Fuzzy coloured Petri nets for modelling biological systems with uncertain kinetic parameters

Von der Fakultät 1 - MINT - Mathematik, Informatik, Physik,
Elektro- und Informationstechnik
der Brandenburgischen Technischen Universität
Cottbus–Senftenberg

genehmigte Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften
(*Dr. rer. nat.*)

vorgelegt von

M.Sc. Informatik Ingenieurwesen
George Assaf

geboren am 20. 02. 1989 in Damaskus (Damascus Suburb), Syrien

Vorsitzender: Prof. Dr.-Ing. Andriy Panchenko
Gutachterin: Prof. Dr.-Ing. Monika Heiner
Gutachter: Prof. Dr. rer. nat. habil. Wolfgang Marwan
Tag der mündlichen Prüfung: 15.12.2021

DOI: 10.26127/BTUOpen-5851

Eidesstattliche Erklärung

Hiermit erkläre ich Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und angefertigt habe, nur die angegebenen Quellen und Hilfsmittel und keine anderen benutzt bzw. verwendet habe und die wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Veröffentlichung der Dissertation verletzt keine bestehenden Schutzrechte.

Cottbus, den 25.07.2021
George Assaf

Statutory Declaration

I herewith declare that I have produced this thesis without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This thesis has not previously been presented in identical or similar form to any other German or foreign examination board.

The thesis was conducted from 2018 to 2021 under the supervision of Prof. Dr.-Ing Monika Heiner at Brandenburg University of Technology Cottbus-Senftenberg

Cottbus, den 25.07.2021
George Assaf

Acknowledgements

In this moment, I can not find those words which are able to express my gratitude, especially for the people who enriched my academic life with their experience. However, I will try to translate my feelings into some words.

I would like to express my profound gratitude to Prof. Dr.-Ing. Monika Heiner for giving me this research opportunity and for all kinds of support that she provided to me in both scientific and personal aspects. I must admit that I would not have reached this moment without her advice and guidance.

Next, I would like to thank Prof. Dr. rer. nat. habil. Wolfgang Marwan, Otto-von-Guericke Universität Magdeburg for supporting my scholarship application as well as for the biological information that he provided to us in his research lab.

Next, I would like to express my sincere gratitude to Prof. Dr. rer. nat. Fei Liu, South China University of Technology for his guidance and advice during this research. I highly appreciate his fruitful cooperation in publishing the research results that I came up in this research.

Further, I would like to thank all the staff members at the Data Structure and Software Dependability Chair for their assistance, including Mrs Sigrid Schenk and Mr Sergey Romanov.

Special thank goes for my colleague Mr Jacek Chodak, who taught me how to improve my *C++* developing skills and for all the daily (fruitful) discussions with him.

I have to thank the institute of computer science at the Brandenburg Technical University (BTU) for all the facilities that were provided to me to achieve this research.

Last but not least, I would like to acknowledge the financial support of the KAAD scholarship (Katholischer Akademischer Ausländer-Dienst) from 2019/October to 2021 /December.

Cottbus, den 25.07.2021
George Assaf

Abstract

Over the last twenty years, Petri nets have been increasingly adopted for modelling and simulating biological systems, as they offer an intuitive and graphical approach for this purpose. Their usability convenience comes from the fact that they offer many types of elements to describe systems in a qualitative and quantitative way. Coloured Petri nets are particularly useful to model systems with repeated components in a compact fashion. Our tool Snoopy for modelling and simulating Petri nets is one of the most well-known tools supporting a family of related Petri net classes comprising stochastic, continuous and hybrid Petri nets, and covering uncoloured and coloured Petri nets alike. However, kinetic information of a biological system, i.e. kinetic parameters may be uncertain, due to many reasons, e.g. environmental factors. Besides, coloured Petri nets as they were previously supported in Snoopy suffered from some inconsistencies. Due to these inconsistencies, exploring the model behaviour using different sizes (scaleability) was not feasible. Both challenges call for a new and more powerful approach integrating the modelling of uncertainties together with modelling features supporting repeated structures in a compact and scalable way.

This thesis comprises two major contributions: Firstly, we introduce the definition and present the simulation algorithm for both uncoloured and coloured fuzzy Petri nets, by extending the existing quantitative uncoloured and coloured Petri nets in Snoopy. This includes discretising the uncertain kinetic parameters to crisp values by using sampling strategies. Secondly, we harmonise coloured Petri nets in Snoopy with their uncoloured counterparts and we extend the Snoopy's coloured Petri nets by all the features, which are supported by the coloured abstract net description language - an exchange format of coloured Petri nets in our *PetriNuts* tool family.

By performing fuzzy simulation, one can obtain two kinds of output: fuzzy bands of each output variable and their corresponding timed-membership functions. Each fuzzy band describes the uncertainties associated with the input, whereas membership functions give more accurate information about the associated uncertainties. The most important features that we obtain by harmonising coloured Petri nets are to develop scaleable models, by defining scaling factors as constants and unifying the usage of coloured Petri nets with the other tools in our *PetriNuts* tool family.

Keywords Uncertain Biological Systems; Modelling and Simulation; Quantitative Fuzzy Petri Nets; Coloured Quantitative Fuzzy Petri Nets; Software Harmonisation.

Zusammenfassung

In den letzten zwanzig Jahren wurden Petri-Netze zunehmend für die Modellierung und Simulation von biologischen Systemen eingesetzt. Sie bieten dafür sowohl eine qualitative als auch quantitative Modellierungsmethode an. Farbige Petri-Netze sind nützlich für die Modellierung von Systemen mit sich wiederholenden Komponenten. *Snoopy* ist eines der bekanntesten Werkzeuge, das eine Familie von Petri-Netzen unterstützt, darunter sowohl ungefärbte als auch gefärbte stochastische, kontinuierliche und hybride Petri-Netze.

In dieser Dissertation werden zwei Themen behandelt. Einerseits können Biologische Systeme aus vielen Gründen unbestimmt sein, zum Beispiel aus Umweltgründen, andererseits hatten farbige Petri-Netze in *Snoopy* Inkonsistenzen, die den Bau skalierbarer Modelle verhindern.

Die Ergebnisse der Fuzzy-Simulation sind sowohl das Fuzzy-Band jeder interessierenden Variablen als auch zeitliche Zugehörigkeitsfunktionen für jede Variable, damit man die Unsicherheiten erforschen kann, die mit der Eingabe verbunden sind. Durch die Harmonisierung von farbigen Petri-Netzen können wir skalierbare Modelle konstruieren und dann das Verhalten des Modells mit verschiedenen Größen des vorliegenden Modells untersuchen.

Freie Schlagwörter: Unbestimmte Systembiologie; Modellierung und Simulation; Quantitative Unbestimmte Petri-Netze; Farbige Quantitative Unbestimmte Petri-Netze; Software Hamonisierung.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

$\mathbb{N}$	Set of positive integer numbers
$\mathbb{N}_0$	Set of non-negative integer numbers including zero
$\mathbb{R}_0^+$	Set of non-negative real numbers including zero
$\emptyset$	Empty set
${}^\bullet t$	Pre-place(s) of the transition t
$t^\bullet$	Post-place(s) of the transition t
${}^\bullet p_i$	Pre-transition(s) of the place p_i
$p_i^\bullet$	Post-transition(s) of the place p_i
$m[t\rangle$	The transition t is enabled under the marking m
τ_0	Start simulation time
τ_{end}	End simulation time
$m(\tau_0)$	System's initial state
τ	Current time
$m(\tau)$	System's current state
$\delta\tau$	The next time a transition to occur (fire)
N	Number of chemical species (tokens)
M	Number of chemical reactions
$\overline{m}(\tau)$	Mean system's state
P_{disc}	Set of discrete places
P_{cont}	Set of continuous places
T_s	Set of stochastic transitions
T_i	Set of immediate transitions
T_d	Set of deterministically delayed transitions
T_{sched}	Set of scheduled transitions
T_{cont}	Set of continuous transitions
A_{cont}	Set of continuous arcs
A_{disc}	Set of discrete arcs
A_T	Set of read arcs
A_I	Set of inhibitor arcs
A_E	Set of equal arcs
A_R	Set of reset arcs
A_M	Set of modifier arcs
Σ	Set of colour sets
$C(p)$	Colour set of the place p

EXP	Colour expression
$C(p)_{MS}$	Multiset expression
$p(c)$	Place instance
$t(b)$	Transition instance
$B(t)$	Bindings of a set of variables of the transition t
$g(t)$	guard of the transition t
I_t	Set of all transition instances
I_p	Set of all place instances
$\mathbb{X}$	Universal set
α	Alpha level (membership degree)
$\alpha - cut$	Alpha cut set of certain alpha level
$\tilde{\xi}$	A fuzzy set
$\mu_{\tilde{\xi}}$	membership function of a fuzzy set
$\tilde{\xi}_\alpha$	membership degree at certain alpha level α
Γ	Set of fuzzy kinetic parameters
θ	Kinetic parameter

List of Abbreviations

ANDL	Abstract Net Description Language
BDD	Binary Decision Diagrams
CANDL	Coloured Abstract Net Description Language
CLI	Command Line Interface
$\mathcal{CPN}$	Continuous Petri Nets
$\mathcal{CPN}^{\mathcal{C}}$	Coloured Continuous Petri Nets
CSP	Constraint Satisfaction Problem
CTMC	Continuous Time Markov Chain
DAG	Directed Acyclic Graphs
$\mathcal{FCPN}$	Fuzzy Continuous Petri Nets
$\mathcal{FCPN}^{\mathcal{C}}$	Coloured Continuous Fuzzy Petri Nets
$\mathcal{FHPN}$	Fuzzy Hybrid Petri Nets
$\mathcal{FHPN}^{\mathcal{C}}$	Coloured Hybrid Fuzzy Petri Nets
FIS	Fuzzy Inference System
$\mathcal{FPN}$	Fuzzy Petri Nets
$\mathcal{FPN}^{\mathcal{C}}$	Coloured Fuzzy Petri nets
$\mathcal{FSPN}$	Fuzzy Stochastic Petri Nets
$\mathcal{FSPN}^{\mathcal{C}}$	Coloured Stochastic Fuzzy Petri Nets
GCD	Greatest Common Divisor
GSPN	Generalised Stochastic Petri nets
GUI	Graphical User Interface
$\mathcal{HPN}$	Hybrid Petri Nets
$\mathcal{HPN}^{\mathcal{C}}$	Coloured Hybrid Petri Nets
IDD	Interval Decision Diagrams

LHS	Latin Hypercube Sampling
ODE	Ordinary Differential Equations
$\mathcal{PN}$	Petri Nets
$\mathcal{PN}^{\mathcal{C}}$	Coloured Petri Nets
$\mathcal{QFPN}$	Quantitative Fuzzy Petri nets
$\mathcal{QFPN}^{\mathcal{C}}$	Coloured Quantitative Fuzzy Petri nets
$\mathcal{QPN}$	Quantitative Petri Nets
$\mathcal{QPN}^{\mathcal{C}}$	Coloured Quantitative Petri Nets
RNG	Random Number Generator
$\mathcal{SPN}$	Stochastic Petri Nets
$\mathcal{SPN}^{\mathcal{C}}$	Coloured Stochastic Petri Nets
SSA	Stochastic Simulation Algorithms
TFN	Triangular Fuzzy Number
UA	Uncertainty Analysis
$\mathcal{XPN}$	Extended Petri Nets
$\mathcal{XPN}^{\mathcal{C}}$	Extended Coloured Petri Nets
$\mathcal{XSPN}$	Extended Stochastic Petri nets

1 Introduction

1.1 Motivation

Petri nets are an attractive approach to model biological systems. Instead of directly dealing with mathematical formulas to describe systems, Petri nets offer an easy and intuitive way to graphically develop models in a well-organised and tidy way. They come generally in many flavours, ranging from uncoloured and qualitative Petri nets to coloured and quantitative ones. Among these flavours, (coloured) quantitative Petri nets [MRH12, LH14, BHM15] are interesting to study and analyse the behaviour of many biological phenomena by introducing the notion of time. Our Petri nets tool Snoopy [HHL+12] is one of the well-know tools supporting many types of Petri nets for teaching and research purposes.

