100 000 simulation runs

Table 6.13: The size of the state space for different initial markings of \mathcal{GSPN}_{FMS} computed with MARCIE's symbolic state space generation.

N	states	N	states	N	states	N	states
2	3,444	8	2.480×10^8	14	9.928×10^{10}	20	6.029×10^{12}
4	438,600	10	2.501×10^9	16	4.520×10^{11}	30	7.737×10^{14}
6	15,126,440	12	1.790×10^{10}	18	1.760×10^{12}	50	4.240×10^{17}

Table 6.14: Transient analysis up to time point $\tau = 1$ for different number of items N of \mathcal{GSPN}_{FMS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.

N	Pr	CI
1	2.521×10^{-1}	$[2.516 \times 10^{-1}, 2.525 \times 10^{-1}]$
2	1.796×10^{-1}	$[1.792 \times 10^{-1}, 1.800 \times 10^{-1}]$
4	7.063×10^{-2}	$[7.032 \times 10^{-2}, 7.083 \times 10^{-2}]$
6	3.048×10^{-2}	$[3.026 \times 10^{-2}, 3.061 \times 10^{-2}]$
8	—	$[1.352 \times 10^{-2}, 1.375 \times 10^{-2}]$
10	—	$[6.223 \times 10^{-3}, 6.382 \times 10^{-3}]$

\mathcal{GSPN} . A pure \mathcal{SPN} model can be derived from the \mathcal{GSPN} model by applying the elimination rules for immediate transitions given in [Ajm+95]. In contrast to [SRH11], we consider the \mathcal{GSPN} model to demonstrate the application of simulative model checking on \mathcal{GSPN} models. Furthermore, the FMS model contains arcs with marking-dependent weights. MARCIE does not support such arcs as they potentially destroy the locality principle. Instead, our model simulates the marking dependencies by additional transitions, each representing a specific firing situation in the original model. We achieve this by using coloured stochastic Petri nets and let the unfolding create the additional transitions. Besides that, the coloured model is equivalent to the uncoloured one, that is why we still speak about a \mathcal{GSPN} model. Figure 6.20 shows the \mathcal{GSPN}^c model without coloured annotations, the full specification in CANDL syntax is given in Appendix A.5. The FMS is scalable concerning the number of items which can be processed by the machines. The places $P1$, $P2$ and $P3$ carry initially N tokens. The model can be easily scaled by increasing the value of N . The number of reachable states for different initial markings are shown in Table 6.13. All experiments in this section were carried out on *machine 1*.

Table 6.15: Steady state analysis for different number of items N of \mathcal{GSPN}_{FMS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 128 simulation runs.

N	Pr	CI
1	0.96319	[0.95980, 0.96757]
2	0.97434	[0.97337, 0.97487]
4	0.97496	[0.97406, 0.97448]
6	0.97609	[0.97500, 0.97514]
8	—	[0.97596, 0.97616]
10	—	[0.97708, 0.97726]

and for each N the run-time is cut nearly into halves as the number of worker threads doubles.

Now, we want to explore the steady state probability that there are no tokens on the places $P1$, $P2$, $P3$ and $P12$. This property is expressed in the following formula:

$$\mathcal{S}_{=?} [P1 = 0 \wedge P2 = 0 \wedge P3 = 0 \wedge P12 = 0] \text{ .}$$

The results in Table 6.15 show a different picture than in the transient analysis. The probability is above 96% and increases slightly with increasing number of items. Thus the probability is below 4% that at least one machine has at least

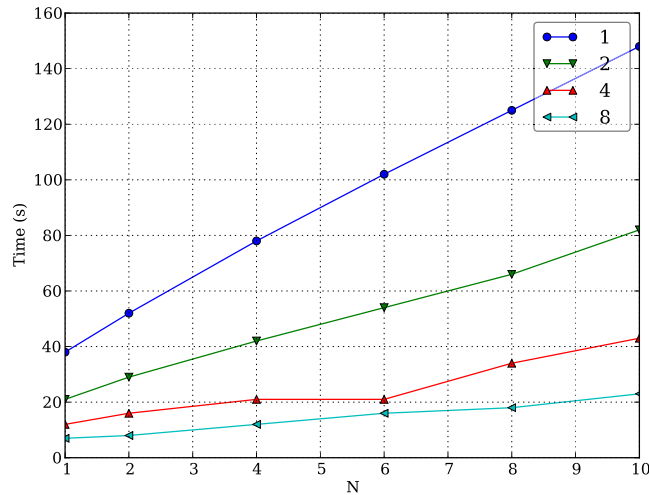


Figure 6.21: Transient analysis up to time point $\tau = 1$ for different number of items N of \mathcal{GSPN}_{FMS} . The total run-time is given for several number of workers after 6 634 234 simulation runs.

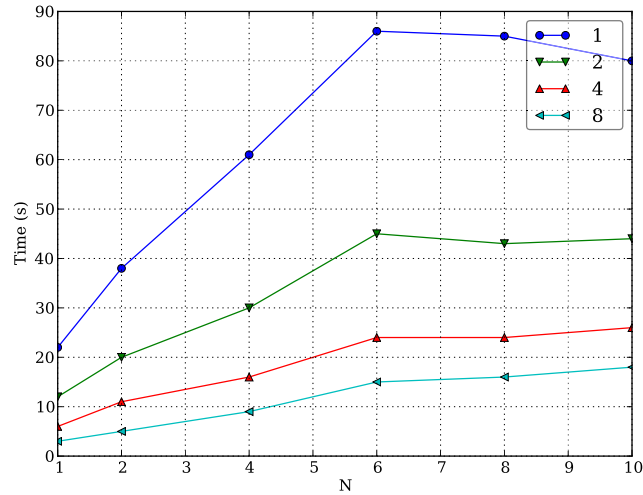


Figure 6.22: Steady state analysis for different number of items N of \mathcal{GSPN}_{FMS} . The total run-time is given for different numbers of workers after 128 simulation runs.

one item that needs to be processed, i.e., the degree of utilisation is above 96% in the long run. Table 6.15 reveals a caveat concerning \mathcal{GSPN} models. The expected probabilities for $N = 3$ and $N = 4$ are slightly higher than the confidence intervals computed by the simulative engine. Despite the small differences, this is in need for further investigation.

The run-times, shown in Figure 6.22, exhibit the same characteristics as in the transient analysis, i.e., the simulative model checking scales quite well with the number of worker threads.

6.7 Cyclic Server Polling System

Polling systems have a wide range of applications for which they provide good performance estimates, e.g., computer science, manufacturing, telecommunications. We discuss a cyclic single-server polling system, the simplest and most common polling system. It comprises one waiting line per station to be served and the waiting line is filled with customers from the outside world. One server cycles through the stations and provides service to the customers if needed. Afterwards the customers disappear from the system.

Such polling systems were presented as \mathcal{GSPN} in [IT90] and [ADN89]. Later

on it was shown how to reduce them to \mathcal{SPN} by applying several reduction rules [Ajm+95].

We provide a coloured stochastic Petri net of a cyclic single-server polling system \mathcal{SPN}_{CSPS}^c , shown in Figure 6.23. The server is modelled by 2 places s and a , place s denotes the station that the server is currently investigating. The number of tokens on s refers to the index of the station si_n and is initialised with one. The place a carries no token as long as the server is looking for a station to serve, if the server found a station then a token is placed on a and it is removed after service. A station si is modelled by a single place, representing its queue that holds the customer (token) to be served. The model is scalable by the number N of stations si . All experiments in this section were carried out on *machine 1*.

The number of reachable states for different amount of stations N are shown in Table 6.16.

We start our analysis of the model by means of transient analysis. Let us check the probability that station 1 is awaiting service at time point $\tau = 10$ with the following formula

$$\mathcal{P}_{=?} [true \ U^{\tau, \tau} [si_1 = 1 \wedge \neg[s = 1 \wedge a = 1]]].$$

Table 6.17 shows the expected probability \Pr computed by the numerical en-

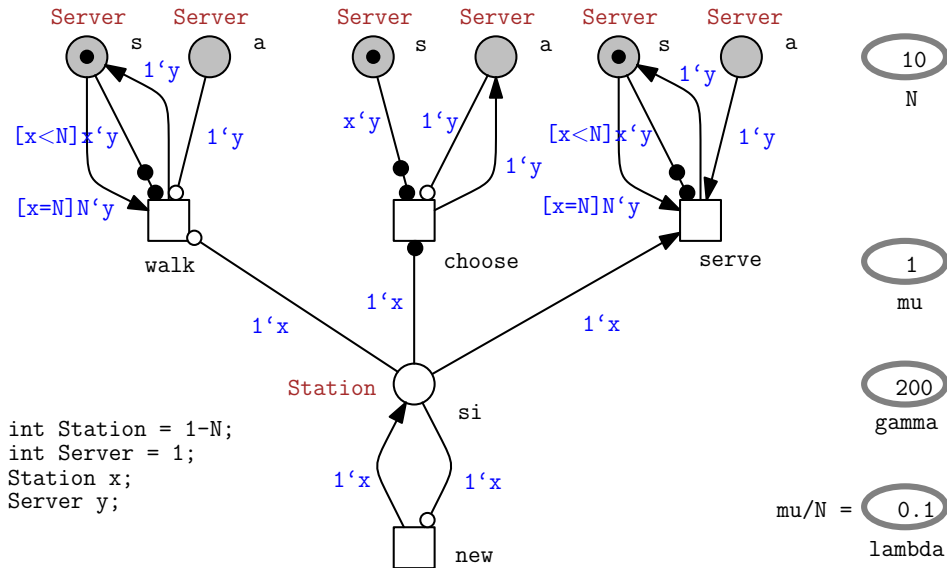


Figure 6.23: Coloured Stochastic Petri Net of the Cyclic Server Polling System.

Table 6.16: The size of the state space for different number of stations N of \mathcal{SPN}^c_{CSPS} computed with MARCIE's symbolic state space generation.

N	states	N	states	N	states	N	states
5	240	20	31,457,280	50	8.444×10^{16}	80	1.451×10^{26}
10	15,360	30	4.832×10^{10}	60	1.038×10^{20}	90	1.671×10^{29}
15	737,280	40	6.597×10^{13}	70	1.240×10^{23}	100	1.901×10^{32}

Table 6.17: Transient analysis up to time point $\tau = 10$ for different number of stations N of \mathcal{SPN}^c_{CSPS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.

N	Pr	CI
5	0.14198	[0.14177, 0.14246]
10	0.12294	[0.12252, 0.12318]
15	0.10152	[0.10118, 0.10179]
20	0.08560	[0.08516, 0.08572]
25	—	[0.07360, 0.07413]

gine and the confidence interval CI computed by the simulate engine after 6 634 234 simulation runs. The CI covers the expected value for all instances. The numerical engine is not able to compute the probability for $N = 25$ or greater, because of the state space explosion. Instead the simulative engine can evaluate the formula for even greater values than 25 just at the cost of larger run-time.

Figure 6.24 shows the run-time of the reachability analysis for different number of stations N and for several number of workers after 6 634 234 simulation runs. The run-time increases as the number of stations increases, which is to be expected, because of the increasing model size. With an increasing number of workers the run-time decreases in a close to linear way.

Next we compute the probability that in the long run station 1 is awaiting service with the following formula

$$\mathcal{S}_{=?} [si_1 = 1 \wedge \neg[s = 1 \wedge a = 1]] \text{ .}$$

The expected probability Pr in the long run computed by the numerical engine

Table 6.18: Steady state analysis for different number of stations N of \mathcal{SPN}_{CSPS}^c . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 128 simulation runs.