One of the critical problems accompanying the modelling of biological systems is to determine the kinetic parameters (reaction rate parameters), as biological models generally come with a large number of kinetic parameters [KDS09]. In most cases, such kinetic parameters can not be directly measured, as experiments typically measure concentrations rather than rates [VLG+18]. Even when such parameters can be measured directly, this is usually in experimental conditions that are significantly different from the cellular environment we wish to study [VLG+18].

Uncertainty analysis is performed to investigate the uncertainty in the model output that is generated from uncertainty in parameter inputs [MHRK08]. A model is called deterministic, if the output of the model is completely determined by the input parameters and structure of the model. The same input will produce the same output if the model were simulated multiple times. Therefore, the only uncertainty affecting the output is generated by input variation. This type of uncertainty is termed epistemic (or subjective, reducible, type B uncertainty), which derives from a lack of knowledge about the adequate value for a parameter that is assumed to be constant throughout model analysis [HJSS06]. In contrast, a stochastic model will generally not produce the same output when repeated with the same inputs because of inherent randomness in the behaviour of the system. This type of uncertainty is termed aleatory (or stochastic, irreducible, type A) [HJSS06]. Exploring uncertainties has been and still is an area of interest and study in the field of systems biology.

Fuzzy logic is a powerful approach to deal with imprecise knowledge of, e.g. uncertain kinetic parameters by representing each kinetic parameter as a connected set of possible values (each value can be interpreted differently to describe the associated

uncertainties), rather than one single value (crisp value). Furthermore, by applying the Zadeh's extension principle [Zad65] we can obtain output bands of the variables of interest describing the uncertainties associated with the input. More importantly, we can also obtain fuzzy timed-membership functions of an output variable, which give more accurate information about the associated uncertainties [LHG20].

Combining (coloured) quantitative Petri nets with fuzzy logic (or fuzzy Petri nets for short) offers a new quality in user support with sophisticated modelling and analysis features [AHL21a], as two main issues can be addressed: the first is to model systems with repeated structures by means of colours, whereas the second is to model uncertain kinetic parameters by means of fuzzy sets. As we noticed from the literature, there is no tool except Matlab [Mat] dedicated to model and simulate uncertain biological models. However, Matlab may be a good choice for achieving this purpose, but it is not easy to be used, especially by non-informaticians, e.g. biologists. Therefore, supporting fuzzy Petri nets in Snoopy [HHL$^+$12] offers a graphical and an easy-to-modify way to develop biological models with uncertain kinetic parameters, which is one of the contributions that we present in this thesis.

Coloured Petri nets [Liu12, LHR12a] as they are supported by Snoopy so far have some inconsistencies, which are related to user-defined declarations including constants, colour functions, observers, and colour expressions. For example, previously, it was not possible to use constants as scaling factors in the model on hand, which means that a modeller could just re-define constants in order to scale the model. Moreover, some models need to use some operations for colour expressions that were not implemented. Fixing all these shortcomings will increase the modelling capabilities of coloured Petri nets.

The coloured abstract net description language (*CANDL*) is a human and machine readable format for different types of coloured Petri nets [ACR$^+$21]. *CANDL* is used as coloured Petri net exchange format among the family of Petri net tools *PetriNuts* [Pet]. Therefore, our second contribution is to unify the usage of coloured Petri nets in our tools, which will be achieved, as we will see later, by harmonising the coloured Petri nets with their Snoopy's uncoloured counterparts and extending them by all the features which are supported by the *CANDL* format. This step has the advantage of overcoming all the existing inconsistencies in Snoopy as well as unifying the usage of coloured Petri nets in the *PetriNuts* family of Petri net tools, e.g. *Marcie* [SRH11] and Charlie [HSW15]. Software harmonising [WVR$^+$12] is a branch of software engineering, which is crucial for bridging the gap between the software technical design and its implementation.

1.2 Objectives and Contributions

This section outlines the objectives and contributions of this thesis. The objectives are:

- Supporting the modelling of complex biological systems with uncertain kinetic parameters using the Petri net modelling approach.
- Supporting the simulation of uncertain biological systems in order to describe the uncertainties in output which reflect the uncertainties associated with input.
- Harmonising coloured Petri nets with their uncoloured counterparts to be able to model, animate and simulate scaleable models.
- Increasing the expressive power of coloured Petri nets by supporting new operations/features.

The list of contributions sketched as follows:

- **Modelling uncertainties associated with kinetic parameters**

  Modelling and analysing of uncertainties in biological systems is still a challenge, as precise kinetic parameters are not always available due to various reasons. Thus, the ability to model uncertain (fuzzy) kinetic parameters and then analyse the corresponding output will have many advantages for better understanding biological phenomena. A fuzzy set is a set whose elements have degrees of membership, and thus it can represent uncertain parameters by associating each uncertain kinetic parameter with a fuzzy number, e.g. triangular fuzzy number [Zim10]. Extending quantitative Petri nets by fuzzy kinetic parameters will increase their expressive power by allowing the kinetic parameters to be modelled either as fuzzy numbers or crisp values.

- **Colouring fuzziness for systems biology**

  Coloured Petri nets are excellent for modelling systems with repeated components by means of colours. This has advantages for modelling large and scaleable systems in a compact fashion. Therefore, combining coloured Petri nets with fuzzy logic yields a new powerful modelling approach, especially for modelling both uncertainty and systems with similar components.

- **Fuzzy simulation**

  Modelling uncertain kinetic parameters for biological systems is one side of the coin. Fuzzy simulation is crucial for uncertainty analysis, which gives for each output of interest a fuzzy band describing the uncertainties associated with the input. Furthermore, much more accurate information can be obtained by generating for each output variable its membership functions over simulation time.

- **Efficient sampling** Fuzzy simulation requires high computational and memory resources, as each fuzzy kinetic parameter has to be discretised into a set of continuous ranges, each called α-cut (or α level), and then to perform simulation for each separated sample (crisp value) [Zad65, Zim10]. For this purpose, sampling strategies have to be utilised. Sampling each α-cut set would probably cause producing similar samples on every level, which obviously leads to perform not needed simulations. Thus an efficient sampling strategy has to be chosen for avoiding producing redundant samples. Overall, sampling strategies are crucial for performing the fuzzy simulation efficiently. On one hand, they discretise each fuzzy kinetic parameter into crisp values. On the other hand, they have an impact on the total number of simulations that have to be performed, thus on the total simulation time. It is important to support an efficient sampling strategy which minimizes the total number of simulation as much as possible.

- **Harmonising coloured Petri nets**

  The expressive power of coloured Petri nets comes from the fact, that they allow to define different kinds of declarations which are used in the same way as in programming languages. Firstly, constants are crucial for developing scaleable models as well as for assigning kinetic parameters for transition rates. Secondly, colour functions are useful for annotating the model with short-cut expressions making the final model easy to understand and tidy. Here it could happen that one function can be used (called) by other functions in a nested way, yielding a nice feature for using them in the model. Last but not least, observers are mathematical functions over model variables/transitions which are useful for observing the involved variables/transitions as the modelled system evolves over time.

  Moreover, there are some forms of colour expressions which are not allowed to be used for, e.g. initialising the coloured places. Finally, there is a need for using some new operations for colour expressions, which increase the expressive power of coloured Petri nets. Most of these mentioned features suffer from inconsistencies due to the lack of implementation. Overcoming these inconsistencies can be achieved by harmonising coloured Petri nets with their uncoloured counterparts and extending them by those which are supported by the *CANDL* format.

1.3 Organisation of the Thesis

- **Chapter 1: Introduction**

  Chapter 1 (this chapter) presents an overview of the overall thesis, the motivation behind this work, the objectives and contributions, and the organisation of the thesis.

- **Chapter 2: Background**

  Chapter 2 provides background information about modelling and simulating biological systems using (coloured) stochastic/continuous/hybrid Petri nets. We present for each kind of Petri net class its formal definition together with its simulation algorithm.

- **Chapter 3: Fuzzy Petri Nets**

  Chapter 3 introduces fuzzy Petri nets and recent related work. Then we present some basics for fuzzy logic and how to combine it with quantitative Petri nets in order to represent biological systems with uncertain kinetic parameters. After that, we introduce the fuzzy simulation algorithm together with three sampling strategies. Then, we present a set of case studies which can be modelled and simulated using the family of fuzzy Petri nets. Finally, we sketch some experimental results illustrating how much burden one should expect when utilising fuzzy simulation.

- **Chapter 4: Some Implementation Aspects**

  Chapter 4 presents some of the existing inconsistencies in Snoopy's coloured Petri nets, and some implementation aspects of the harmonising procedure. Furthermore, we present some new operations/feature that are useful for increasing the expressive power of coloured Petri nets.

- **Chapter 5: Conclusion and Outlook**

  Finally, this chapter concludes the thesis by summing up the overall presented information and proposes possible extensions for future work.

2 Background

2.1 Introduction

Petri nets originate from the dissertation of Carl Adam Petri [Pet62] who introduced the idea of place/transition nets. Petri nets ($\mathcal{PN}$ for short) are a powerful modelling approach, which has been used in many applications in various fields, including natural sciences, engineering sciences and life sciences [SRL+20]. These applications use Petri nets for both modelling and exploration of the model behaviour. For this purpose, Petri nets come generally with many analyse techniques, ranging from structural to behavioural analysis. Furthermore, some applications call for model animation and simulation. On one hand, model animation is helpful for exploring a model behaviour. On the other hand, model simulation (execution of the model) using its initial state and available kinetic parameters is crucial for predicting a model behaviour over time.

In this chapter, we are going to illustrate how to make use of Petri nets for modelling and simulating of systems, e.g. biological systems. For this purpose, we recall various types of Petri nets and their formal definitions, ranging from standard qualitative and quantitative Petri nets to coloured qualitative and quantitative ones. Section 2.2 presents qualitative Petri nets and the running example used to demonstrate all Petri net classes introduced in this chapter. Section 2.3 introduces extended Petri nets and their modelling features. In Section 2.4, we cover quantitative Petri nets, including stochastic, continuous and hybrid Petri nets. Here we recall in more details the semantic behind each one. in Section 2.5, we introduce coloured Petri nets and the advantages of colour information, especially for systems biology. In Section 2.6, we outline coloured quantitative Petri nets and their semantics. In order to be able to reuse the uncoloured Petri net analysis techniques, we describe unfolding of coloured Petri nets together with the unfolding algorithms in Section 2.7.

Please note that all definitions in this Chapter reflect the Petri net classes as supported by Snoopy [HHL+12].