N	Pr	CI
5	0.14492	[0.13930, 0.15171]
10	0.14021	[0.13828, 0.14123]
15	0.13073	[0.12952, 0.13109]
20	0.12266	[0.12167, 0.12274]
25	—	[0.11630, 0.11712]

is covered well by the confidence interval computed by the simulate engine after 128 simulation runs, see Table 6.18.

The run-time of the steady state analysis for different number of stations N is increasing per N and decreases for several number of workers, see Figure 6.25.

Now we want to know the expected time station 1 is waiting to be served up to time point $\tau = 10$. Therefore, we use the following state reward function

```
rewards [waiting] { si_1=1 & ! [s_1=1 & a_1=1] : 1; }
```

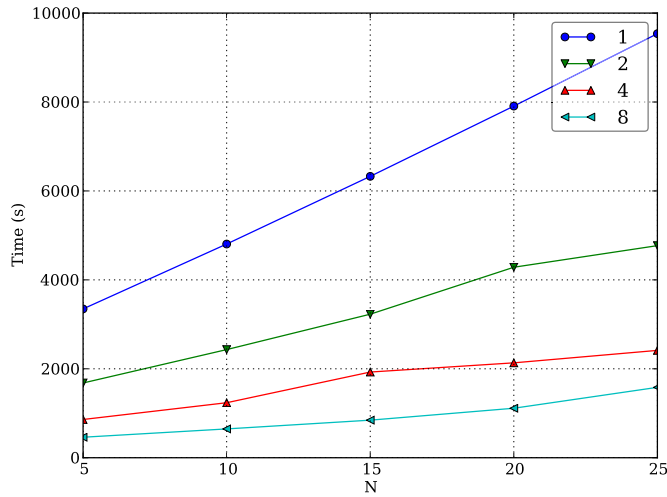


Figure 6.24: Transient analysis up to time point $\tau = 10$ for different number of stations N of \mathcal{SPN}_{CSPS}^c . The total run-time is given for several number of workers after 6 634 234 simulation runs.

Table 6.19: Reward analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The expected reward value R is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.

N	R	CI
5	1.25457	[1.25410, 1.25659]
10	0.65473	[0.65398, 0.65493]
15	0.44157	[0.44110, 0.44199]
20	0.33245	[0.33194, 0.33288]
25	—	[0.26601, 0.26690]

to compute the accumulated reward with the following formula

$$\mathcal{R}_{=?} [C^{\leq 1}] .$$

As in the previous computations, the confidence interval of the simulative algorithm covers the expected reward value.

Figure 6.26 shows the run-times of the reward analysis for different number of stations N and for several number of workers after 6 634 234 simulation runs. The developing of the run-times of the reward analysis look the same as in the transient analysis, but they are about 20% higher.

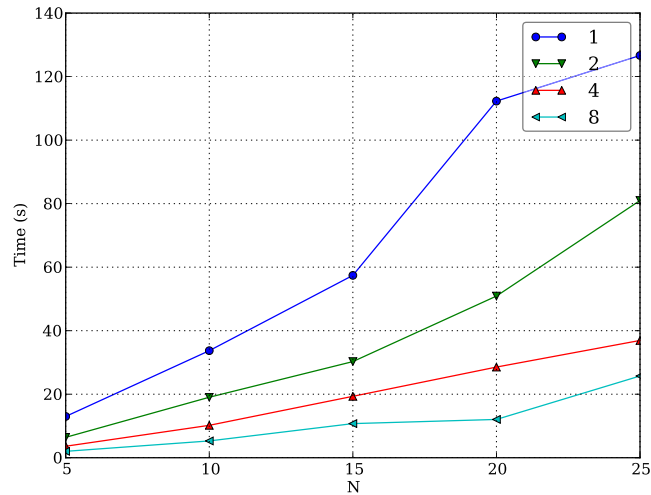


Figure 6.25: Steady state analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The total run-time is given for different numbers of workers after 128 simulation runs.

Table 6.20: Performability analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The probability Pr is computed by the numerical engine and the confidence interval CI by the simulative engine after 6 634 234 simulation runs.

N	Pr	CI
5	8.447×10^{-2}	$[8.419 \times 10^{-2}, 8.475 \times 10^{-2}]$
10	9.849×10^{-2}	$[9.819 \times 10^{-2}, 9.879 \times 10^{-2}]$
15	8.844×10^{-2}	$[8.815 \times 10^{-2}, 8.872 \times 10^{-2}]$
20	7.748×10^{-2}	$[7.721 \times 10^{-2}, 7.775 \times 10^{-2}]$
25	—	$[6.812 \times 10^{-2}, 6.863 \times 10^{-2}]$

And last but not least, we conduct the performability analysis that station 1 is awaiting service at time point $\tau = 10$ and with an accumulated reward of at most 1. This is done by evaluating the following CSRL formula

$$\mathcal{P}_{=?} \left[\text{true } U_{0,y}^{\tau,\tau} [si_1 = 1 \wedge \neg[s = 1 \wedge a = 1]] \right] .$$

Performability analysis is literally a combination of transient analysis and reward analysis. Thus it is no surprise that the numerical engine's expected probability is covered by the confidence interval, see Table 6.20.

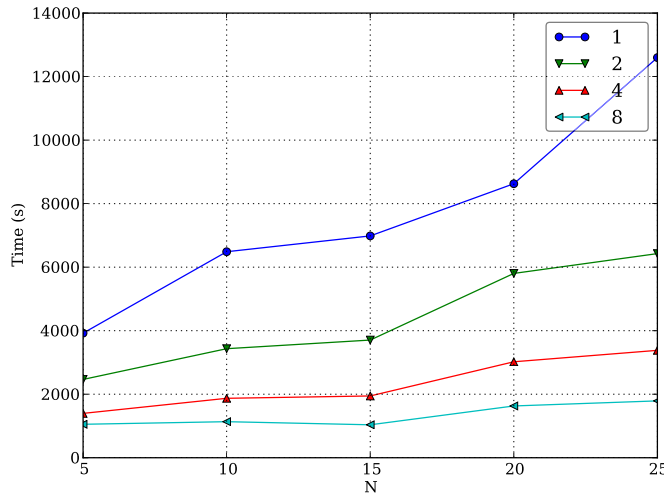


Figure 6.26: Reward analysis for different number of stations N of \mathcal{SPN}^c_{CSPS} . The total run-time is given for different numbers of workers after 6 634 234 simulation runs.

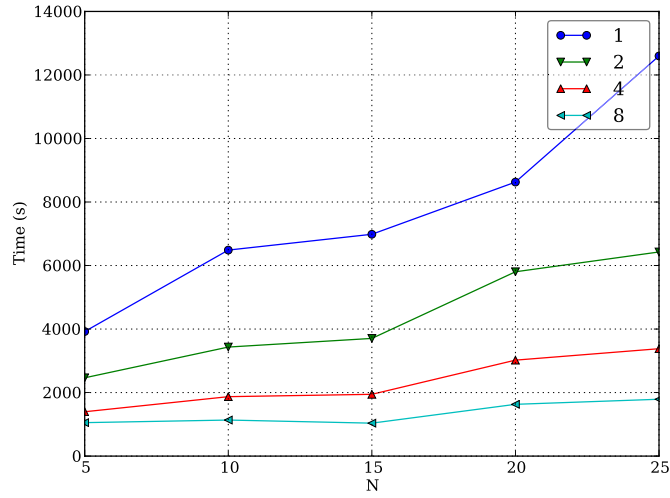


Figure 6.27: Performability analysis for different number of stations N of \mathcal{SPN}_{CSPS}^c . The total run-time is given for different numbers of workers after 6 634 234 simulation runs.

The run-times of the performability analysis is shown in Figure 6.27. They are similar to the reward analysis for all number of stations N and for all number of workers. That means the simulative CSRL model checking has only little overhead compared to simulative CSL model checking.

6.8 Closing Remarks

In this chapter we used five biochemical case studies and two technical systems in order to demonstrate the capabilities of simulative analysis and simulative model checking. We illustrated that the discrete-time leap method computes reasonable results and has a very good run-time performance especially for larger and dense networks. It is less sensitive to higher number of tokens and thus higher transition rates than stochastic simulation algorithms in terms of run-time. This recommends δ -leaping for models, which stochastic simulation is not capable of simulating in reasonable time, e.g., genome scale metabolic models.

We showed the verification of time-bounded and time-unbounded PLTLc formulas including transient and steady state analysis. This was done for CSL formulas as well, and in addition we exemplified reward and performability

analysis. The simulation, analysis and model checking algorithms are parallelised in at least one out of two possible ways: first in terms of shared memory and multi-threading, and second for distributed memory and multi-processing. We demonstrated the scalability of both implementations, and it turned out that both scale very well with the number of workers most of the time.

But there is one caveat that needs further investigation, the simulative steady state analysis of \mathcal{GSPN} models. Whereas in all other cases the computed confidence intervals covered the expected values, they did not cover the expected values in two instances of the flexible manufacturing system.

Beyond the given seven case studies, there are a couple of published non-trivial case studies, some of them deploying coloured \mathcal{XSPN} , which were made possible by the advanced simulation and analysis features efficiently supported by Marcie, among them [Pâr+15; BR15; Blä+14; LH14; Liu+14; LHY14; LH13a; LH13b; Blä+13; Gao+13; Gil+13; Gao+11].

Chapter 7

Conclusions and Outlook

7.1 Conclusions

In this thesis we introduced the discrete-time leap method for the simulation of stochastic Petri nets. It converts the underlying CTMC into a stochastically equivalent DTMC. Generating paths through the DTMC is as expensive as for the CTMC and we would not gain any efficiency by doing it in an exact way. That is why we are leaping over several states. The discrete time model and the maximum firing rule in combination with binomial sampling and weighted random shuffling of the transitions make an efficient simulation algorithm, that leads to comparable results. Furthermore, we presented exact and approximate stochastic simulation algorithms that are state-of-the-art, and performed an exemplary comparison of δ -leaping and direct method.

We can conclude that the discrete-time leap method computes reasonable results and has a very good run-time performance especially for larger and dense networks. It is less sensitive to higher number of tokens and thus higher transition rates than other stochastic simulation algorithms in terms of run-time. This recommends δ -leaping for models, which can not be simulated in reasonable time, e.g., genome scale metabolic models. Furthermore, it is suitable for in silico experiments, where the behavioural differences between modified models are of interest, such as knock-out scenarios of certain species or reactions. The approximation error is moderate as long as the condition in Equation 3.14 is fulfilled. There is surely space for discussion on how significant the reported approximation error is and the interpretation may be subjective, but a rela-

tive error below 0.05 can be seen as sufficiently good. A larger approximation error may be an indication for one the following situations. First, the model's time-scale is smaller than the chosen δ , i.e., reducing the δ would gain better approximation. Second, some transition's rate functions are not scaled correctly, i.e., stochastic reaction rates have to be scaled with respect to their reaction order.

For particular models it might be necessary to adapt the kinetic rate constants in order to obtain similar results, see Section 3.7.4. Here, further investigations are needed whether we can compute the required adaptations from the net structure. The discrete-time leap method is implemented and publicly available in our tools Snoopy [Hei+12] and MARCIE [HRS13].

Moreover, we presented simulative analysis of stochastic Petri nets and we showed that simulative analysis is not restricted to trace generation. We are able to compute approximations of transient solutions and steady state distributions. In case of transient solutions, we introduced some optimizations to make the computation more efficient. The computation of derived measures (observers) paved us a way to a whole new class of models, namely Markov reward models.