2.2 Petri Nets

Petri nets are directed bipartite graphs, combining a well-defined mathematical theory with graphical elements for describing concurrent, asynchronous and parallel systems in an intuitive graphical notation. They basically comprise two types of nodes, transi-

tions (i.e. events that may occur) and places (i.e. local states) of a given system under consideration. Places and transitions are connected to each other by weighted arcs. The initial state of a system is specified by the initial marking of all places. In systems biology, places represent species; their total number or concentrations are specified by a discrete number of tokens or a real number of tokens residing in the places. Transitions represent biochemical reactions or transport steps that may occur, whereas arc weights help to deal with stoichiometries [AHL21a].

Petri nets are generally useful to perform some analysis of the studied model, ranging from structural to behavioural analysis, or even animating the model to gain some initial trust in the model behaviour. The formal definition of Petri nets is given as follows.

Definition 1 (Petri net [LH14])
A *Petri net* is a 5-tuple $N = \langle P, T, A, f, m_0 \rangle$, where:

- P is a finite, non-empty set of places.
- T is a finite, non-empty set of transitions.
- $P \cap T = \emptyset$.
- $A \subseteq (P \times T) \cup (T \times P)$ is a finite set of directed arcs.
- $f : A \to \mathbb{N}$ is a function that assigns a positive integer number as an arc weight to each arc $a \in A$. Please note that an arc weight is not allowed to be zero, because this would mean that there is no connecting arc between a place and a transition which does not respect the $\mathcal{PN}$ connectivity rules.
- $m_0 : P \to \mathbb{N}_0$, is a function that assigns a non-negative integer number to each place as an initial marking.

Please note that $\mathbb{N}$ and $\mathbb{N}_0$ denote the sets of positive and non-negative integer numbers including zero, respectively.

Each transition $t \in T$ may have one or more pre-places connected to it. These pre-places have an influence on the transition's enabledness. In the same way, each transition may be connected to one or more post-places, but they do not have an influence on the transition's enabledness. Both pre-places and post-places get generally affected by firing the enabled transition by changing their markings. The notations ${}^\bullet t$ and $t^\bullet$ denote the pre-places and post-places of the transition t, respectively. An enabling of a given transition is formally given as follows.

Definition 2 (Enabling condition)
Let $N = \langle P, T, A, f, m_0 \rangle$ be a Petri net, m a marking of N, and $t \in T$ a transition. The transition t is enabled in the marking m, denoted by $m[t\rangle$, if the following condition holds.

- $\forall p \in {}^\bullet t, m(p) \geq f(p,t)$, if $(p,t) \in A$.

If there is no enabled transition, the Petri net will be in a dead state. Once a transition gets enabled, it can be fired. The firing rule of an enabled transition is formally given as follows.

Definition 3 (Transition firing)
Let $N = \langle P, T, A, f, m_0 \rangle$ a Petri net, m be a marking of N, and $t \in T$ a transition. enabled in the marking m. The transition t can be fired and reach a new marking m', denoted by $m[t\rangle m'$, with

$$m'(p) = m(p) - f(p,t) + f(t,p).$$

It may happen that two transitions are enabled, but the firing of one transition disables the other one, i.e. a conflict occurs. The occurrence of dynamic conflicts indicates alternative (branching) system behaviour. To solve dynamic conflicts, decisions between alternatives are taken non-deterministically for the model animation, whereas typically one transition fires after the other one for model analysis.

In the following, we are going to introduce a well-known ecological system called Predator/Prey system [Lot09] (also called Lotka-Volterra) which will be used as a running example. The Lotka-Volterra system has been frequently used to describe the dynamics of biological systems. It comprises two kinds of species that interact with each other, one as predator and the other as prey. It is generally characterized by its oscillation behaviour, because the population of predator species, e.g. foxes, increases by consuming prey species, e.g. rabbits, which will be decreased consequently and vice versa.

Figure 2.1 gives a Petri net model of the Lotka-Volterra system. Both kinds of species are represented as places *Prey* and *Predator*, respectively. The number of tokens in each place (determined by two integer constants N and M) gives the initial state of the system. The model has three transitions representing the interactions among species. The transition *reproduce_prey* describes the reproduction of preys, whereas the transition *death_predator* describes the death of predators and the transition *consumption_of_prey* gives the interaction between preys and predators. Please note that the number of reproduced preys and predators is determined by the constant *Births* occurring as arc weight.

Let us discuss the possible behaviours of the predator/prey system using the corresponding Petri net model

- Firing the transition *death_predator* M times changes the number of tokens in the place *Predator* to zero. As a result, the transition *consumption_of_prey* will never occur and the transition *reproduce_prey* will never be disabled, thus it always occurs causing the number of prey species to explode.

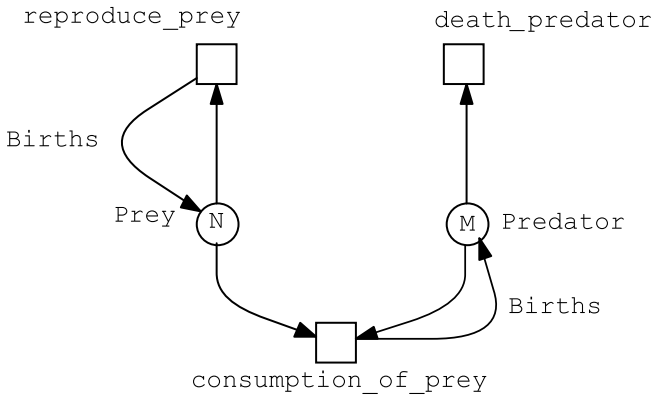


Figure 2.1: Lotka Volterra Petri net model in Snoopy; net elements: place ◯, transition □, standard arc ⟶. *Births, N* and *M* are integer constants. The latter two are used as initial marking of the places *Prey* and *Predator*, respectively, while the constant *Births* is used as an arc weight.

- Firing the transition *reproduce_prey* N times yields $N \cdot Births$ tokens in the place *Prey*. Assuming $M > N$, firing the transition *consumption_of_prey* N times changes the number of tokens in the place *Prey* to zero, and the number of tokens in the place *Predator* to $M - N + N \cdot Births$. Consequently, the number of tokens in the place *Prey* will never increase again. After that, the system will reach a dead state by firing the transition *death_predator* $M - N + N \cdot Births$ times.

- Let us assume $M < N$, firing the transition *consumption_of_prey* N times changes the number of tokens in the place *Prey* to zero and the number of tokens in the place *Predator* to $(N \cdot Births) - (N - M)$. After that, the transitions *consumption_of_prey* and *reproduce_prey* will never occur, thus firing the transition *death_predator* $(N \cdot Births) - (N - M)$ times will cause the system to reach a dead state.

2.3 Extended Petri Nets

Extended Petri nets [HLGM09] (or $\mathcal{XPN}$ for short) extend standard Petri nets by offering four special arcs, making a Petri net model more compact and increasing the modelling power of the standard Petri nets, but decreasing the analysis power. These

arcs are: inhibitory arcs, read (test) arcs, equal arcs and reset arcs. Please note that all these arcs are directed from a place to a transition; compare Figure 2.2.

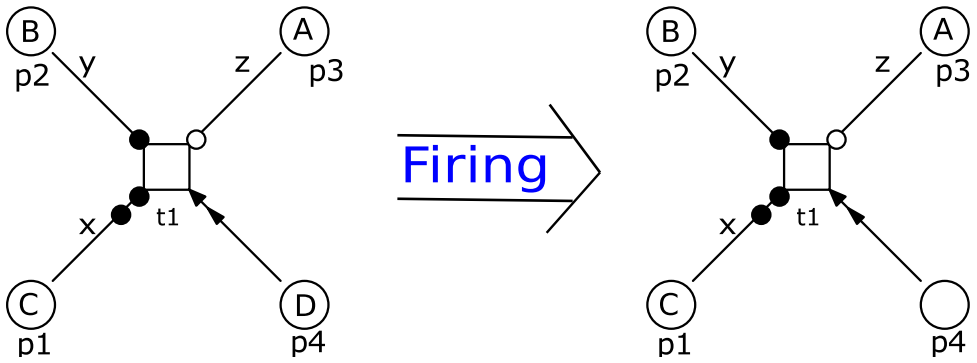


Figure 2.2: Special arcs in Snoopy's extended Petri net class: read (test) arc ——•; inhibitory arc ——○; equal arc ——•• and reset arc ——➝. A, B, C, D, x, y and z are constant integers; where the following conditions have to hold for enabling the transition $t1$: $x = C$, $y \leq B$ and $A < z$. Each condition corresponds to a specific type of the special arcs. Only the place *p4* gets affected by firing the transition *t1*, whereas the markings of the other places remain the same.

The read arcs, inhibitory arcs and equal arcs add constraints on firing a transition connected with them, based on arc weights and the marking value of the transition's pre-place. Moreover, these arcs do not have an influence on the marking of the pre-places upon firing the transition. They only control the firing of a transition by obeying the following rules:

- A transition connected with a place using a read arc may fire when the marking of the pre-place is equal or greater than the arc weight.
- A transition connected with a place using an inhibitory arc may fire when the marking of the pre-place is less than the arc weight. This means that the inhibitory arc inhibits the enabling of a transition when the marking of a place is greater than or equal to a certain threshold (arc weight of inhibitory arc).
- A transition connected with a place using an equal arc may fire when the marking of the pre-place is exactly equal to the arc weight.

The reset arc does not add any constraint to the firing of a transition, but it cleans the pre-place (changes its marking to zero) as soon as the connected transition fires. The formal definition of extended Petri nets is given as follows.

Definition 4 (Extended Petri net [HLGM09])
An *extended Petri net* is a tuple $N = \langle P, T, A, f, m_0 \rangle$ where:

- P is a finite, non-empty set of places.
- T is a finite, non-empty set of transitions.
- $P \cap T = \emptyset$.
- A is a finite set of directed arcs. A is defined as the union of five disjunctive arc sets, i.e. $A := A_S \cup A_I \cup A_T \cup A_E \cup A_R$ with:
  - $A_S \subseteq (P \times T) \cup (T \times P)$ is the set of standard arcs,
  - $A_I \subseteq P \times T$ is the set of inhibitory arcs,
  - $A_T \subseteq P \times T$ is the set of test/read arcs,
  - $A_E \subseteq P \times T$ is the set of equal arcs, and
  - $A_R \subseteq P \times T$ is the set of reset arcs.
- $f : A \to \mathbb{N}$ is a function that assigns a positive integer to each arc $a \in F \setminus A_R$, which means that all arc types take a non-negative integer as an arc weight, except reset arcs.
- $m_0 : P \to \mathbb{N}_0$ gives the initial marking.

In comparison with standard Petri nets, the enabling conditions are formally given as follows.

Definition 5 (Extended enabling condition)
Let $N = \langle P, T, A, f, m_0 \rangle$ be an extended Petri net and m be a marking of N. A transition $t \in T$ is enabled in the marking m, denoted by $m[t\rangle$, if the following conditions are satisfied.

- $\forall p \in {}^\bullet t$, $m(p) \geq f(p,t)$, if $(p,t) \in A_S$,
- $\forall p \in {}^\bullet t$, $m(p) < f(p,t)$, if $(p,t) \in A_I$,
- $\forall p \in {}^\bullet t$, $m(p) \geq f(p,t)$, if $(p,t) \in A_T$,
- $\forall p \in {}^\bullet t$, $m(p) = f(p,t)$, if $(p,t) \in A_E$.