A more advanced way for the verification of system properties is simulative model checking. We presented an infinite time horizon model checking algorithm plus steady state operator for probabilistic linear-time temporal logic. In addition, we gave simulative model checking algorithms of continuous stochastic logic formulas including reward extensions and time-unbounded temporal operators. In Section 5.2, we stated that simulative CSL model checking reduces the memory consumption to some constant value compared to simulative PLTLc model checking where it increases with the length of the generated trace. In Figure 7.1 we compare the peak memory consumption of simulative CSL and PLTLc model checking up to different time points τ and it confirms the previous statement. The memory consumption of simulative CSL model checking remains constant throughout all time points, whereas the memory consumption of simulative PLTLc model checking increases with τ , because the length of the generated trace increases. The increase is not linear, because of some storage optimizations, i.e., not the full trace is stored, but only the number of tokens of places used in the formula and successive equal numbers

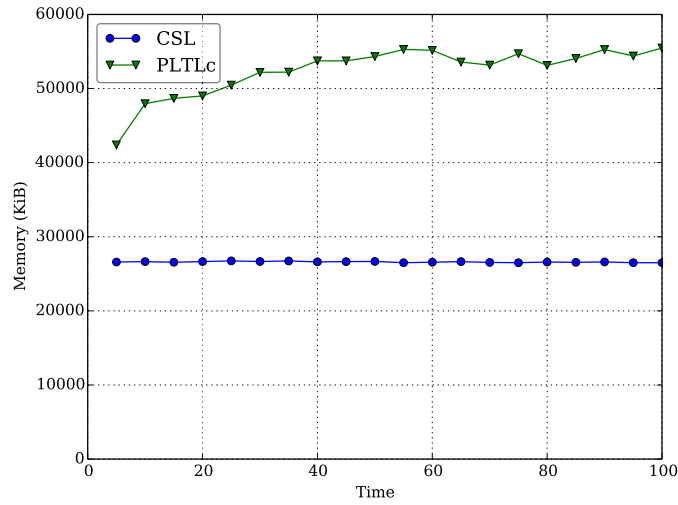


Figure 7.1: The peak memory consumption is given for transient analysis up to different time points τ for $N = 10$ stations of \mathcal{SPN}^c_{CSPS} using simulative CSL and PLTLc model checking.

are compressed.

To the best of our knowledge, we introduced the first simulative continuous stochastic reward logic model checking algorithm. We used five biochemical case studies and two technical systems in order to demonstrate the capabilities of simulative analysis and simulative model checking. We verified the results of the simulative approach against the numerical solutions of the Jacobi and Gauss-Seidel methods. We proved the efficiency of our algorithm and the scalability by using several worker threads in the shared memory and multi-threading implementation, and by using several worker processes in the distributed memory and multi-processing implementation. It turned out that both implementations scale very well with the number of workers most of the time.

As our algorithms are based on stochastic simulation, their run-time does not directly depend on the size of the state space, as for the numerical methods, but on the rate functions of the transitions and the size of the Petri net, i.e., number of places, transitions and arcs. The greater the sum of the transitions rates, the smaller the time steps are, and the more simulation steps need to be done to reach a certain time point. Thus, the main drawback of simulation-based methods remains. The achieved accuracy depends on the number of

simulation runs and the number of required runs grows exponentially with the expected accuracy. Therefore, methods of choice for bounded and medium-sized models are numerical, otherwise simulation plays out its strength.

7.2 Outlook

As in any other case, there is space for improvements. On short term, the depicted caveats in the discrete-time leap method and in the steady state analysis of \mathcal{GSPN} models are in need of investigations. The weighted random shuffle of δ -leaping needs further improvements to obtain even better approximations of the stochastic simulation. Further investigation is in need for the caveat found in Section 3.7.4. One question is, are there any other situations that behave in the same way, and if so, how can we find them in larger models. It is necessary to strengthen the approximation quality of δ -leaping by applying additional quantitative comparison methods like linear regression or ANOVA [Fis92], and Tukey’s range test [Tuk49] as well as cross-validation. Another aspect that deserves attention relates to the circumstances under which the approximation accuracy suffers, e.g., at which points in a simulation trace one would expect significant differences. Out of the experience so far with delta-leaping, one could expect to get such significant differences in conflict situations (see Section 3.7.2) and in situations with very high transition rates (very stiff reactions). Such situations could be identified with a comparison of the rate of change in the number of tokens of affected places and the relative error, over time. A closer look at the steady state analysis of \mathcal{GSPN} models is necessary to resolve the caveat found in Section 6.6, whether it is a model, an implementation or a methodical issue.

On long term, there is a broad range of possibilities to develop and implement simulative analysis of stochastic Petri nets. General purpose graphics processing units (GPGPUs) offer a new way of parallel computation and show great potential to speed-up simulative analysis. In case of stochastic simulation there are already great improvements if it comes to the parallelisation of several simulation runs [LP07], i.e., speed-up 150 – 170 times compared to single-threaded CPU, but the parallelisation of an individual simulation run is still an open issue, the current speed-up is not even 2 times [DC09]. Our

δ -leaping method opened up a new class of models for simulative analysis, i.e., genome scale metabolic models, but there is still work to do, e.g., automatic δ adaptation.

The presented computation of transient solutions can be adapted to simulate rare events without any additional effort. Therefore, we would perform $\lceil N \cdot |\pi(\tau_j)|^{-1} \rceil$ simulation runs for any visited state. The probability of the reached state at τ_{j+1} would be then increased by the quotient $\pi_i(\tau_j) / \lceil N \cdot |\pi(\tau_j)|^{-1} \rceil$ of the probability of the starting state i and the number of simulation runs, calculated before. We can adapt Equation 4.21 and the probability to be in state i at the end of the time interval (τ_j, τ_{j+1}) would be approximately

$$\pi_i(\tau_{j+1}) \approx \sum_{k=1}^{|\pi(\tau_j)|} \sum_{n=1}^{\lceil N \cdot |\pi(\tau_j)|^{-1} \rceil} \begin{cases} \pi_k(\tau_j) / \lceil N \cdot |\pi(\tau_j)|^{-1} \rceil & X(\tau_{j+1}) = i \\ 0 & otherwise \end{cases}.$$

But there is a drawback. The higher the probability of a state, the greater the discrepancy of the expected value. Thus, it would be of use only if one is interested in the rare events and not in the states with higher probability.

The support of rewards is not yet included in PLTLc, but adding it would close the gap to CSRL. Probably the main issue of simulative CS(R)L model checking is nesting of the probability operator. As we have stated in Section 5.2.1, it is possible, but not in an efficient way. One possible way to lessen the effort would be to distribute the verification of the inner formulas, but this is only a drop in the bucket.

Appendix A

Appendix

We give the syntax of all case studies used in Chapter 6 except for Section 6.5, because these are still under investigation. We use the abstract net definition language (ANDL) [SRH14] for all uncoloured Petri nets, e.g., RKIP inhibited ERK pathway, Mitogen-activated Protein Kinase and Angiogenesis. We use the coloured abstract net definition language (CANDL) [LHR12] for all coloured Petri nets, e.g., Repressilator, Flexible Manufacturing System and Cyclic Server Polling System.

A.1 ANDL Syntax of RKIP inhibited ERK pathway

```
1 spn [erk_N]
2 {
3 constants:
4 parameter:
5     double c1 = 0.53;
6     double c2 = 0.0072;
7     double c3 = 0.625;
8     double c4 = 0.00245;
9     double c5 = 0.0315;
10    double c6 = 0.8;
11    double c7 = 0.0075;
12    double c8 = 0.071;
13    double c9 = 0.92;
14    double c10 = 0.00122;
15    double c11 = 0.87;
16    double fs = 2.5;
17 marking:
18     int N = 1;
```

```

19
20 places:
21   Raf1Star = N;
22   RKIP = N;
23   Raf1Star_RKIP = 0;
24   ERKPP = 0;
25   MEKPP_ERK = 0;
26   Raf1Star_RKIP_ERKPP = 0;
27   RKIPP_RP = 0;
28   MEKPP = N;
29   ERK = N;
30   RKIPP = 0;
31   RP = N;
32
33 transitions:
34   r1
35   :
36   : [Raf1Star_RKIP + 1] & [Raf1Star - 1] & [RKIP - 1]
37   : MassAction( c1*fs/N )
38   ;
39   r2
40   :
41   : [Raf1Star + 1] & [RKIP + 1] & [Raf1Star_RKIP - 1]
42   : MassAction( c2 )
43   ;
44   r3
45   :
46   : [Raf1Star_RKIP_ERKPP + 1] & [Raf1Star_RKIP - 1] & [ERKPP - 1]
47   : MassAction( c3*fs/N )
48   ;
49   r4
50   :
51   : [Raf1Star_RKIP + 1] & [ERKPP + 1] & [Raf1Star_RKIP_ERKPP - 1]
52   : MassAction( c4 )
53   ;
54   r5
55   :
56   : [ERK + 1] & [RKIPP + 1] & [Raf1Star + 1] & [Raf1Star_RKIP_ERKPP - 1]
57   : MassAction( c5 )
58   ;
59   r6
60   :
61   : [MEKPP_ERK + 1] & [MEKPP - 1] & [ERK - 1]
62   : MassAction( c6*fs/N )
63   ;
64   r7
65   :
66   : [ERK + 1] & [MEKPP + 1] & [MEKPP_ERK - 1]
67   : MassAction( c7 )
68   ;
69   r8
70   :
71   : [ERKPP + 1] & [MEKPP + 1] & [MEKPP_ERK - 1]

```

```

72 : MassAction( c8 )
73 ;
74 r9
75 :
76 : [RKIPP_RP + 1] & [RP - 1] & [RKIPP - 1]
77 : MassAction( c9*fs/N )
78 ;
79 r10
80 :
81 : [RP + 1] & [RKIPP + 1] & [RKIPP_RP - 1]
82 : MassAction( c10 )
83 ;
84 r11
85 :
86 : [RKIP + 1] & [RP + 1] & [RKIPP_RP - 1]
87 : MassAction( c11 )
88 ;
89 }

```

A.2 ANDL Syntax of Mitogen-activated Protein Kinase

```

1 spn [levchenko_N]
2 {
3 constants:
4 parameter:
5 double k1 = 1.0;
6 double k10 = 3.3;
7 double k4 = 0.5;
8 double k3 = 0.1;
9 double k6 = 0.1;
10 double k7 = 3.3;
11 double k8 = 0.42;
12 double k9 = 0.1;
13 double k2 = 0.4;
14 double k21 = 0.1;
15 double k22 = 20.0;
16 double k23 = 0.6;
17 double k24 = 0.1;
18 double k25 = 5.0;
19 double k26 = 0.4;
20 double k27 = 0.1;
21 double k28 = 5.0;
22 double k29 = 0.4;
23 double k30 = 0.1;
24 double k11 = 0.4;
25 double k12 = 0.1;
26 double k13 = 10.0;
27 double k14 = 0.8;
28 double k15 = 0.1;

```

```

29     double k16 = 10.0;
30     double k17 = 0.8;
31     double k18 = 0.1;
32     double k19 = 20.0;
33     double k20 = 0.6;
34     double sf = 0.1;
35     double k5 = 0.5;
36 marking:
37     int N = 1;
38
39 places:
40     Raf = N*4;
41     RasGTP = N;
42     Raf_RasGTP = 0;
43     RafP = 0;
44     RafP_Phase1 = 0;
45     MEK_RafP = 0;
46     MEKP_RafP = 0;
47     MEKP_Phase2 = 0;
48     MEKPP_Phase2 = 0;
49     ERK = N*3;
50     ERK_MEKPP = 0;
51     ERKP_MEKPP = 0;
52     ERKP = 0;
53     MEKPP = 0;
54     ERKPP_Phase3 = 0;
55     ERKP_Phase3 = 0;
56     MEKP = 0;
57     ERKPP = 0;
58     Phase2 = N*2;
59     Phase3 = N*3;
60     MEK = N*2;
61     Phase1 = N*3;
62
63 transitions:
64     k3
65     :
66     : [RafP + 1] & [RasGTP + 1] & [Raf_RasGTP - 1]
67     : MassAction(k3)
68     ;
69     k6
70     :
71     : [Raf + 1] & [Phase1 + 1] & [RafP_Phase1 - 1]
72     : MassAction(k6)
73     ;
74     k21
75     :
76     : [ERKP + 1] & [MEKPP + 1] & [ERK_MEKPP - 1]
77     : MassAction(k21)
78     ;
79     k18
80     :
81     : [MEK + 1] & [Phase2 + 1] & [MEKP_Phase2 - 1]

```

```

82      : MassAction(k18)
83      ;
84      k9
85      :
86      : [MEKP + 1] & [RafP + 1] & [MEK_RafP - 1]
87      : MassAction(k9)
88      ;
89      k12
90      :
91      : [MEKPP + 1] & [RafP + 1] & [MEKP_RafP - 1]
92      : MassAction(k12)
93      ;
94      k15
95      :
96      : [Phase2 + 1] & [MEKP + 1] & [MEKPP_Phase2 - 1]
97      : MassAction(k15)
98      ;
99      k24
100     :
101     : [ERKPP + 1] & [MEKPP + 1] & [ERKP_MEKPP - 1]
102     : MassAction(k24)
103     ;
104     k1
105     :
106     : [Raf_RasGTP + 1] & [Raf - 1] & [RasGTP - 1]
107     : MassAction(k1*sf/N)
108     ;
109     k2
110     :
111     : [RasGTP + 1] & [Raf + 1] & [Raf_RasGTP - 1]
112     : MassAction(k2)
113     ;
114     k27
115     :
116     : [Phase3 + 1] & [ERKP + 1] & [ERKPP_Phase3 - 1]
117     : MassAction(k27)
118     ;
119     k30
120     :
121     : [ERK + 1] & [Phase3 + 1] & [ERKP_Phase3 - 1]
122     : MassAction(k30)
123     ;
124     k16
125     :
126     : [MEKP_Phase2 + 1] & [Phase2 - 1] & [MEKP - 1]
127     : MassAction(k16*sf/N)
128     ;
129     k17
130     :
131     : [Phase2 + 1] & [MEKP + 1] & [MEKP_Phase2 - 1]
132     : MassAction(k17)
133     ;
134     k29

```

```

135      :
136      : [Phase3 + 1] & [ERKP + 1] & [ERKP_Phase3 - 1]
137      : MassAction(k29)
138      ;
139      k28
140      :
141      : [ERKP_Phase3 + 1] & [Phase3 - 1] & [ERKP - 1]
142      : MassAction(k28*sf/N)
143      ;
144      k14
145      :
146      : [Phase2 + 1] & [MEKPP + 1] & [MEKPP_Phase2 - 1]
147      : MassAction(k14)
148      ;
149      k13
150      :
151      : [MEKPP_Phase2 + 1] & [Phase2 - 1] & [MEKPP - 1]
152      : MassAction(k13*sf/N)
153      ;
154      k5
155      :
156      : [Phase1 + 1] & [RafP + 1] & [RafP_Phase1 - 1]
157      : MassAction(k5)
158      ;
159      k4
160      :
161      : [RafP_Phase1 + 1] & [Phase1 - 1] & [RafP - 1]
162      : MassAction(k4*sf/N)
163      ;
164      k26
165      :
166      : [Phase3 + 1] & [ERKPP + 1] & [ERKPP_Phase3 - 1]
167      : MassAction(k26)
168      ;
169      k25
170      :
171      : [ERKPP_Phase3 + 1] & [Phase3 - 1] & [ERKPP - 1]
172      : MassAction(k25*sf/N)
173      ;
174      k11
175      :
176      : [RafP + 1] & [MEKP + 1] & [MEKP_RafP - 1]
177      : MassAction(k11)
178      ;
179      k8
180      :
181      : [RafP + 1] & [MEK + 1] & [MEK_RafP - 1]
182      : MassAction(k8)
183      ;
184      k23
185      :
186      : [MEKPP + 1] & [ERKP + 1] & [ERKP_MEKPP - 1]
187      : MassAction(k23)

```



```

188     ;
189     k20
190     :
191     : [MEKPP + 1] & [ERK + 1] & [ERK_MEKPP - 1]
192     : MassAction(k20)
193     ;
194     k7
195     :
196     : [MEK_RafP + 1] & [RafP - 1] & [MEK - 1]
197     : MassAction(k7*sf/N)
198     ;
199     k10
200     :
201     : [MEKP_RafP + 1] & [RafP - 1] & [MEKP - 1]
202     : MassAction(k10*sf/N)
203     ;
204     k19
205     :
206     : [ERK_MEKPP + 1] & [MEKPP - 1] & [ERK - 1]
207     : MassAction(k19*sf/N)
208     ;
209     k22
210     :
211     : [ERKP_MEKPP + 1] & [MEKPP - 1] & [ERKP - 1]
212     : MassAction(k22*sf/N)
213     ;
214 }

```

A.3 ANDL Syntax of Angiogenesis

```

1  spn [angiogenesis_N]
2  {
3  constants:
4  parameter:
5      double Receptor = 1;
6      double Survival = 10;
7      double Regeneration = 1;
8      double Proliferation = 1;
9  all:
10     int N = 1;
11
12  places:
13     Akt = N;
14     AktP3 = 0;
15     AktStar = 0;
16     DAG = N;
17     DAGE = 0;
18     Enz = N;
19     Gab1 = N;
20     GP3 = 0;
21     GStarP3 = 0;
22     GStarP3kP3 = 0;

```

```

23     GStarPgP3 = 0;
24     KdStar = N;
25     KdStarG = 0;
26     KdStarGP3 = 0;
27     KdStarGStar = 0;
28     KdStarGStarP3 = 0;
29     KdStarGStarP3k = 0;
30     KdStarGStarP3kP3 = 0;
31     KdStarGStarP3kStar = 0;
32     KdStarGStarP3kStarP2 = 0;
33     KdStarGStarP3kStarP3 = 0;
34     KdStarGStarP3kStarP3P2 = 0;
35     KdStarGStarPg = 0;
36     KdStarGStarPgP3 = 0;
37     KdStarGStarPgStar = 0;
38     KdStarGStarPgStarP2 = 0;
39     KdStarGStarPgStarP3 = 0;
40     KdStarGStarPgStarP3P2 = 0;
41     KdStarPg = 0;
42     KdStarPgStar = 0;
43     KdStarPgStarP2 = 0;
44     Pip2 = N;
45     Pip3 = 0;
46     P3k = N;
47     Pg = N;
48     Pten = N;
49     PtP2 = 0;
50     PtP3 = 0;
51     PtP3P2 = 0;
52
53 transitions:
54     k0
55         :
56         : [KdStarG + 1] & [Gab1 - 1] & [KdStar - 1]
57         : MassAction( Receptor )
58         ;
59     k1
60         :
61         : [Gab1 + 1] & [KdStar + 1] & [KdStarG - 1]
62         : MassAction( Receptor )
63         ;
64     k10
65         :
66         : [KdStarGStarP3 + 1] & [KdStarGStar - 1] & [Pip3 - 1]
67         : MassAction( Receptor )
68         ;
69     k11
70         :
71         : [KdStarGStar + 1] & [Pip3 + 1] & [KdStarGStarP3 - 1]
72         : MassAction( Receptor )
73         ;
74     k12
75         :

```

```

76      : [GStarP3kP3 + 1] & [GStarP3 - 1] & [P3k - 1]
77      : MassAction( Survival )
78      ;
79      k13
80      :
81      : [GStarP3 + 1] & [P3k + 1] & [GStarP3kP3 - 1]
82      : MassAction( Survival )
83      ;
84      k14
85      :
86      : [KdStarGStarP3kP3 + 1] & [GStarP3kP3 - 1] & [KdStar - 1]
87      : MassAction( Survival )
88      ;
89      k15
90      :
91      : [GStarP3kP3 + 1] & [KdStar + 1] & [KdStarGStarP3kP3 - 1]
92      : MassAction( Survival )
93      ;
94      k16
95      :
96      : [KdStarGStarP3k + 1] & [KdStarGStar - 1] & [P3k - 1]
97      : MassAction( Survival )
98      ;
99      k17
100     :
101     : [KdStarGStar + 1] & [P3k + 1] & [KdStarGStarP3k - 1]
102     : MassAction( Survival )
103     ;
104     k18
105     :
106     : [KdStarGStarP3kStar + 1] & [KdStarGStarP3k - 1]
107     : MassAction( Survival )
108     ;
109     k19
110     :
111     : [KdStarGStarP3kStarP2 + 1] & [KdStarGStarP3kStar - 1] & [Pip2 - 1]
112     : MassAction( Survival )
113     ;
114     k2
115     :
116     : [KdStarGStar + 1] & [KdStarG - 1]
117     : MassAction( Receptor )
118     ;
119     k20
120     :
121     : [KdStarGStarP3kStar + 1] & [Pip2 + 1] & [KdStarGStarP3kStarP2 - 1]
122     : MassAction( Survival )
123     ;
124     k21
125     :
126     : [KdStarGStarP3k + 1] & [Pip3 + 1] & [KdStarGStarP3kStarP2 - 1]
127     : MassAction( Survival )
128     ;

```

```

129     k22
130     :
131     : [KdStarGStarP3kP3 + 1] & [KdStarGStarP3 - 1] & [P3k - 1]
132     : MassAction( Survival )
133     ;
134     k23
135     :
136     : [KdStarGStarP3 + 1] & [P3k + 1] & [KdStarGStarP3kP3 - 1]
137     : MassAction( Survival )
138     ;
139     k24
140     :
141     : [KdStarGStarP3kStarP3 + 1] & [KdStarGStarP3kP3 - 1]
142     : MassAction( Survival )
143     ;
144     k25
145     :
146     : [KdStarGStarP3kStarP3P2 + 1] & [KdStarGStarP3kStarP3 - 1] & [Pip2 - 1]
147     : MassAction( Survival )
148     ;
149     k26
150     :
151     : [KdStarGStarP3kStarP3 + 1] & [Pip2 + 1] & [KdStarGStarP3kStarP3P2 - 1]
152     : MassAction( Survival )
153     ;
154     k27
155     :
156     : [KdStarGStarP3kP3 + 1] & [Pip3 + 1] & [KdStarGStarP3kStarP3P2 - 1]
157     : MassAction( Survival )
158     ;
159     k28
160     :
161     : [AktP3 + 1] & [Pip3 - 1] & [Akt - 1]
162     : MassAction( Survival )
163     ;
164     k29
165     :
166     : [Pip3 + 1] & [Akt + 1] & [AktP3 - 1]
167     : MassAction( Survival )
168     ;
169     k3
170     :
171     : [GP3 + 1] & [Gab1 - 1] & [Pip3 - 1]
172     : MassAction( Receptor )
173     ;
174     k30
175     :
176     : [AktStar + 1] & [Pip3 + 1] & [AktP3 - 1]
177     : MassAction( Survival )
178     ;
179     k31
180     :
181     : [KdStarPg + 1] & [KdStar - 1] & [Pg - 1]