Definition 6 (Extended firing)
Let $N = \langle P, T, A, f, m_0 \rangle$ be an extended Petri net, m be a marking of N, and t a transition $t \in T$ enabled in the marking m. The transition t can be fired and reach a new marking m', denoted by $m[t\rangle m'$, with

$$m'(p) = \begin{cases} m(p) - f(p,t) + f(t,p) & \text{if } (p,t) \in A_S \text{ and } f(t,p) \in A_S, \\ m(p) + f(t,p) & \text{if } (p,t) \in A_I \text{ and } f(t,p) \in A_S, \\ m(p) + f(t,p) & \text{if } (p,t) \in A_T \text{ and } f(t,p) \in A_S, \\ m(p) + f(t,p) & \text{if } (p,t) \in A_E \text{ and } f(t,p) \in A_S, \\ f(t,p) & \text{if } (p,t) \in A_R \text{ and } f(t,p) \in A_S. \end{cases}$$

Adding special arcs to our running example given in Figure 2.1 gives the extended Petri net shown in Figure 2.3. The production of preys will be inhibited when the number of preys reaches the upper bound specified by the constant *Limit*; this means that the place *Prey* is bounded by *Limit*-many tokens. Please note that enabling the transition *reproduce_prey* requires at least two tokens in the place *Prey* (specified by the weight of the connected test arc). The transitions *r_prey* and *r_pred* have been added to the model in order to reset the system to its initial state; for this purpose, we make use of reset arcs together with inhibitory arcs. In this model, both transitions *r_prey* and *r_pred* describe an alternative behaviour as firing one of them precludes the other one, because firing one of these transitions will reinitialise both places *Prey* and *Predator* and thus the other place will have a token number larger than the weight of the connected inhibitory arc (here is one). Reset arcs are utilised to ensure that the marking of the places *Prey* and *Predator* will not get doubled upon firing one of these two transitions. Compared to the basic Lotka Volterra system given in Figure 2.1, the extended version will not reach a dead state because of the *Reset* mechanism.

2.4 Quantitative Petri Nets

Quantitative Petri nets ($\mathcal{QPN}$) are an extension of standard time-free Petri nets (also called qualitative Petri nets) by a notion of time [GHL07, BHM15]. Compared to qualitative Petri nets, $\mathcal{QPN}$ associate each transition with a rate function, which is an arbitrary mathematical function.

Firing rates are typically state-dependent, which means a transition's pre-places are allowed to occur as variables in rate functions. This rule has been set up to prevent that net structure and rate functions diverge [BHM15]. Furthermore, rate functions often follow certain kinetic patterns which have biological interpretations, such as mass/action kinetics, and involve kinetic parameters (constants).

The rate functions of the models presented in this thesis follow the pattern *Mass-Action(k)*, whose formula is given as follows.

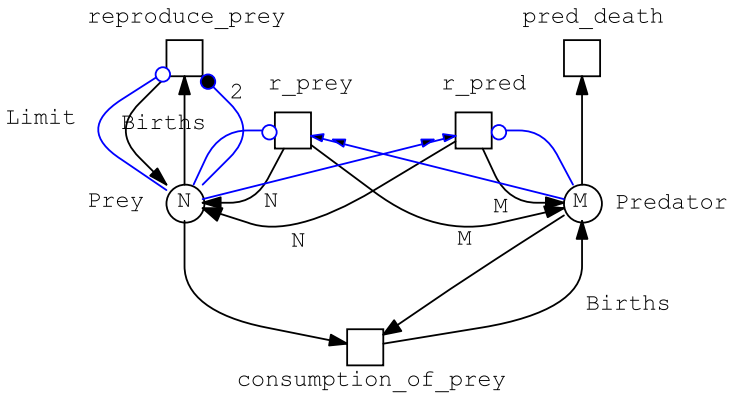


Figure 2.3: The Lotka Volterra system modelled as an extended Petri net. The extended version permits resetting the system to its initial state and avoids accumulation of tokens in the place *Prey*.

$$k_j \prod_{l=1}^{N_j} [S_l]^{\alpha_{jl}}, \tag{2.1}$$

where k_j is the kinetic parameter of the reaction j, N is the number of reactant species in reaction j, and α_{jl} is the stoichiometric coefficient of reactant species S_l.

Please note that the default rate function of a transition in Snoopy's $\mathcal{QPN}$ is *MassAction(1)*, i.e. mass-action kinetics with parameter 1. Besides, Snoopy provides a special feature supporting state-dependent firing rates which is a modifier arc, represented as a dashed arc, which always goes from a place to a transition and allows to use this place in the transition's rate function even if it is not a standard pre-place. Please note that the modifier arc does not have an influence on the transition's enabledness (contrary to $\mathcal{XPN}$'s special arcs). For an example compare Figure 2.5.

Rate functions can be interpreted in different ways and we obtain - depending on the interpretation - stochastic Petri nets ($\mathcal{SPN}$), continuous Petri nets ($\mathcal{CPN}$), or hybrid Petri nets ($\mathcal{HPN}$) [HH14].

2.4.1 Stochastic Petri Nets

Stochastic Petri nets basically share the structure of time-free Petri nets which means that each place and arc get assigned a discrete number of tokens or arc weight, respectively.

In contrast to standard time-free Petri nets, each transition is associated with a stochastic firing rate function determining a stochastic waiting time before an enabled

transition actually fires [HH18b]. The definition of stochastic Petri nets is formally given as follows.

Definition 7 (Stochastic Petri nets ($\mathcal{SPN}$) [ROH17])
Stochastic Petri nets are a 6-tuple $\mathcal{SPN} = \langle P, T, A, F, V, m_0 \rangle$ where:

- P is a finite, non-empty set of discrete places.
- $T = T_s$ is the set of stochastic transitions, which fire stochastically after an exponentially distributed waiting time.
- $P \cap T = \emptyset$.
- $A = A_S \cup A_M$ is the set of directed arcs, with:
  1. $A_S \subseteq (P \times T) \cup (T \times P)$ defines the set of standard arcs.
  2. $A_M \subseteq (P \times T)$ defines the set of modifier arcs.
- $F : A \to \mathbb{N}$ is a function which assigns a positive integer number to each arc in $A_S \setminus A_M$, which means that modifier arcs are not allowed to have arc weights.
- $V : T \to H$ is a function which assigns a firing rate function h_t to each transition $t \in T$, whereby $H = \{h_t | h_t : \mathbb{R}_0^{+|^\bullet t|} \to \mathbb{R}_0^+, t \in T\}$ is the set of all firing rate functions, and $V(t) = h_t, \forall t \in T$.
- $m_0 : P \to \mathbb{N}_0$, is a function which assigns a non-negative integer number to each place as the initial marking.

Please note that the notation $\mathbb{R}_0^+$ denotes the set of non-negative real numbers.

Traditional $\mathcal{SPN}$ have been extended by $\mathcal{XPN}$'s special arcs given in Definition 4 and a special type of stochastic transitions called *immediate* transition yielding Generalised Stochastic Petri nets ($\mathcal{GSPN}$ for short). An *immediate* transition has the highest firing priority as its stochastic waiting time is always zero. To control the decision of conflicts, each immediate transition is associated with a transition weight. Thus, the transition with the largest weight has the highest firing priority. If there are two immediate transitions with the same weight, then the firing decision is taken in a nondeterministic way. $\mathcal{GSPN}$ is formally defined as follows [ROH17].

Definition 8 (Generalised stochastic Petri nets [ROH17])
Generalised stochastic Petri nets are a 6-tuple $\mathcal{GSPN} = \langle P, T, A, F, V, m_0 \rangle$ where:

- P is a finite and non-empty set of discrete places.
- $T = T_s \cup T_i$, is a finite and non-empty set of transitions with:
  1. T_s is the set of stochastic transitions, which fire stochastically after an exponentially distributed waiting time.
  2. T_i is the set of immediate transitions, which fire with waiting time zero; they have higher priority compared with other transition types.
- $P \cap T = \emptyset$.
- $A = A_S \cup A_I \cup A_T \cup A_E \cup A_R \cup A_M$ is the set of directed arcs, defined in the same way as in Definition 4 .
- $F : A \to \mathbb{N}$ is a function which assigns a positive integer number to each arc, except reset and modifier arcs.
- $V = V_s \cup V_i$, is a set of functions with:
  1. $V_s : T_s \to H$ is a function which assigns to each stochastic transition a stochastic rate function $V_s(t) = h_{ts}$.
  2. $V_i : T_i \to \mathbb{R}_0^+$ is a function which assigns to each immediate transition a non-negative real value as weight.
- $m_0 : P \to \mathbb{N}_0$, is a function which assigns a non-negative integer number to each place as the initial marking.

$\mathcal{GSPN}$ have been further extended by two types of transitions: *deterministic* and *scheduled* yielding extended stochastic Petri nets ($\mathcal{XSPN}$ for short) [HLGM09]. A deterministic transition fires after a deterministic firing delay (waiting time). The delay is always relative to the time point where a transition gets enabled. Deterministic transitions are helpful to minimize the $\mathcal{PN}$ size, by replacing a sequence of stochastic transitions with a deterministic one; with the delay amount set to the sum of those stochastic ones. Scheduled transitions belong to the deterministic transitions; the deterministic firing occurs according to a schedule specifying absolute points of the simulation time. A schedule can specify just a single time point, or equidistant time points within a given interval triggering the firing once or periodically. To specify the firing interval, a scheduled transition has three arguments which have to be specified: the beginning and end of the interval and the repetition value which determines the distance between each firing in this interval. Setting both the beginning and end time points to the same value will cause the transition to fire once (at the specified time

point). Assuming the following interval $[5, 1, 50]$, the scheduled transition will fire 45 times starting from the time point 5, till the time point 50. In contrast, the values $[50, x, 50]$ will cause the transition to fire at the time point 50, and the repetition value can be ignored. However, transitions only fire at their scheduled time points if they are enabled.

Definition 9 (Extended stochastic Petri nets [ROH17])
Extended stochastic Petri nets are a 6-tuple $\mathcal{XSPN} = \langle P, T, A, F, V, m_0 \rangle$ where:

- P is a finite and non-empty set of discrete places.
- $T = T_s \cup T_i \cup T_d \cup T_{sched}$ is a finite and non-empty set of transitions with:
  1. T_s is the set of stochastic transitions, which fire stochastically after an exponentially distributed waiting time.
  2. T_i is the set of immediate transitions, which fire with waiting time zero; they have higher priority compared with other transition types.
  3. T_d is the set of deterministically delayed transitions, which fire after a deterministic time delay.
  4. T_{sched} is the set of scheduled transitions, which fire at predefined time points.
- $P \cap T = \emptyset$.
- $A = A_S \cup A_I \cup A_T \cup A_E \cup A_R \cup A_M$ is the set of directed arcs, defined in the same way as in Definition 4 .
- $F : A \to \mathbb{N}$ is a function which assigns a positive integer number to each arc, except reset and modifier arcs.
- $V = V_s \cup V_i \cup V_d \cup V_c$, is a set of functions with:
  1. $V_s : T_s \to H$ is a function which assigns to each stochastic transition a stochastic rate function $V_s(t) = h_{ts}$.
  2. $V_i : T_i \to \mathbb{R}_0^+$ is a function which assigns to each immediate transition a non-negative real value as weight.
  3. $V_d : T_d \to \mathbb{R}_0^+$ is a function which assigns to each deterministic transition a non-negative deterministic waiting time.
  4. $V_c : T_{sched} \to H$ is a function which assigns to each scheduled transition three real values representing the beginning of the firing interval, the repetition value, and the end of the firing interval; respectively.
- $m_0 : P \to \mathbb{N}_0$, is a function which assigns a non-negative integer number to each place as the initial marking.