```

```

182      : MassAction( Proliferation )
183      ;
184    k32
185      :
186      : [KdStar + 1] & [Pg + 1] & [KdStarPg - 1]
187      : MassAction( Proliferation )
188      ;
189    k33
190      :
191      : [KdStarPgStar + 1] & [KdStarPg - 1]
192      : MassAction( Proliferation )
193      ;
194    k34
195      :
196      : [KdStarPgStarP2 + 1] & [KdStarPgStar - 1] & [Pip2 - 1]
197      : MassAction( Proliferation )
198      ;
199    k35
200      :
201      : [KdStarPgStar + 1] & [Pip2 + 1] & [KdStarPgStarP2 - 1]
202      : MassAction( Proliferation )
203      ;
204    k36
205      :
206      : [KdStarPg + 1] & [DAG + 1] & [KdStarPgStarP2 - 1]
207      : MassAction( Proliferation )
208      ;
209    k37
210      :
211      : [KdStarGStarPg + 1] & [KdStarGStar - 1] & [Pg - 1]
212      : MassAction( Proliferation )
213      ;
214    k38
215      :
216      : [KdStarGStar + 1] & [Pg + 1] & [KdStarGStarPg - 1]
217      : MassAction( Proliferation )
218      ;
219    k39
220      :
221      : [KdStarGStarPgStar + 1] & [KdStarGStarPg - 1]
222      : MassAction( Proliferation )
223      ;
224    k4
225      :
226      : [Gab1 + 1] & [Pip3 + 1] & [GP3 - 1]
227      : MassAction( Receptor )
228      ;
229    k40
230      :
231      : [KdStarGStarPgStarP2 + 1] & [KdStarGStarPgStar - 1] & [Pip2 - 1]
232      : MassAction( Proliferation )
233      ;
234    k41

```

```

235      :
236      : [KdStarGStarPgStar + 1] & [Pip2 + 1] & [KdStarGStarPgStarP2 - 1]
237      : MassAction( Proliferation )
238      ;
239  k42
240      :
241      : [KdStarGStarPg + 1] & [DAG + 1] & [KdStarGStarPgStarP2 - 1]
242      : MassAction( Proliferation )
243      ;
244  k43
245      :
246      : [KdStarGStarPgP3 + 1] & [KdStarGStarP3 - 1] & [Pg - 1]
247      : MassAction( Proliferation )
248      ;
249  k44
250      :
251      : [KdStarGStarP3 + 1] & [Pg + 1] & [KdStarGStarPgP3 - 1]
252      : MassAction( Proliferation )
253      ;
254  k45
255      :
256      : [KdStarGStarPgStarP3 + 1] & [KdStarGStarPgP3 - 1]
257      : MassAction( Proliferation )
258      ;
259  k46
260      :
261      : [KdStarGStarPgStarP3P2 + 1] & [KdStarGStarPgStarP3 - 1] & [Pip2 - 1]
262      : MassAction( Proliferation )
263      ;
264  k47
265      :
266      : [KdStarGStarPgStarP3 + 1] & [Pip2 + 1] & [KdStarGStarPgStarP3P2 - 1]
267      : MassAction( Proliferation )
268      ;
269  k48
270      :
271      : [KdStarGStarPgP3 + 1] & [DAG + 1] & [KdStarGStarPgStarP3P2 - 1]
272      : MassAction( Proliferation )
273      ;
274  k49
275      :
276      : [GStarPgP3 + 1] & [GStarP3 - 1] & [Pg - 1]
277      : MassAction( Proliferation )
278      ;
279  k5
280      :
281      : [KdStarGP3 + 1] & [GP3 - 1] & [KdStar - 1]
282      : MassAction( Receptor )
283      ;
284  k50
285      :
286      : [GStarP3 + 1] & [Pg + 1] & [GStarPgP3 - 1]
287      : MassAction( Proliferation )

```

```

288     ;
289     k51
290     :
291     : [KdStarGStarPgP3 + 1] & [GStarPgP3 - 1] & [KdStar - 1]
292     : MassAction( Proliferation )
293     ;
294     k52
295     :
296     : [GStarPgP3 + 1] & [KdStar + 1] & [KdStarGStarPgP3 - 1]
297     : MassAction( Proliferation )
298     ;
299     k53
300     :
301     : [PtP3 + 1] & [Pip3 - 1] & [Pten - 1]
302     : MassAction( Regeneration )
303     ;
304     k54
305     :
306     : [Pip3 + 1] & [Pten + 1] & [PtP3 - 1]
307     : MassAction( Regeneration )
308     ;
309     k55
310     :
311     : [Pip2 + 1] & [Pten + 1] & [PtP3 - 1]
312     : MassAction( Regeneration )
313     ;
314     k56
315     :
316     : [PtP2 + 1] & [Pip2 - 1] & [Pten - 1]
317     : MassAction( Regeneration )
318     ;
319     k57
320     :
321     : [Pip2 + 1] & [Pten + 1] & [PtP2 - 1]
322     : MassAction( Regeneration )
323     ;
324     k58
325     :
326     : [PtP3P2 + 1] & [PtP2 - 1] & [Pip3 - 1]
327     : MassAction( Regeneration )
328     ;
329     k59
330     :
331     : [PtP2 + 1] & [Pip3 + 1] & [PtP3P2 - 1]
332     : MassAction( Regeneration )
333     ;
334     k6
335     :
336     : [GP3 + 1] & [KdStar + 1] & [KdStarGP3 - 1]
337     : MassAction( Receptor )
338     ;
339     k60
340     :

```

```

341      : [PtP2 + 1] & [Pip2 + 1] & [PtP3P2 - 1]
342      : MassAction( Regeneration )
343      ;
344  k61
345      :
346      : [DAGE + 1] & [DAG - 1] & [Enz - 1]
347      : MassAction( Regeneration )
348      ;
349  k62
350      :
351      : [DAG + 1] & [Enz + 1] & [DAGE - 1]
352      : MassAction( Regeneration )
353      ;
354  k63
355      :
356      : [Pip2 + 1] & [Enz + 1] & [DAGE - 1]
357      : MassAction( Regeneration )
358      ;
359  k7
360      :
361      : [KdStarGStarP3 + 1] & [KdStarGP3 - 1]
362      : MassAction( Receptor )
363      ;
364  k8
365      :
366      : [GStarP3 + 1] & [KdStar + 1] & [KdStarGStarP3 - 1]
367      : MassAction( Receptor )
368      ;
369  k9
370      :
371      : [KdStarGStarP3 + 1] & [GStarP3 - 1] & [KdStar - 1]
372      : MassAction( Receptor )
373      ;
374  }

```

A.4 CANDL Syntax of Repressilator

```

1  colspn [repressilator]
2  {
3  constants:
4  param:
5  double g = 0.05;
6  double d = 0.003;
7  double dr = 0.003;
8  double a0 = 0.5;
9  double a1 = 0.01;
10
11  colorsets:
12  genes = {1,2,3};
13
14  variables:
15  genes : x;

```



```

16
17 places:
18 discrete:
19 genes e = 1'all;
20 genes r = 0'all;
21 genes p = 0'all;
22
23 transitions:
24 t4
25 :
26 : [e + {x}] & [p + {-x}] & [r - {x}]
27 : a1*r
28 ;
29 t3
30 :
31 : [r + {x}] & [e - {x}] & [p - {-x}]
32 : a0*e*p
33 ;
34 t1
35 :
36 : [p + {x}] & [e + {x}] & [e - {x}]
37 : g*e
38 ;
39 t2
40 :
41 : [p - {x}]
42 : d*p
43 ;
44 t5
45 :
46 : [e + {x}] & [r - {x}]
47 : dr*r
48 ;
49
50 }

```

A.5 CANDL Syntax of Flexible Manufacturing System

```

1 colgspn [fms_gspn]
2 {
3 constants:
4 all:
5     int N = 10;
6     int np = 3*N/2;
7
8 colorsets:
9     Dot = {dot};
10    CS = {1..N};
11    Dummy = {1};

```

```

12
13 variables:
14     CS : x;
15
16 places:
17 discrete:
18     Dummy P1 = N'all;
19     Dummy P1wM1 = 0'all;
20     Dummy P1M1 = 0'all;
21     Dummy M1 = 3'all;
22     Dummy P1s = 0'all;
23     Dummy P12s = 0'all;
24     Dummy P12M3 = 0'all;
25     Dummy M3 = 2'all;
26     Dummy P12wM3 = 0'all;
27     Dummy P12 = 0'all;
28     Dummy P1wP2 = 0'all;
29     Dummy P2wP1 = 0'all;
30     Dummy P2 = N'all;
31     Dummy P2wM2 = 0'all;
32     Dummy P2M2 = 0'all;
33     Dummy M2 = 1'all;
34     Dummy P2s = 0'all;
35     Dummy P3 = N'all;
36     Dummy P3M2 = 0'all;
37     Dummy P3s = 0'all;
38     Dummy P1d = 0'all;
39     Dummy P2d = 0'all;
40
41 transitions:
42     tP3
43         : [P1 {1'1}] & [P2 {1'1}] & [P12 {1'1}]
44         : [P3M2 + {1'1}] & [P3 - {1'1}]
45         : P3*max(1,np/(P1+P2+P3+P12))
46         ;
47     tP3M2
48         :
49         : [M2 + {1'1}] & [P3s + {1'1}] & [M2 - {1'1}] & [P3M2 - {1'1}]
50         : 0.5
51         ;
52     tP1
53         : [P2 {1'1}] & [P12 {1'1}] & [P3 {1'1}]
54         : [P1wM1 + {1'1}] & [P1 - {1'1}]
55         : P1*max(1,np/(P1+P2+P3+P12))
56         ;
57     tP2
58         : [P1 {1'1}] & [P12 {1'1}] & [P3 {1'1}]
59         : [P2wM2 + {1'1}] & [P2 - {1'1}]
60         : P2*max(1,np/(P1+P2+P3+P12))
61         ;
62     tP12
63         : [P1 {1'1}] & [P2 {1'1}] & [P3 {1'1}]
64         : [P12wM3 + {1'1}] & [P12 - {1'1}]

```

```

65      : P12*max(1,np/(P1+P2+P3+P12))
66      ;
67      tP1M1
68      :
69      : [M1 + {1'1}] & [P1d + {1'1}] & [P1M1 - {1'1}]
70      : P1M1/4
71      ;
72      tP12M3
73      :
74      : [P12s + {1'1}] & [M3 + {1'1}] & [P12M3 - {1'1}]
75      : P12M3
76      ;
77      tP2M2
78      :
79      : [M2 + {1'1}] & [P2d + {1'1}] & [P2M2 - {1'1}]
80      : 1/6
81      ;
82      tP1s
83      : [P1s = {x'1}]
84      : [P1 + {x'1}] & [P1s - {x'1}]
85      : 1/60
86      ;
87      tP2s
88      : [P2s = {x'1}]
89      : [P2 + {x'1}] & [P2s - {x'1}]
90      : 1/60
91      ;
92      tP3s
93      : [P3s = {x'1}]
94      : [P3 + {x'1}] & [P3s - {x'1}]
95      : 1/60
96      ;
97      tP12s
98      : [P12s = {x'1}]
99      : [P1 + {x'1}] & [P2 + {x'1}] & [P12s - {x'1}]
100     : 1/60
101     ;
102     immediate:
103     tM1
104     :
105     : [P1M1 + {1'1}] & [P1wM1 - {1'1}] & [M1 - {1'1}]
106     : 1
107     ;
108     tP1e
109     :
110     : [P1s + {1'1}] & [P1d - {1'1}]
111     : 0.8
112     ;
113     tP1j
114     :
115     : [P1wP2 + {1'1}] & [P1d - {1'1}]
116     : 0.2
117     ;

```

```

118     tx
119     :
120     : [P12 + {1'1}] & [P1wP2 - {1'1}] & [P2wP1 - {1'1}]
121     : 1
122     ;
123     tM3
124     :
125     : [P12M3 + {1'1}] & [P12wM3 - {1'1}] & [M3 - {1'1}]
126     : 1
127     ;
128     tM2
129     :
130     : [P2M2 + {1'1}] & [P2wM2 - {1'1}] & [M2 - {1'1}]
131     : 1
132     ;
133     tP2j
134     :
135     : [P2wP1 + {1'1}] & [P2d - {1'1}]
136     : 0.4
137     ;
138     tP2e
139     :
140     : [P2s + {1'1}] & [P2d - {1'1}]
141     : 0.6
142     ;
143 }

```

A.6 CANDL Syntax of Cyclic Server Polling System

```

1  colspn [polling]
2  {
3  constants:
4  all:
5      int N = 10;
6      int M = 1;
7  param:
8      double mu = 1;
9      double gamma = 200;
10     double lambda = 1/N;
11
12  colorsets:
13     Station = {1..N};
14     Server = {1..M};
15
16  variables:
17     Station : x;
18     Server : y;
19
20  places:

```

```

21 discrete:
22     Server s = 1'all;
23     Server a = 0'all;
24     Station si = 0'all;
25
26 transitions:
27     walk
28     : [si < {1'x}] & [a < {1'y}] & [s = {[x<N]x'y}]
29     : [s + {1'y}] & [s - {[x=N]N'y}]
30     : gamma
31     ;
32     choose
33     : [a < {1'y}] & [s = {x'y}] & [si >= {1'x}]
34     : [a + {1'y}]
35     : gamma
36     ;
37     new
38     : [si < {1'x}]
39     : [si + {1'x}]
40     : lambda
41     ;
42     serve
43     : [s = {[x<N]x'y}]
44     : [s + {1'y}] & [si - {1'x}] & [a - {1'y}] & [s - {[x=N]N'y}]
45     : mu
46     ;
47 }

```


Bibliography

- [ACK06] A. Auger, P. Chatelain, and P. Koumoutsakos. “R-leaping: Accelerating the stochastic simulation algorithm by reaction leaps”. In: *The Journal of chemical physics* 125.8 (2006), p. 084103.
- [ACR01] G. Agha, F. de Cindio, and G. Rozenberg, eds. *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*. Vol. 2001. Lecture Notes in Computer Science. Springer, 2001.
- [ADN89] M. Ajmone Marsan, S. Donatelli, and F. Neri. “GSPN models of multiserver multiqueue systems”. In: *Petri Nets and Performance Models, 1989. PNPM89., Proceedings of the Third International Workshop on*. IEEE. 1989, pp. 19–28.
- [ADO00] W. M. P. van der Aalst, J. Desel, and A. Oberweis, eds. *Business Process Management, Models, Techniques, and Empirical Studies*. Vol. 1806. Lecture Notes in Computer Science. Springer, 2000.
- [AG07] S. Asmussen and P. W. Glynn. *Stochastic simulation: algorithms and analysis*. Vol. 57. Springer Science & Business Media, 2007.
- [Ajm+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. 2nd Edition. Wiley Series in Parallel Computing, John Wiley and Sons, 1995.
- [Azi+00] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. “Model checking continuous-time Markov chains”. In: *ACM Trans. on Computational Logic* 1.1 (2000), pp. 162–170.
- [Bai+00a] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. “Model checking continuous-time Markov chains by transient analysis”. In: *Proc. CAV 2000*. LNCS 1855, Springer, 2000, pp. 358–372.

- [Bai+00b] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. “On the logical characterisation of performability properties”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2000, pp. 780–792.
- [Bai98] C. Baier. “On algorithmic verification methods for probabilistic systems”. Habilitation thesis. University of Mannheim, 1998.
- [Bal+04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. “Model-based performance prediction in software development: A survey”. In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 295–310.
- [Bal+09] P. Ballarini, M. Forlin, T. Mazza, and D. Prandi. “Efficient Parallel Statistical Model Checking of Biochemical Networks”. In: *Proc. PDMC*. 2009, pp. 47–61.
- [Bal+10] P. Baldan, N. Cocco, A. Marin, and M. Simeoni. “Petri nets for modelling metabolic pathways: a survey”. In: *Natural Computing* 9.4 (2010), pp. 955–989.
- [BCD02] L. D. Brown, T. T. Cai, and A. DasGupta. “Confidence Intervals for a binomial proportion and asymptotic expansions”. In: *Annals of Statistics* 30.1 (2002), pp. 160–201.
- [BGH09] S. Basu, A. P. Ghosh, and R. He. “Approximate Model Checking of PCTL Involving Unbounded Path Properties”. In: *ICFEM*. 2009, pp. 326–346.
- [Bir31] G. D. Birkhoff. “Proof of the ergodic theorem”. In: *Proceedings of the National Academy of Sciences* 17.12 (1931), pp. 656–660.
- [BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [Blä+13] M. Blätke, A. Dittrich, C. Rohr, M. Heiner, F. Schaper, and W. Marwan. “JAK/STAT signalling - an executable model assembled from molecule-centred modules demonstrating a module-oriented database concept for systems and synthetic biology”. In: *Molecular BioSystem* 9.6 (2013), pp. 1290–1307.

- [Blä+14] M. Blätke, C. Rohr, M. Heiner, and W. Marwan. “A Petri Net based Framework for Biomodel Engineering”. In: *Large-Scale Networks in Engineering and Life Sciences*. Ed. by P. Benner, R. Findeisen, D. Flockerzi, U. Reichl, and K. Sundmacher. Modeling and Simulation in Science, Engineering and Technology. Springer, Birkhäuser Mathematics, Dec. 2014, pp. 317–366.
- [BP03] N. Bahi-Jaber and D. Pontier. “Modeling transmission of directly transmitted infectious diseases using colored stochastic petri nets”. In: *Mathematical Biosciences* 185 (2003), pp. 1–13.
- [BR15] M. Blätke and C. Rohr. “A Colored Petri net approach for spatial Biomodel Engineering based on the modular model composition framework Biomodelkit”. In: *Proc. Int. Workshop on Biological Processes & Petri Nets (BioPPN 2015)*. Vol. 1373. CEUR Workshop Proceedings. CEUR-WS.org, June 2015, pp. 37–54.
- [Cal+06] M. Calder, A. Duguid, S. Gilmore, and J. Hillston. “Stronger computational modelling of signalling pathways using both continuous and discrete-state methods”. In: *Proc. CMSB 2006*. LNBI 4210, Springer, 2006, pp. 63–78.
- [CE81] E. M. Clarke and E. A. Emerson. “Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic”. In: *Proceedings of the Workshop on Logics of Programs*. LNCS #131. Springer-Verlag, 1981, pp. 52–71.
- [CGP01] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. Cambridge: MIT Press, 2001.
- [CGP06] Y. Cao, D. T. Gillespie, and L. R. Petzold. “Efficient step size selection for the tau-leaping simulation method.” In: *J Chem Phys* 124.4 (Jan. 2006), p. 044109.
- [CGP07] Y. Cao, D. T. Gillespie, and L. R. Petzold. “Adaptive explicit-implicit tau-leaping method with automatic tau selection.” In: *J Chem Phys* 126.22 (June 2007), p. 224101.

- [CH06] L. Cloth and B. R. H. M. Haverkort. “Five Performability Algorithms. A Comparison”. In: *MAM 2006: Markov Anniversary Meeting, Charleston, SC, USA*. Boson Books, 2006, pp. 39–54.
- [Cha07] C. Chaouiya. “Petri Net Modelling of Biological Networks”. In: *Briefings in Bioinformatics* 8.4 (2007), pp. 210–219.
- [Cho+03] 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. CMSB 2003*. LNCS 2602, Springer, 2003, pp. 127–141.
- [Cia94] G. Ciardo. “Petri Nets with Marking-Dependent Arc Cardinality: Properties and Analysis”. In: *Application and Theory of Petri Nets 1994*. Ed. by R. Valette. Vol. 815. LNCS. Springer-Verlag, 1994, pp. 179–198.
- [Clo+05] L. Cloth, J.-P. Katoen, M. Khattri, and R. Pulungan. “Model checking Markov reward models with impulse rewards”. In: *2005 International Conference on Dependable Systems and Networks (DSN’05)*. IEEE. 2005, pp. 722–731.
- [Clo06] L. Cloth. “Model checking algorithms for Markov reward models”. PhD thesis. University of Twente, 2006.
- [CLP04] Y. Cao, H. Li, and L. Petzold. “Efficient formulation of the stochastic simulation algorithm for chemically reacting systems.” In: *J Chem Phys* 121.9 (Sept. 2004), pp. 4059–4067.
- [CT93] G. Ciardo and K. S. Trivedi. “A Decomposition Approach for Stochastic Reward Net Models”. In: *Performance Evaluation* 18.1 (1993), pp. 37–59.
- [CX07] X. Cai and Z. Xu. “K-leap method for accelerating stochastic simulation of coupled chemical reactions”. In: *Journal of Chemical Physics* 126.7 (2007), pp. 74102–74102.
- [DC09] C. Dittamo and D. Cangelosi. “Optimized parallel implementation of gillespie’s first reaction method on graphics processing units”. In: *Computer Modeling and Simulation, 2009. ICCMS’09. International Conference on*. IEEE. 2009, pp. 156–161.

- [DDO08] R. M. Dijkman, M. Dumas, and C. Ouyang. “Semantics and analysis of business process models in BPMN”. In: *Information and Software technology* 50.12 (2008), pp. 1281–1294.
- [DG08a] R. Donaldson and D. Gilbert. “A Model Checking Approach to the Parameter Estimation of Biochemical Pathways”. In: *Proc. 6th International Conference on Computational Methods in Systems Biology (CMSB 2008)*. LNCS 5307, Springer, 2008, pp. 269–287.
- [DG08b] R. Donaldson and D. Gilbert. *A Monte Carlo Model Checker for Probabilistic LTL with Numerical Constraints*. Tech. rep. University of Glasgow, Dep. of CS, 2008.
- [Did+09] F. Didier, T. A. Henzinger, M. Mateescu, and V. Wolf. “Fast Adaptive Uniformization for the Chemical Master Equation”. In: *HIBI*. IEEE Comp. Soc., 2009, pp. 118–127.
- [Doi+99] A. Doi, R. Drath, M. Nagaska, H. Matsuno, and S. Miyano. “Protein Dynamics Observations of Lambda-Phage by Hybrid Petri net”. In: *Genome Informatics* (1999), pp. 217–218.
- [Dur+04] W. Duridanova, W. Hummel, O. Fengler, and W. Fengler. “Verifikation von Spezifikationsmodellen mit Intervall-Petri-Netzen”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen. 7. GI/ITG/GMM-Workshop zu Modellierung und Verifikation*. Ed. by D. Stoffel and W. Kunz. Kaiserslautern: Shaker-Verlag, 2004, pp. 184–193.
- [Dur64] R. Durstenfeld. “Algorithm 235: Random Permutation”. In: *Commun. ACM* 7.7 (July 1964), p. 420.
- [EL00] M. B. Elowitz and S. Leibler. “A synthetic oscillatory network of transcriptional regulators”. In: *Nature* 403.6767 (2000), pp. 335–338.
- [ESK15] P. Erdrich, R. Steuer, and S. Klamt. “An algorithm for the reduction of genome-scale metabolic network models to meaningful core models”. In: *BMC systems biology* 9.1 (2015).