Figure 2.4 gives the graphical representation of $\mathcal{XSPN}$'s transitions as they are supported in Snoopy.

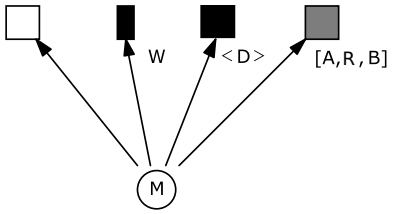


Figure 2.4: Graphical representation of $\mathcal{SPN}$ transitions in Snoopy: stochastic, immediate, deterministic and scheduled; ordered from left to right. M is an integer constant used as initial marking, whereas W, D, A, R and B are double constants representing the weight of the immediate transition, time delay of the deterministic transition and the start time point, repetition value and end time point for the scheduled transition, respectively.

We are going to illustrate stochastic Petri nets using the Lotka Volterra system comprising four reactions. Species are represented as discrete places, while each reaction is represented as a stochastic transition getting assigned a stochastic rate function. Modifier arcs are used to allow the places (*prey* and *Predator*) to occur in the rate functions of the transitions *consumption_of_prey1* and *consumption_of_prey2*, respectively. Compare Figure 2.5. The populations change over time according to a pair of equations 2.2 and 2.3 [KHR10], [Wik21b]:

$$\frac{dx}{d\tau} = \alpha x - \beta xy, \tag{2.2}$$

$$\frac{dy}{d\tau} = \delta xy - \gamma y, \tag{2.3}$$

where x is the number of preys, y is the number of predators, α, β, δ and γ are positive real parameters which can be interpreted as follows: α is the growth rate constant of preys, β is the death rate constant of preys due to, e.g. being killed (eaten) by predators, δ is the growth rate constant of predators and γ is the death rate constant of predators. In order to explain the modelling of standard Petri nets, we assume that $\beta = \delta$.

Stochastic Simulation

The underlying semantic of an $\mathcal{SPN}$ is defined by a *Continuous Time Markov Chain* ($\mathcal{CTMC}$) [HGD08]. Setting up the *CTMC* of a given $\mathcal{SPN}$ may be infeasible as the state space can be very large or even infinite. In order to approximate the *CTMC*, we

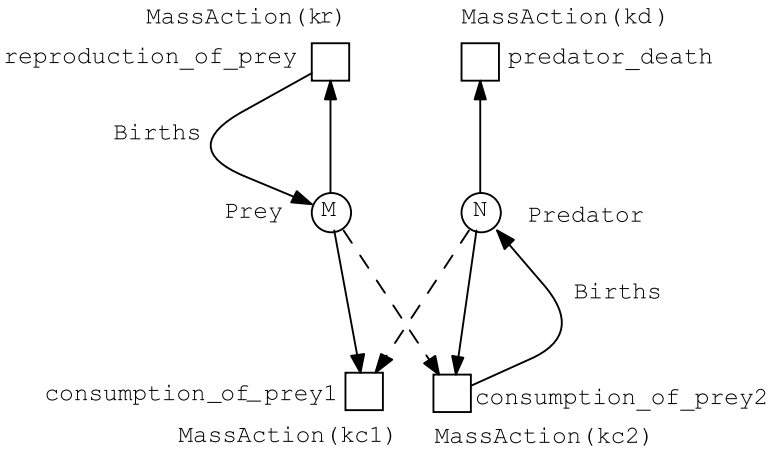


Figure 2.5: Original Lotka Volterra system represented as $\mathcal{SPN}$. N, M and $Births$ are integer constants, while k_r, k_{c1}, k_{c2} and k_d are double constants (kinetic parameters), representing α, β, δ and γ, respectively.

simulate an $\mathcal{SPN}$ model by generating different paths through it instead of computing the $CTMC$ directly [ROH17].

A path of the CTMC can be generated in the following way. We start from the initial marking m_0, then transitions have to be fired repeatedly. Here arise two questions:

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

The probability density function 2.4 [Gil76] answers these two questions which is given as follows.

$$\begin{aligned} P(\tau, t_j|m)d_\tau \equiv \text{probability at given state } m(\tau) \\ \text{that transition } t_j \text{ will fire in} \\ \text{the next time interval } [\tau, \tau + \delta\tau), \end{aligned} \tag{2.4}$$

where $\tau + \delta\tau$ is the next time at which the transition t_j will fire.

An $\mathcal{SPN}$ model can be simulated using one of the stochastic simulation algorithms (SSA). Gillespie's stochastic simulation algorithm is one of the most popular stochastic simulation algorithms (also knows as direct method) [Gil76, Gil77].

Gillespie's SSA algorithm (also called direct method) generates random walks through the CTMC. Algorithm 2.1 gives the basic idea. Based on the basic algorithm, some improvements were considered e.g. in [Her13, ROH17]. The algorithm takes an $\mathcal{SPN}$ model

to be simulated together with the simulation time as the input and returns the output trace of stored system states (one possible path through the CTMC). First of all, some initialisation steps are required, the current time (τ) is initialised with the start simulation time (line 2), the current state $m(\tau)$ is initialised with the initial state of the model (line 3) and the output with the current system state (line 4). Then, the algorithm goes in a loop (which stops when the current time reaches the end simulation time τ_{end}), and for each iteration, it executes the following. It computes the duration until the next transition will fire (line 6) and advances the current time by this values (line 7). Afterwards, the transition t to be fired is selected depending on the current system state (line 10). As soon as the transition is selected, it fires and the system state is updated accordingly (line 9). Finally, the output trace is updated with the current system state (line 10). A reliable insight into the system behaviour is only possible if the system states of several runs are examined. Each run starts from the same initial state, in which a random generator is initialised with a random seed (line 1). This allows for various runs of a stochastic process. The system state at time point τ of each simulation run is recorded and the mean state at this point is given by Equation 2.5.

$$\overline{m}(\tau) = (1/N) \sum_{n=1}^{N} m(n, \tau), \tag{2.5}$$

where N is the number of simulation runs to be averaged.

Algorithm 2.1: Direct simulation method [ROH17].

Input: An $\mathcal{SPN}$ model with its initial state $m(\tau_0)$;
simulation interval $[\tau_0, \tau_{end}]$;
Output: stochastic simulation trace over time.

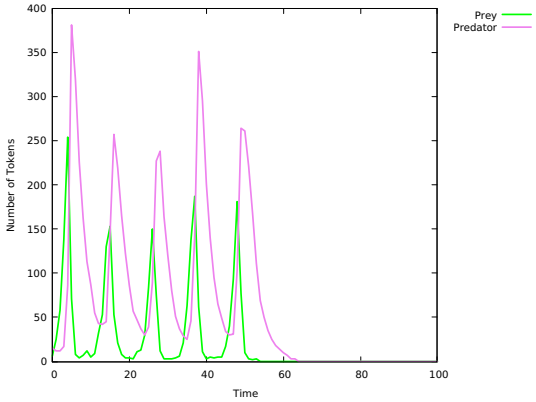
1: initRandom(seed);
2: time $\tau = \tau_0$; /* initialise the current time with the start simulation time */
3: state $m(\tau) = m(\tau_0)$; /* assign the initial state to the current state * /
4: $m(\tau) \rightarrow store$; /* add the current state to the output trace */
5: **while** $\tau \leq \tau_{end}$ **do**
6: $\delta\tau$ = determine duration until next firing by computing rate function h of all transitions depending on the current state $m(\tau)$;
7: $\tau = \tau + \delta\tau$; /* determine the next time point */
8: t = select the next transition to be fired depending on the current state $m(\tau)$;
9: $m(\tau) = t \rightarrow fire$; /* fire the transition and update the system state accordingly */
10: $m(\tau) \rightarrow store$; /* add $m(\tau)$ to the output trace */
11: **end while**

Figure 2.6 presents some stochastic simulation results of the original Lotka Volterra system (Figure 2.1). Sub-figure 2.6a illustrates how the system sooner or later will reach the dead state. Sub-figure 2.6b gives the averaged simulation over twenty runs. Sub-figure 2.6c demonstrates how the population of preys explodes over time, because the transition *death_predator* occurs more often than the transition *reproduce_prey* causing the predator population to be dropped rapidly to zero before the prey population becomes extinct; which means the system will never reach the dead state. Please note that for this figure, we stopped plotting the curve at time 50 because after that time the prey population will keep exponentially growing in a way that is not numerically traceable. The same discussion is applied for Sub-figure 2.6d presenting the averaged simulation over five runs.

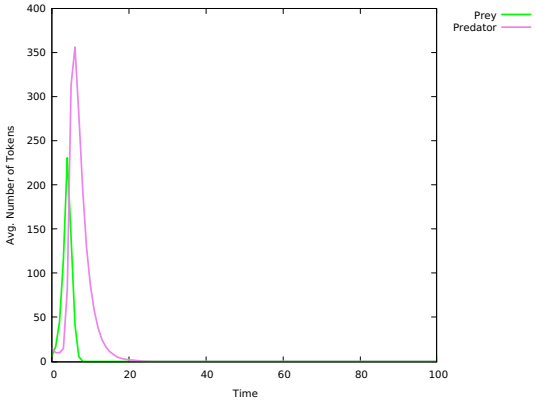
Figure 2.7 shows stochastic simulation results for the original Lotka Volterra system shown in 2.1, whereby the growth rate constant of preys is equal to the death rate constant of predators with four scenarios (by changing the initial marking of the model places). Sub-figure 2.7a shows that prey species explode, as the transition *death_predator* had the chance to fire more often than the transition *reproduce_prey* causing predator species to be extinct very quickly before prey species. This enables prey species to multiply over time. Please note in this scenario we stopped the simulation at time 20 due to the same reason mentioned previously. The other sub-figures show that prey species become extinct earlier than predator species.

Figure 2.8 presents stochastic simulation results of the extended version of our running example (Figure 2.3). In Figure 2.6 we notice that the system always reaches a dead state as preys will be consumed by predators and predators will be killed by, e.g. hunters (model shown in Figure 2.1). In contrast, Figure 2.8 obviously shows that the dead state is never reached due to the reset mechanism, according to the model shown in Figure 2.3.

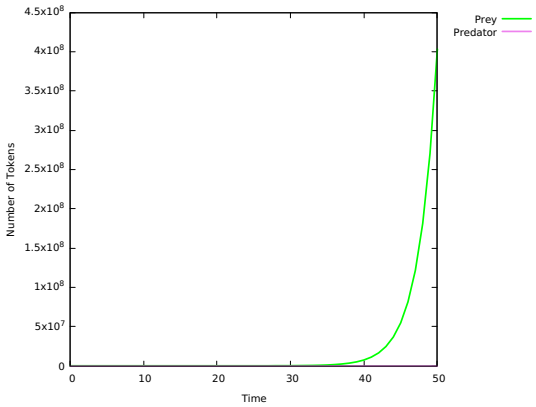
Figure 2.9 gives stochastic simulation results of the extended version of the Lotka Volterra system. Here we give simulation traces of the transitions *r_prey* and *r_pred*. The occurrence of one of them describes the re-initialisation of the system. Interestingly, the transition *r_pred* occurs all the time, because the rate of the transition *pred_death* is higher than the rate of the transition *reproduce_prey*; thus the place *Predator* gets clean (zero tokens) before the place *Prey*, and this causes the transition *r_pred* to fire and re-initialisate the system. Please note that if the transition *reproduce_prey* has a higher rate, then this transition will occur all the time due to the same reason.