- [F+63] R. A. Fisher, F. Yates, et al. *Statistical tables for biological, agricultural and medical research*. 6th. Oliver and Boyd, Edinburgh, 1963.
- [FA73] M. J. Flynn and T. Agerwala. “Comments on Capabilities, Limitations and Correctness of Petri Nets”. In: *Proceedings of the 1st Annual Symposium on Computer Architecture*. ACM Press, 1973, pp. 81–86.
- [Fis92] R. Fisher. “Statistical methods for research workers”. In: *Breakthroughs in Statistics*. Springer, 1992, pp. 66–70.
- [FR07] F. Fages and A. Rizk. “On the Analysis of Numerical Data Time Series in Temporal Logic”. In: *Proc. CMSB 2007*. LNCS/LNBI 4695, Springer, 2007, pp. 48–63.
- [Gao+11] Q. Gao, F. Liu, D. Gilbert, M. Heiner, and D. Tree. “A Multiscale Approach to Modelling Planar Cell Polarity in Drosophila Wing using Hierarchically Coloured Petri Nets”. In: *Proc. 9th International Conference on Computational Methods in Systems Biology (CMSB 2011)*. Paris: ACM digital library, Sept. 2011, pp. 209–218.
- [Gao+13] Q. Gao, D. Gilbert, M. Heiner, F. Liu, D. Maccagnola, and D. Tree. “Multiscale Modelling and Analysis of Planar Cell Polarity in the Drosophila Wing”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 10.2 (2013). online 01 August 2012, pp. 337–351.
- [GB00] M. A. Gibson and J. Bruck. “Efficient exact stochastic simulation of chemical systems with many species and many channels”. In: *The Journal of Physical Chemistry A* 104 (2000), pp. 1876–1889.
- [GBC07] L. Gomes, J. Barros, and A. Costa. “Petri Nets Tools and Embedded Systems Design”. In: *Proc. Workshop on Petri Nets and Software Engineering (PNSE07) at Int. Conf. on Application and Theory of Petri Nets (ICATPN ’07 Siedlce)*. 2007, pp. 214–219.
- [Ger01] R. German. *Performance analysis of communication systems with non-Markovian stochastic Petri nets*. Wiley, 2001.

- [GH06] D. Gilbert and M. Heiner. “From Petri nets to differential equations - an integrative approach for biochemical network analysis”. In: *Proc. ICATPN 2006*. LNCS 4024, Springer, 2006, pp. 181–200.
- [GH16] D. Gilbert and M. Heiner. *E.coli K-12 genome scale metabolic model*. personal communication. 2016.
- [GHL07] D. Gilbert, M. Heiner, and S. Lehrack. “A Unifying Framework for Modelling and Analysing Biochemical Pathways Using Petri Nets”. In: *Proc. CMSB 2007*. LNCS/LNBI 4695, Springer, 2007, pp. 200–216.
- [Gil+13] D. Gilbert, M. Heiner, F. Liu, and N. Saunders. “Colouring Space - A Coloured Framework for Spatial Modelling in Systems Biology”. In: *Proc. PETRI NETS 2013*. Ed. by J. Colom and J. Desel. Vol. 7927. LNCS. Milano: Springer, June 2013, pp. 230–249.
- [Gil01] D. T. Gillespie. “Approximate accelerated stochastic simulation of chemically reacting systems”. In: *The Journal of Chemical Physics* 115.4 (2001), pp. 1716–1733.
- [Gil76] D. Gillespie. “A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Species”. In: *Journal of Computational Physics* 22 (1976), pp. 403–434.
- [Gil77] D. Gillespie. “Exact stochastic simulation of coupled chemical reactions”. In: *The Journal of Physical Chemistry* 81(25) (1977), pp. 2340–2361.
- [Gil92] D. T. Gillespie. “A rigorous derivation of the chemical master equation”. In: *Physica A: Statistical Mechanics and its Applications* 188.1 (1992), pp. 404–425.
- [GP98] P. J. E. Goss and J. Peccoud. “Quantitative modeling of stochastic systems in molecular biology by using stochastic Petri nets”. In: *Proc. Natl. Acad. Sci. USA* 95.June (1998), pp. 2340–2361.
- [GS01] G. Grimmett and D. Stirzaker. *Probability and random processes*. Oxford university press, 2001.
- [Haa03] P. Haas. *Stochastic Petri nets: Modelling, Stability, Simulation*. Springer, 2003.

- [Haa04] P. J. Haas. “Stochastic petri nets for modelling and simulation”. In: *Simulation Conference, 2004. Proceedings of the 2004 Winter*. Vol. 1. IEEE. 2004.
- [Hav+02] B. Haverkort, L. Cloth, H. Hermanns, J.-P. Katoen, and C. Baier. “Model Checking Performability Properties”. In: *IEEE CS Press* (2002), pp. 103–112.
- [HB03] R. Hamadi and B. Benatallah. “A Petri net-based model for web service composition”. In: *Proceedings of the 14th Australasian database conference-Volume 17*. Australian Computer Society, Inc. 2003, pp. 191–200.
- [HDG10] M. Heiner, R. Donaldson, and D. Gilbert. “Petri Nets for Systems Biology”. In: *Symbolic Systems Biology: Theory and Methods*. Ed. by M. Iyengar. Jones & Bartlett Learning, LCC, 2010. Chap. 3, pp. 61–97.
- [HDS99] M. Heiner, P. Deussen, and J. Spranger. “A Case Study in Design and Verification of Manufacturing System Control Software with Hierarchical Petri Nets”. In: *Journal of Advanced Manufacturing Technology* 15 (1999), pp. 139–152.
- [Hei+09] M. Heiner, S. Lehrack, D. Gilbert, and W. Marwan. “Extended Stochastic Petri Nets for Model-Based Design of Wetlab Experiments”. In: LNBI 5750, Springer. 2009, pp. 138–163.
- [Hei+10] M. Heiner, C. Rohr, M. Schwarick, and S. Streif. “A Comparative Study of Stochastic Analysis Techniques”. In: *Proc. 8th International Conference on Computational Methods in Systems Biology (CMSB 2010)*. Trento: ACM digital library, Sept. 2010, pp. 96–106.
- [Hei+12] M. Heiner, M. Herajy, F. Liu, C. Rohr, and M. Schwarick. “Snoopy - a unifying Petri net tool”. In: *Proc. PETRI NETS 2012*. Vol. 7347. LNCS. Hamburg: Springer, June 2012, pp. 398–407.
- [Hei+16] M. Heiner, C. Rohr, M. Schwarick, and A. A. Tovchigrechko. “MARCIE’s Secrets of Efficient Model Checking”. In: *Transactions on Petri Nets and Other Models of Concurrency XI*. Ed.

- by M. Koutny, J. Desel, and J. Kleijn. Vol. 9930. LNCS. Berlin, Heidelberg: Springer, 2016, pp. 286–296.
- [HG11] M. Heiner and D. Gilbert. “How Might Petri Nets Enhance Your Systems Biology Toolkit”. In: *Proc. PETRI NETS 2011*. LNCS 6709, Springer, 2011, pp. 17–37.
- [HGD08] M. Heiner, D. Gilbert, and R. Donaldson. “Petri Nets in Systems and Synthetic Biology”. In: *SFM*. LNCS 5016, Springer, 2008, pp. 215–264.
- [HJ94] H. Hansson and B. Jonsson. “A Logic for Reasoning about Time and Reliability”. In: *Formal Aspects of Computing* 6.5 (1994), pp. 512–535.
- [HLR14] M. Herajy, F. Liu, and C. Rohr. “Coloured hybrid Petri nets for systems biology”. In: *Proc. of the 5th International Workshop on Biological Processes & Petri Nets (BioPPN), satellite event of PETRI NETS 2014*. Vol. 1159. CEUR Workshop Proceedings. CEUR-WS.org, June 2014, pp. 60–76.
- [HRS13] M. Heiner, C. Rohr, and M. Schwarick. “MARCIE - Model checking And Reachability analysis done effiCIently”. In: *Proc. PETRI NETS 2013*. Ed. by J. Colom and J. Desel. Vol. 7927. LNCS. Milano: Springer, June 2013, pp. 389–399.
- [HSH13] M. Herajy, M. Schwarick, and M. Heiner. “Hybrid Petri Nets for Modelling the Eukaryotic Cell Cycle”. In: *ToPNoC VIII, LNCS 8100* (2013), pp. 123–141.
- [Huc+03] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, and et al. “The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models”. In: *J. Bioinformatics* 19 (2003), pp. 524–531.
- [IT90] O. Ibe and K. Trivedi. “Stochastic Petri Net Models of Polling Systems”. In: *IEEE Journal on Selected Areas in Communications* 8.9 (1990), pp. 1649–1657.

- [Jen53] A. Jensen. “Markoff chains as an aid in the study of Markoff processes”. In: *Scandinavian Actuarial Journal* 1953.sup1 (1953), pp. 87–91.
- [Jen81] K. Jensen. “Coloured Petri nets and the invariant-method”. In: *Theoretical Computer Science* 14 (1981), pp. 317–336.
- [Joh75] D. B. Johnson. “Finding all the elementary circuits of a directed graph”. In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84.
- [KNP07] M. Kwiatkowska, G. Norman, and D. Parker. “Stochastic Model Checking”. In: *SFM*. LNCS 4486, Springer, 2007, pp. 220–270.
- [Knu97] D. E. Knuth. “Volume 2: Seminumerical Algorithms”. In: *The Art of Computer Programming* 192 (1997).
- [Koh+11] C. H. Koh, M. Nagasaki, A. Saito, C. Li, L. Wong, and S. Miyano. “MIRACH: Efficient Model Checker for Quantitative Biological Pathway Models”. In: *Bioinformatics* 27 (2011).
- [Kur72] T. G. Kurtz. “The relationship between stochastic and deterministic models for chemical reactions”. In: *The Journal of Chemical Physics* 57.7 (1972), pp. 2976–2978.
- [LB07] A. Loinger and O. Biham. “Stochastic simulations of the repressilator circuit”. In: *Physical Review E* 76.5 (2007), p. 051917.
- [LBS00] A. Levchenko, J. Bruck, and P. Sternberg. “Scaffold proteins may biphasically affect the levels of mitogen-activated protein kinase signaling and reduce its threshold properties”. In: *Proc Natl Acad Sci USA* 97.11 (2000), pp. 5818–5823.
- [LDT06] P. L’Ecuyer, V. Demers, and B. Tuffin. “Splitting for rare-event simulation”. In: *Proceedings of the 2006 winter simulation conference*. IEEE, 2006, pp. 137–148.
- [Leh07] S. Lehrack. “A Tool to Model and Simulate Stochastic Petri Nets in the Context of Biochemical Networks (in German)”. MA thesis. Brandenburg University of Technology Cottbus, Computer Science Dept., 2007.

- [LH13a] F. Liu and M. Heiner. “Modeling membrane systems using colored stochastic Petri nets”. In: *Nat. Computing* 12.4 (2013), pp. 617–629.
- [LH13b] F. Liu and M. Heiner. “Multiscale modelling of coupled Ca²⁺-channels using coloured stochastic Petri nets”. In: *IET Systems Biology* 7.4 (Aug. 2013), pp. 106–113.
- [LH14] F. Liu and M. Heiner. “Petri Nets for Modeling and Analyzing Biochemical Reaction Networks”. In: *Approaches in Integrative Bioinformatics*. Ed. by M. Chen and R. Hofestädt. Springer, 2014. Chap. 9, pp. 245–272.
- [LHR12] F. Liu, M. Heiner, and C. Rohr. *Manual for Colored Petri Nets in Snoopy*. Tech. rep. 02–12. BTU Cottbus, Computer Science Institute, Mar. 2012.
- [LHY14] F. Liu, M. Heiner, and M. Yang. “Modeling and analyzing biological systems using colored hierarchical Petri nets, illustrated by *C. elegans* vulval development”. In: *WSPC Journal of Biological Systems* 22.3 (2014), pp. 463–493.
- [Liu+14] F. Liu, M. Blätke, M. Heiner, and M. Yang. “Modelling and simulating reaction–diffusion systems using coloured Petri nets”. In: *Computers in Biology and Medicine* 53 (Oct. 2014). online July 2014, pp. 297–308.
- [Liu12] F. Liu. “Colored Petri Nets for Systems Biology”. English. PhD thesis. BTU Cottbus, Dep. of CS, Jan. 2012.
- [LP06] H. Li and L. Petzold. “Logarithmic direct method for discrete stochastic simulation of chemically reacting systems”. In: *Journal of Chemical Physics* (2006).
- [LP07] H. Li and L. R. Petzold. “Stochastic simulation of biochemical systems on the graphics processing unit”. In: *Bioinformatics. Cité pages 32 et 35* (2007).
- [Mar15] S. Marsland. *Machine learning: an algorithmic perspective*. CRC press, 2015.

- [McC+06] J. M. McCollum, G. D. Peterson, C. D. Cox, M. L. Simpson, and N. F. Samatova. “The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior”. In: *Comput. Biol. Chem.* 30.1 (2006), pp. 39–49.
- [McQ67] D. A. McQuarrie. “Stochastic Approach to Chemical Kinetics”. In: *Journal of Applied Probability* 4.3 (1967), pp. 413–478.
- [MN98] M. Matsumoto and T. Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Trans. Model. Comput. Simul.* 8.1 (1998), pp. 3–30.
- [Mon+13] J. M. Monk, P. Charusanti, R. K. Azizb, J. A. Lermant, N. Premyodhinb, J. D. Orth, A. M. Feist, and B. O. Palsson. “Genome-scale metabolic reconstructions of multiple *Escherichia coli* strains highlight strain-specific adaptations to nutritional environments”. In: *PNAS* 110.50 (2013), pp. 20338–20343.
- [MRH12] W. Marwan, C. Rohr, and M. Heiner. “Petri nets in Snoopy: A unifying framework for the graphical display, computational modelling, and simulation of bacterial regulatory networks”. In: *Methods in Molecular Biology – Bacterial Molecular Networks*. Ed. by J. Helden, A. Toussaint, and D. Thieffry. Vol. 804. Methods in Molecular Biology. Humana Press, 2012. Chap. 21, pp. 409–437.
- [MS11] S. Mauch and M. Stalzer. “Efficient Formulations for Exact Stochastic Simulation of Chemical Systems”. In: *IEEE/ACM Trans. Comput. Biol. Bioinformatics* 8 (1 Jan. 2011), pp. 27–35.
- [Nap+09] L. Napione et al. “On the Use of Stochastic Petri Nets in the Analysis of Signal Transduction Pathways for Angiogenesis Process”. In: *Proc. CMSB 2009*. LNCS/LNBI 5688, Springer. 2009, pp. 281–295.
- [OFP10] J. D. Orth, R. M. Fleming, and B. O. Palsson. “Reconstruction and use of microbial metabolic networks: the core *Escherichia coli* metabolic model as an educational guide”. In: *EcoSal Plus* 4.1 (2010).

- [OSW69] I. Oppenheim, K. Shuler, and G. Weiss. “Stochastic and deterministic formulation of chemical rate equations”. In: *The Journal of Chemical Physics* 50.1 (1969), pp. 460–466.
- [Pâr+15] O. Pârva, D. Gilbert, M. Heiner, F. Liu, N. Saunders, and S. Shaw. “Spatial-temporal modelling and analysis of bacterial colonies with phase variable genes”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 25.2 (May 2015), 25p.
- [Paw90] K. Pawlikowski. “Steady-state Simulation of Queueing Processes: Survey of Problems and Solutions”. In: *ACM Comput. Surv.* 22.2 (1990), pp. 123–170.
- [Pec98] J. Peccoud. “Stochastic Petri Nets for Genetic Networks”. In: *MS-Medicine Sciences* 14. 1998, pp. 991–993.
- [Pet62] C. A. Petri. “Kommunikation mit Automaten”. PhD thesis. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
- [PLM06] F. Panneton, P. L’ecuyer, and M. Matsumoto. “Improved long-period generators based on linear recurrences modulo 2”. In: *ACM Transactions on Mathematical Software (TOMS)* 32.1 (2006), pp. 1–16.
- [Pnu77] A. Pnueli. “The Temporal Logic of Programs”. In: *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*. IEEE Computer Society Press, 1977, pp. 46–57.
- [PP02] A. Papoulis and S. U. Pillai. *Probability, Random Variables, and Stochastic Processes, Fourth Edition*. McGraw-Hill Higher Education, 2002.
- [Pre+07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.
- [RMH10] C. Rohr, W. Marwan, and M. Heiner. “Snoopy - a unifying Petri net framework to investigate biomolecular networks”. In: *Bioinformatics* 26.7 (2010), pp. 974–975.

- [Roh10] C. Rohr. “Simulative CSL model checking of Stochastic Petri nets in IDD-MC”. In: *Proc. 17th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2010)*. Vol. 643. CEUR Workshop Proceedings. CEUR-WS.org, Oct. 2010, pp. 88–93.
- [Roh12] C. Rohr. “Simulative Model Checking of Steady-State and Time-Unbounded Temporal Operators”. In: *Proc. of the 3rd International Workshop on Biological Processes & Petri Nets (BioPPN), satellite event of PETRI NETS 2012*. Vol. 852. CEUR Workshop Proceedings. CEUR-WS.org, June 2012, pp. 62–75.
- [Roh13] C. Rohr. “Simulative Model Checking of Steady State and Time-Unbounded Temporal Operators”. In: *Transactions on Petri Nets and Other Models of Concurrency VIII*. Ed. by M. Koutny, W. Aalst, and A. Yakovlev. Vol. 8100. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 142–158.
- [Roh16] C. Rohr. “Discrete-Time Leap Method For Stochastic Simulation”. In: *Proc. Int. Workshop on Biological Processes & Petri Nets (BioPPN 2016)*. Vol. 1591. CEUR Workshop Proceedings. CEUR-WS.org, June 2016, pp. 362–376.
- [RP09] D. Rabih and N. Pekergin. “Statistical Model Checking Using Perfect Simulation”. In: *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis. ATVA ’09*. Macao, China: Springer, 2009, pp. 120–134.
- [San08] W. Sandmann. “Brief Communication: Discrete-time stochastic modeling and simulation of biochemical networks”. In: *Comput. Biol. Chem.* 32 (4 Aug. 2008), pp. 292–297.
- [Sch14] M. Schwarick. “Symbolic on-the-fly analysis of stochastic Petri nets”. English. PhD thesis. BTU Cottbus, Dep. of CS, June 2014.
- [SH09] M. Schwarick and M. Heiner. “CSL model checking of biochemical networks with Interval Decision Diagrams”. In: *Proc. CMSB 2009*. Bologna, Italy: LNCS/LNBI 5688, Springer, 2009, pp. 296–312.

- [SM08] W. Sandmann and C. Maier. “On the statistical accuracy of stochastic simulation algorithms implemented in Dizzy”. In: *Proc. WCSB 2008*. 2008, pp. 153–156.
- [SRH11] M. Schwarick, C. Rohr, and M. Heiner. “MARCIE - Model checking And Reachability analysis done effiCIently”. In: *Proc. 8th International Conference on Quantitative Evaluation of SysTems (QEST 2011)*. Aachen, Germany: IEEE CS Press, Sept. 2011, pp. 91–100.
- [SRH14] M. Schwarick, C. Rohr, and M. Heiner. *MARCIE Manual: An analysis tool for extended stochastic Petri nets*. Computer Science Reports 03-14. Brandenburg University of Technology Cottbus, Nov. 2014.
- [Ste94] W. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton Univ. Press, 1994.
- [STP08] A. Slepoy, A. P. Thompson, and S. J. Plimpton. “A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks”. In: *The journal of chemical physics* 128.20 (2008), p. 205101.
- [Taf+08] A. Tafazzoli, J. R. Wilson, E. K. Lada, and N. M. Steiger. “Skart: A skewness- and autoregression-adjusted batch-means procedure for simulation analysis”. In: *Winter Simulation Conference*. 2008, pp. 387–395.
- [Taf+11] A. Tafazzoli, J. R. Wilson, E. K. Lada, and N. M. Steiger. “Performance of Skart: A Skewness- and Autoregression-Adjusted Batch Means Procedure for Simulation Analysis”. In: *INFORMS Journal on Computing* 23.2 (2011), pp. 297–314.
- [TB04] T. Tian and K. Burrage. “Binomial leap methods for simulating stochastic chemical kinetics”. In: *The Journal of chemical physics* 121.21 (2004), pp. 10356–10364.
- [TFZ09] W. Tan, Y. Fan, and M. Zhou. “A petri net-based method for compatibility analysis and composition of web services in business

- process execution language”. In: *IEEE Transactions on Automation Science and Engineering* 6.1 (2009), pp. 94–106.
- [Tov08] A. Tovchigrechko. “Model Checking Using Interval Decision Diagrams”. PhD thesis. BTU Cottbus, Dep. of CS, 2008.
- [Tuk49] J. W. Tukey. “Comparing individual means in the analysis of variance”. In: *Biometrics* (1949), pp. 99–114.
- [TW10] A. Tafazzoli and J. R. Wilson. “Skart: A skewness- and auto-regression-adjusted batch-means procedure for simulation analysis”. In: *IIE Transactions* 43.2 (2010), pp. 110–128.
- [Val78] R. Valk. “Self-Modifying Nets, a Natural Extension of Petri Nets”. In: *ICALP*. 1978, pp. 464–476.
- [Wal92] A. Wald. “Sequential tests of statistical hypotheses”. In: *Breakthroughs in Statistics*. Springer, 1992, pp. 256–298.
- [Wen91] L. Wen. “An analytic technique to prove Borel’s strong law of large numbers”. In: *The American Mathematical Monthly* 98.2 (1991), pp. 146–148.
- [Wil06] D. Wilkinson. *Stochastic Modelling for System Biology*. CRC Press, New York, 1st Edition, 2006.
- [Wil22] E. B. Wilson. “Probable inference, the law of succession, and statistical inference”. In: *Journal of the American Statistical Association* 22 (1922), pp. 209–212.
- [YCZ11] H. Younes, E. Clarke, and P. Zuliani. “Statistical Verification of Probabilistic Properties with Unbounded Until”. In: *Formal Methods: Foundations and Applications*. Vol. 6527. Lecture Notes in Computer Science. Springer, 2011, pp. 144–160.
- [You+06] H. Younes, M. Kwiatkowska, G. Norman, and D. Parker. “Numerical vs. Statistical Probabilistic Model Checking”. In: *STTT* 8.3 (2006), pp. 216–228.
- [YS02] H. Younes and R. Simmons. “Probabilistic Verification of Discrete Event Systems using Acceptance Sampling”. In: *Computer Aided Verification*. Vol. 2404. Lecture Notes in Computer Science. LNCS 2404, Springer, 2002, pp. 223–235.

- [YV99] S. Yee and J. Ventura. “A dynamic programming algorithm to determine optimal assembly sequences using Petri nets”. In: *International Journal of Industrial Engineering - Theory, Applications and Practice, Vol.6, No.1* (1999), pp. 27–37.
- [Zap08] I. S. Zapreev. “Model checking Markov chains : techniques and tools”. PhD thesis. Enschede: University of Twente, Mar. 2008.
- [ZC06] J. Zhang and B. H. Cheng. “Model-based development of dynamically adaptive software”. In: *Proceedings of the 28th International Conference on Software Engineering*. ACM. 2006, pp. 371–380.