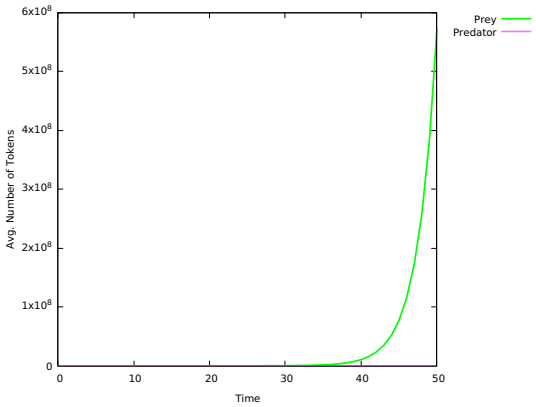
(a) single simulation run with kr=1.1, kc=0.01 and kd=0.4.



(b) averaged simulation results over 20 runs.

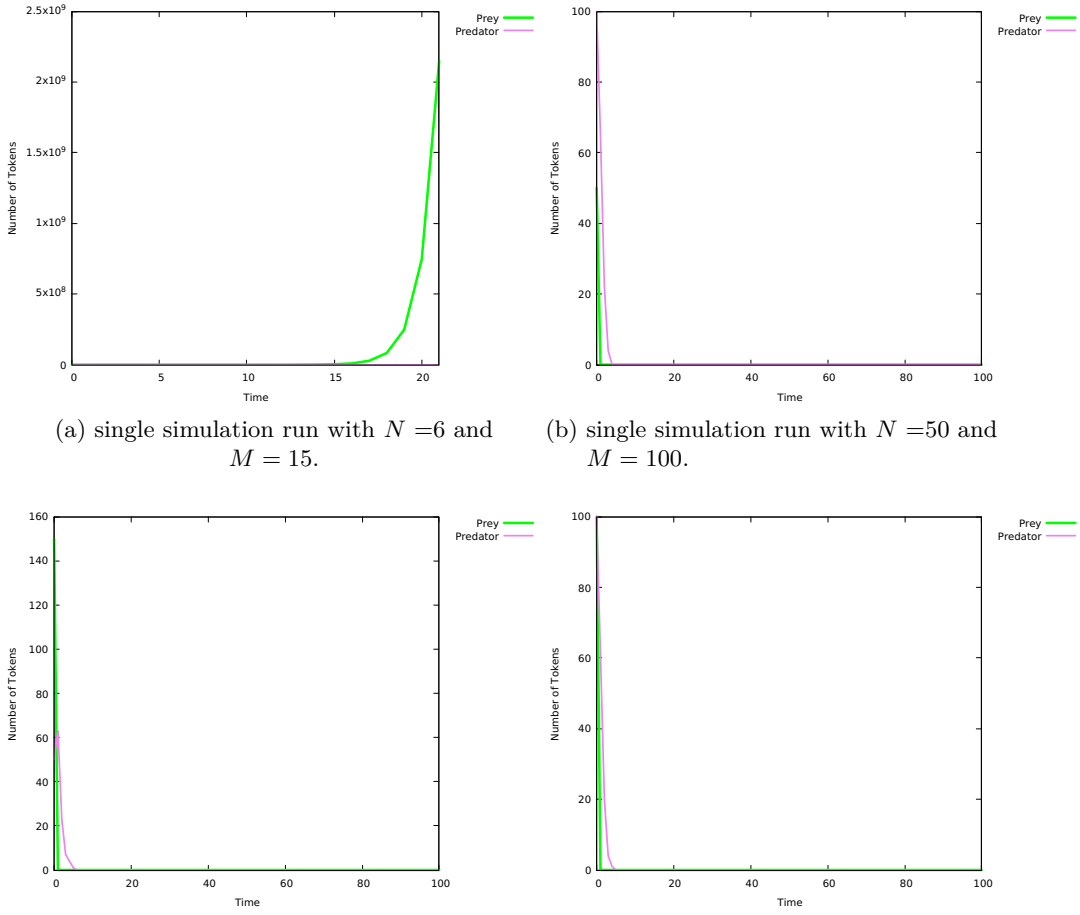


(c) single simulation run with kr=0.4, kc=0.1 and kd=1.1.



(d) averaged simulation results over 5 runs.

Figure 2.6: Stochastic simulation results of the original version of the Lotka Volterra system shown in Figure 2.1. The constants N, M and $Births$ have the values 5, 16 and 2, respectively. The kinetic parameters kr, kc and kd are assigned to the transitions *reproduce_prey*, *consumption_of_prey* and *death_predator*, respectively. Sub-figures (a) and (b) show that the system reaches a dead state in comparison with Sub-figures (c) and (d), due to the kinetic parameter values.



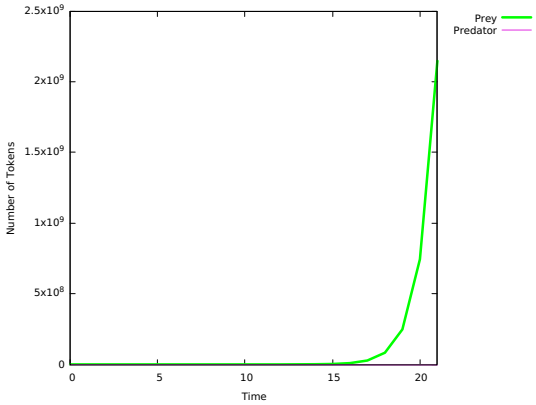
(a) single simulation run with N =6 and $M = 15$.

(b) single simulation run with N =50 and $M = 100$.

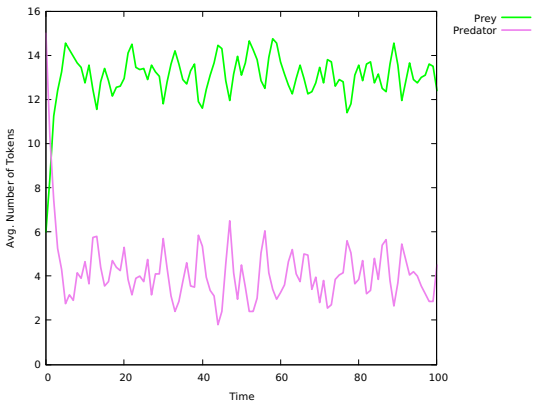
(c) single simulation run with N =150 and $M = 50$.

(d) single simulation run with N =100 and $M = 100$.

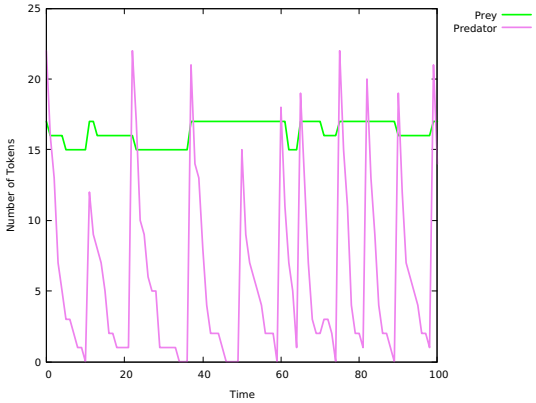
Figure 2.7: Stochastic simulation results of the original version of the Lotka Volterra system shown in Figure 2.1. The constant *Births* has the value 2. The kinetic parameters $kr = kd = 1.1$ and $kc = 0.1$ are assigned to the transitions *reproduce_prey*, *death_predator* and *consumption_of_prey*, respectively.



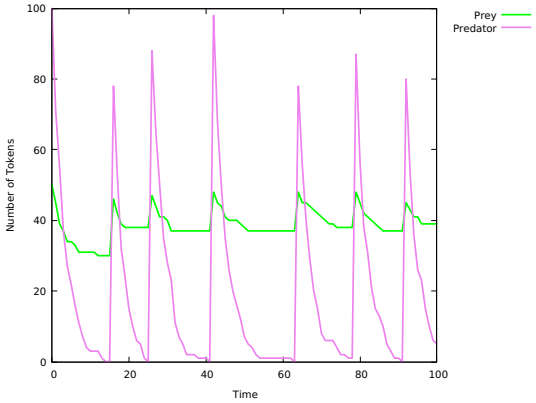
(a) single simulation run with $N = 6$ and $M = 15$.



(b) averaged simulation results over 20 runs with $N = 6$ and $M = 15$.

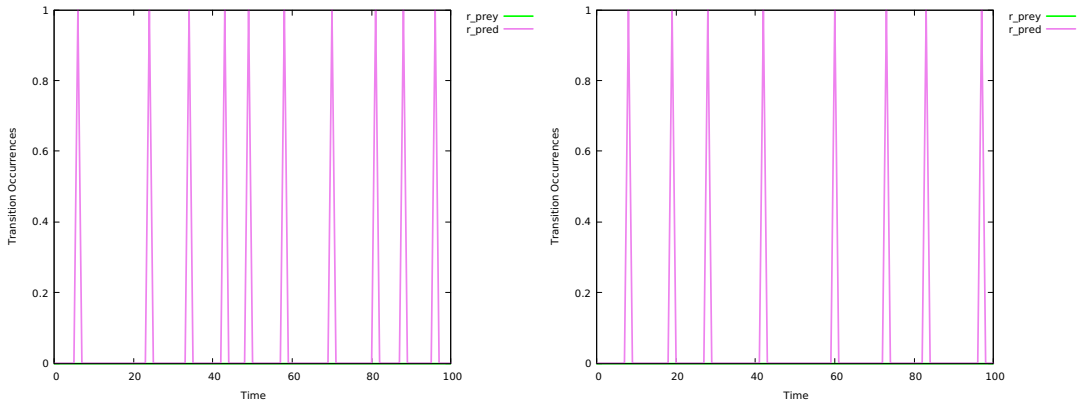


(c) single stochastic simulation run with $N = 17$ and $M = 22$.



(d) single stochastic simulation run with $N = 50$ and $M = 100$.

Figure 2.8: Stochastic simulation results of the extended version of the Lotka Volterra system shown in Figure 2.3. The kinetic parameters $kr = 0.1, kd = 0.4$ and $kc = 0.01$ are assigned to the transitions *reproduce_prey, pred_death* and *consumption_of_prey*, respectively. Both constants $B = 2$ and $Limit = 15$ are used as arc weights. The reset transitions *r_prey* and *r_pred* have the same rate function with the kinetic parameter being 1. Contrary to the traces shown in Sub-figure 2.6a, a dead state is not reached in the extended version due to the reset mechanism.



(a) Single simulation run with N=6, M=15.

(b) Single simulation run with N=17, M=22.

Figure 2.9: Stochastic simulation traces of the reset transitions of the Lotka Volterra system shown in Figure 2.3. The kinetic parameters $kr = 0.1, kd = 0.4$ and $kc = 0.01$ are assigned to the transitions *reproduce_prey, pred_death* and *consumption_of_prey*, respectively. Constant values are $Births = 2$ and $Limit = 15$. Both reset transitions *r_prey* and *r_pred* have the same rate function with the kinetic parameter 1.

2.4.2 Continuous Petri Nets

Continuous Petri nets are another type of quantitative Petri nets. Compared to $\mathcal{SPN}$, each continuous place (represented as shaded line circle) gets assigned a non-negative real number; which can biochemically be interpreted as a concentration of a given species. Moreover, each arc gets assigned a non-negative real number. In $\mathcal{CPN}$, each continuous transition (represented as shaded line rectangle) gets assigned a deterministic firing rate function, which means transitions fire continuously over time. The formal definition of $\mathcal{CPN}$ is given as follows.

Definition 10 (Continuous Petri nets [HH18a])
Continuous Petri nets are a 6-tuple $N = \langle P, T, A, F, V, m_0 \rangle$ where:

- P is a finite, non-empty set of continuous places.
- T is a finite, non-empty set of continuous transitions.
- $A = A_S \cup A_I \cup A_T \cup A_M$ is the set of directed arcs with,
  - A_S, A_I and A_T are sets of the standard, inhibitor and test arcs, respectively. They follow the same definition as in Definition 4.
  - $A_M \subseteq P \times T$ is the set of modifier arcs.
- $P \cap T = \emptyset$.
- $F : A \to \mathbb{R}^+$ is a function which assigns a positive real number to each arc $a \in A$, except modifier arcs.
- $V : T \to H$ is a function which assigns a firing rate function h_t to each transition $t \in T$, whereby $H = \{h_t | h_t : \mathbb{R}_0^{+|{}^\bullet t|} \to \mathbb{R}_0^+, t \in T\}$ is the set of all firing rate functions, and $V(t) = h_t, \forall t \in T$.
- $m_0 : P \to \mathbb{R}_0^+$, is a function which assigns a non-negative real number to each place as initial marking.

Please note that the symbol $\mathbb{R}$ denotes the set of positive real numbers which do not include zero. $\mathcal{CPN}$ do not have equal arcs, as a continuous value can not be precisely tested. Moreover, reset arcs do not exist in $\mathcal{CPN}$ as well. In $\mathcal{CPN}$, for a transition to be enabled, the token value of all its pre-places must be greater than zero.

Continuous Simulation

The underlying semantic of $\mathcal{CPN}$ is best described as a system of ordinary differential equations (ODEs), whereby each place is described by an equation [SH10], expressing the continuous change over time of its token value. Thus, the simulation is performed

by solving the induced system of ODEs. The corresponding ODE describing the change of the place p_i is generated by Equation 2.6, see, e.g., [GH06],

$$\frac{dp_i}{d\tau} = \sum_{t_j \in {}^\bullet p_i} F(t_j, p_i) V_j(\tau) - \sum_{t_j \in p_i^\bullet} F(p_i, t_j) V_j(\tau), \tag{2.6}$$

where p_i is the current marking of the place p_i. With other words, place names are here read as real-valued variables. ${}^\bullet p_i$ and $p_i^\bullet$ denote the pre- and post-transitions of a place p_i, respectively; $V_j(\tau)$ is the rate function of the transition t_j. Summing up all inflow and outflow of a certain place describes exactly Equation 2.6.

Reading our running example in Figure 2.1 as $\mathcal{CPN}$ assuming mass-action kinetics assigned to the transitions *reproduce_prey, pred_death* and *consumption_of_prey* with the kinetic parameters $k1$, $k2$ and $k3$, respectively, gives the following ODEs:

$$\frac{dPrey}{d\tau} = k1 \cdot Prey - k3 \cdot Prey \cdot Predator$$

$$\frac{dPredator}{d\tau} = k3 \cdot Prey \cdot Predator - k2 \cdot Predator$$

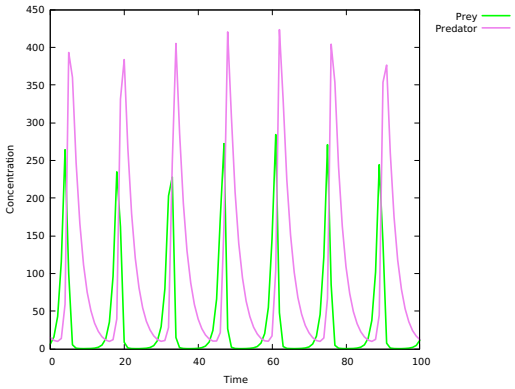
Based on what has been presented above, continuous simulation is performed by means of an ODE integrator to numerically solve the ODEs system of the model on hand; see the basic steps sketched in Algorithm 2.2. The algorithm requires the $\mathcal{CPN}$ model together with its initial state and the interval of simulation time. First, the ODE system corresponding to the input model is constructed (line 1). Afterward, the ODE solver is initialised with the initial state of the system $m(\tau_0)$ (line 3), which then will be added to the output trace (lines 4). Then, the system of ODEs is solved by updating the system state $m(\tau)$ and the current simulation time τ till the simulation end time is reached (lines 6 - 8).

ODE solvers range from simple fixed-step-size solvers (e.g. Euler), which are suitable for unstiff $\mathcal{CPN}$ models, to more sophisticated variable-order, variable-step, multi-step solvers (e.g. Backward Differential Formulas (BDFs)) [HBG+05], which are advisable for stiff $\mathcal{CPN}$ models. Figure 2.10 presents simulation results of the original version of the Lotka Volterra model shown in Figure 2.1 using the stiff solver BDF. As $\mathcal{CPN}$ do not have reset arcs, we can not simulate the extended version of Lotka Volterra in a deterministic way; thus we are going to model it by means of hybrid Petri nets ($\mathcal{HPN}$ for short).

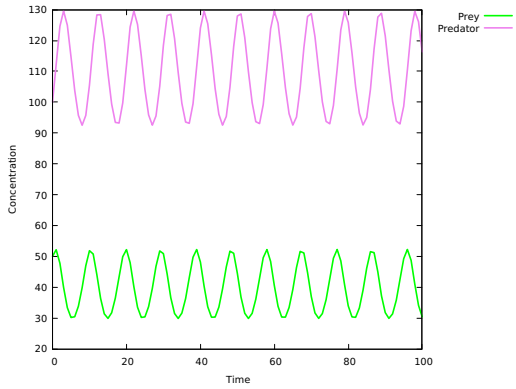
Algorithm 2.2: Basic $\mathcal{CPN}$ simulation algorithm [HH18a].

Input: $\mathcal{CPN}$ with initial state $m(\tau_0)$;
simulation interval $[\tau_0, \tau_{end}]$;
step size $\delta\tau$ with $\delta\tau < (\tau_{end} - \tau_0)$;
Output: Deterministic simulation output trace over time.

1: define function f by constructing the ODEs induced by $\mathcal{CPN}$;
2: time $\tau = \tau_0$; /* intialise current time with start simulation time */
3: state $m(\tau) = m(\tau_0)$; /* assign the initial state to the current state */
4: $m(\tau) \rightarrow store$; /* add the current state to the output trace */
5: **while** $\tau \leq \tau_{end}$ **do**
6: $\tau \leftarrow \tau + \delta\tau$; /* determine next time point */
7: $m(\tau) \leftarrow m(\tau) + \delta\tau.f(s)$; /* compute new state */
8: $m(\tau) \rightarrow store$; /* update the output trace by the current state */
9: **end while**



(a) Constant values are: $N = 6$, $M = 15$ and $Births = 2$.



(b) Constant values are: $N = 50$, $M = 100$ and Births = 2.

Figure 2.10: Continuous simulation traces of the Lotka Volterra system shown in Figure 2.1. The kinetic parameters $kr = 1.1, kd = 0.4$ and $kc = 0.01$ are assigned to the transitions *reproduce_prey*, *death_predator* and *consumption_of_prey*, respectively.

2.4.3 Hybrid Petri Nets

Hybrid Petri nets ($\mathcal{HPN}$) [AD98, HH15] are the third net class of quantitative Petri nets supported in our framework. They increase the modelling power of Petri nets by combining both stochastic and continuous Petri net capabilities in one model. They provide two types of places: discrete and continuous places. Discrete places hold non-negative integer numbers representing number of species. Continuous places hold non-negative real numbers representing the concentration of species [HH14].

Hybrid Petri nets offer all modelling elements (places, transitions and arcs) which are mentioned in the preceding quantitative Petri nets. Thus, modellers have to pay attention when trying to connect elements of different types together. For example, connecting a discrete place to a continuous transition using a standard arc (or vice versa) is not allowed as it contradicts the semantic of continuous transition firing (the standard arc would carry a non-negative real number upon firing the transition). Contrary, connecting a continuous place to a stochastic transition in both directions is allowed as the firing of a stochastic transition will remove an integer value from its pre-place(s) and add an integer value to its post-place(s). Moreover, the special arcs (read, inhibitor and equal arcs) can go from a discrete/continuous place to a stochastic/continuous transition; except connecting a continuous place with a continuous/stochastic transition using an equal arc as a real-valued weight cannot be precisely tested. Furthermore, continuous transitions cannot use reset arcs; see also Figure 2.11.

Th formal definition of $\mathcal{HPN}$ [Her13] is given as follows:

Definition 11 (Hybrid Petri nets ($\mathcal{HPN}$) [Her13])
Hybrid Petri nets are a 6-tuple HPN = $\langle P, T, A, F, V, m_0 \rangle$, where:

- $P = P_{disc} \cup P_{cont}$, whereby P_{disc} is the set of discrete places to which non-negative integer values are assigned, and P_{cont} is the set of continuous places to which non-negative real values are assigned.
- $T = T_s \cup T_i \cup T_d \cup T_{sched} \cup T_{cont}$ with:
  1. T_s is the set of stochastic transitions, which fire stochastically after an exponentially distributed waiting time.
  2. T_i is the set of immediate transitions, which fire with waiting time zero; they have higher priority compared with all other transitions.
  3. T_d is the set of deterministically delayed transitions, which fire after a deterministic time delay.
  4. T_{sched} is the set of scheduled transitions, which fire at predefined time points.
  5. T_{cont} is the set of continuous transitions, which fire continuously over time.

- $P \cap T = \emptyset$.
- $A = A_{cont} \cup A_{disc} \cup A_I \cup A_T \cup A_E \cup A_R \cup A_M$ is the set of directed arcs, with:
  1. $A_{disc} \subseteq (P \times T) \cup (T \times P)$ defines the set of discrete arcs,
  2. $A_{cont} \subseteq (P_{cont} \times T) \cup (T \times P_{cont})$ defines the set of continuous arcs,
  3. $A_T \subseteq (P \times T)$ defines the set of read arcs,
  4. $A_I \subseteq (P \times T)$ defines the set of inhibits arcs,
  5. $A_E \subseteq (P_{disc} \times T)$ defines the set of equal arcs,
  6. $A_R \subseteq (P \times T^D)$ defines the set of reset arcs,
  7. $A_M \subseteq (P \times T)$ defines the set of modifier arcs,

  where $T^D = T_s \cup T_i \cup T_d \cup T_{sched}$ is the set of discrete transitions.
- F is a function

$$F : \begin{cases} A_{cont} \rightarrow, \mathbb{R}^+ \\ A_{disc} \rightarrow \mathbb{N}, \\ A_T \rightarrow \mathbb{R}^+, \\ A_I \rightarrow \mathbb{R}_0^+, \\ A_E \rightarrow \mathbb{N}, \\ A_R \rightarrow \{1\}, \\ A_M \rightarrow \{1\}. \end{cases}$$

  which assigns a positive integer value or a positive rational value as weight to each arc depending on the arc type. If an arc is not explicitly weighted, we assume a weight of 1.
- V is a set of functions $V = \{g, w, d, f\}$ where :
  1. $g : T_s \rightarrow H_s$ is a function which assigns a stochastic hazard function h_{s_t} to each transition $t_j \in T_s$, whereby $H_s = \{h_{s_t} | h_{s_t} : \mathbb{R}_0^{+|^\bullet t|} \rightarrow \mathbb{R}_0^+, t_j \in T_s\}$ is the set of all stochastic hazard functions, and $g(t_j) = h_{s_t}, \forall t_j \in T_s$.
  2. $w : T_i \rightarrow H_w$ is a function which assigns a weight function h_w to each immediate transition $t_j \in T_i$, such that $H_w = \{h_{w_t} | h_{w_t} : \mathbb{R}_0^{|^\bullet t_j|} \rightarrow \mathbb{R}_0^+, t_j \in T_i\}$ is the set of all weight functions, and $w(t_j) = h_{w_t}, \forall t_j \in T_i$.
  3. $d : T_d \cup T_{sched} \rightarrow \mathbb{R}_0^+$, is a function which assigns constant time to each deterministically delayed and three real values to each scheduled transition representing the beginning of the firing interval, the repetition value, and the end of the firing interval; respectively.

4. $f : T_{cont} \rightarrow H_c$ is a function which assigns a rate function h_c to each continuous transition $t_j \in T_{cont}$, such that $H_c = \{h_{c_t} | h_{c_t} : \mathbb{R}_0^{|{}^\bullet t_j|} \rightarrow \mathbb{R}_0^+, t_j \in T_{cont}\}$ is the set of all rates functions and $f(t_j) = h_{c_t}, \forall t_j \in T_{cont}$.

- $m_0 = m_{cont} \cup m_{disc}$ is the initial marking for both the continuous and discrete places, whereby $m_{cont} \in \mathbb{R}_0^{|P_{cont}|}$, $m_{disc} \in \mathbb{N}_0^{|P_{disc}|}$.

It is worth mentioning that Snoopy offers two ways to convert between different types of nodes (only in $\mathcal{HPN}$), while respecting the connectivity rules between two different kinds of elements, compare Figure 2.11. The first way is to select a node, afterwards one should go to *Edit* menu and then choose the command *convert to*; as a result, Snoopy will give the possible target element types to be chosen. The second option is to open the node attribute dialogue (by double clicking the node) and then to choose the target node type from the node type attribute (combo-box user interface (UI)).

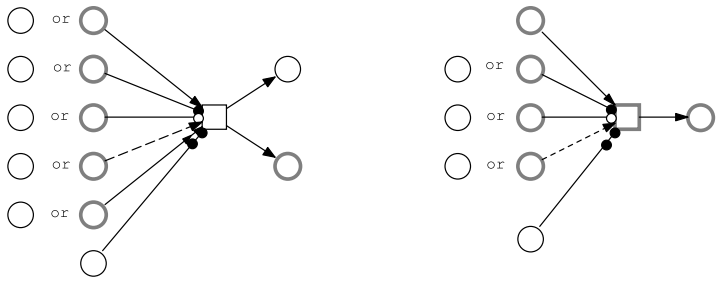


Figure 2.11: Connecting rules among $\mathcal{HPN}$ elements. Graphical representation of nodes in Snoopy: discrete place ○, stochastic transition □, continuous place ○, continuous transition □ [Her13]. Please note that immediate, deterministic and scheduled transitions have the same connection rules as stochastic transitions (left subnet).

Figure 2.12 presents the extended version of the Lotka-Volterra system in a hybrid fashion, in which we add two transitions named *r_prey* and *r_pred*, each resets the system to its initial state when either the place *Prey* or the the place *Predator* gets clean (zero tokens). The transitions *r_prey* and *r_pred* follow the stochastic firing principle, while the transitions *reproduce_prey*, *pred_death* and *consumption_of_prey* follow the deterministic firing principle.

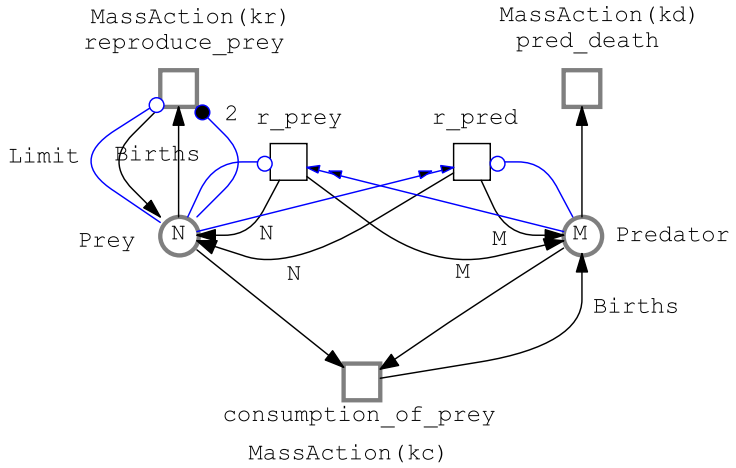


Figure 2.12: Hybrid Petri net model of the extended Lotka Volterra system, for which the reset technique is modelled by the stochastic transitions *r_prey* and *r_pred*; while all other remaining Petri net elements being continuous.

Hybrid Simulation

By simulating a biochemical model by means of the hybrid simulator, the set of reactions is first partitioned into two subsets: the slow and the fast ones. The fast reactions frequently occur and thus it is better to simulate them deterministically, i.e. via an ODE numerical integrator, while the slow reactions occur infrequently and thus it is better to simulate them stochastically, i.e. via a stochastic simulation algorithm. A synchronisation mechanism is utilised to switch between the deterministic and the stochastic sub-systems [IHHA18].

The partitioning of transitions can be done statically before the simulation starts or dynamically while the simulation is in progress; for more details, see [Her13, Pah09]. The synchronisation of the stochastic and continuous sub-systems is crucial to gain accurate simulation results, see [Her13] for more details. The basic hybrid simulation steps are sketched in Algorithm 2.3. The algorithm takes the initial state together with the simulation interval as input. The ODEs corresponding to the continuous part of the $\mathcal{HPN}$ model are constructed (line 1), and the initial state of the system with the initial simulation time are written to the output trace (lines 2-4). Afterwards, the algorithm determines the time at which a discrete event will occur (τ') (line 7), the ODE system is solved (continuous part) and the corresponding trace is written to the output trace till the time point τ' is reached (lines 10-12). Then, the stochastic part begins by determining a stochastic transition to be fired (line 15), then firing the transition and writing the system state to the output trace (lines 16-18).

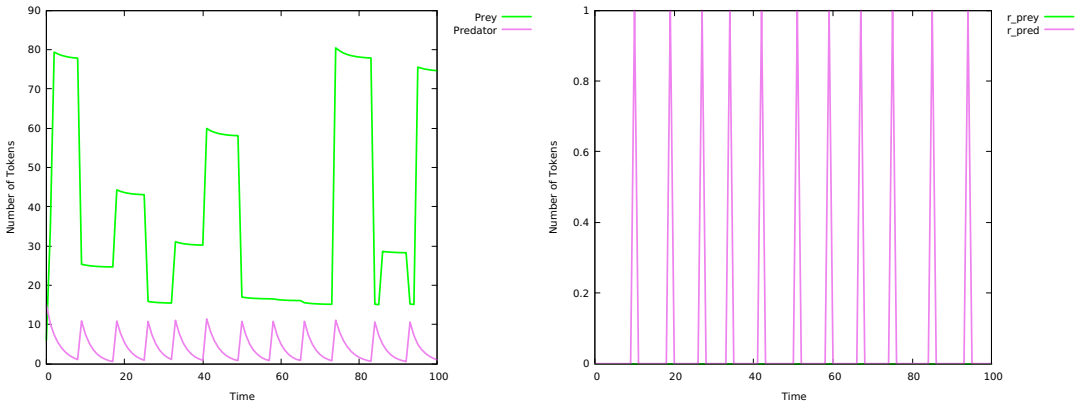
Figure 2.13 presents simulation traces of the extended version in a hybrid fashion.

Algorithm 2.3: Basic $\mathcal{HPN}$ simulation algorithm [Her13].

Input: An $\mathcal{HPN}$ with initial state $m(\tau)$;
simulation interval $[\tau_0, \tau_{end}]$;
step size $\delta\tau$ with $\delta\tau < (\tau_{end} - \tau_0)$;
Output: Hybrid simulation output trace over time.

1: define function f by constructing the ODEs induced by the continuous part of $\mathcal{HPN}$;
2: time $\tau = \tau_0$; /* intialise current time with start simulation time */
3: state $m(\tau) = m(\tau_0)$; /* assign the initial state to the current state */
4: $m(\tau) \rightarrow store$; /* add the current state to the output trace */
5: **while** $\tau \leq \tau_{end}$ **do**
6: **ensure** ODE solver is initialised;
7: determine duration $\delta\tau$ until next stochastic event;
8: $\tau' \leftarrow \tau + \delta\tau$; /* determine next time point */
9: **while** $\tau \leq \tau'$ **do**
10: $\tau \leftarrow \tau + \delta\tau$; /* advance simulation time */
11: $m(\tau) \leftarrow m(\tau) + \delta\tau.f(s)$; /* compute a new state */
12: $m(\tau) \rightarrow store$;
13: **end while**
14: **ensure** $\tau = \tau'$;
15: determine the transition t firing at time τ;
16: $m(\tau) \leftarrow fire(s,t)$; /* actual firing of the stochastic transition and updating the current state */
17: $m(\tau) \rightarrow store$; /* update output trace by the new state */
18: **end while**

Sub-figure 2.13a gives the population change of the preys and predators, whereas Sub-figure 2.13b presents the occurrences of the reset transitions demonstrating that the transition *r_pred* occurs all the time, as the transition *pred_death* has a higher rate than the transition *reproduce_prey*, thus the place *Predator* becomes clean before the place *Prey*. As a result, the transition *r_pred* occurs all the time.



(a) Simulation traces of the variables (places).

(b) Transitions' occurrences.

Figure 2.13: Hybrid simulation traces of the reset transitions of the extended version of the Lotka Volterra system shown in Figure 2.12. Constant values are: *N=6, M=15, Births =2* and *Limit = 15*; simulation traces of the Lotka Volterra hybrid model. The kinetic parameters $kr = 0.1, kd = 0.4$ and $kc = 0.001$ are assigned to the transitions *reproduce_prey, pred_death* and *consumption_of_prey*, respectively. The transitions *r_prey* and *r_pred* got assigned the kinetic rate parameter 1. The time synchronisation method is *Static (exact)* and the used ODE solver is *ARK*.

2.5 Coloured Petri Nets

Coloured Petri nets ($\mathcal{PN}^{\mathcal{C}}$ for short) are a powerful modelling approach, which combine the expressive power of Petri nets with those of programming languages [GL79, Jen81, GL81]. While Petri nets offer a graphical notation for modelling concurrent and communicating/interacting systems, programming languages provide numerous data types which allow us to enrich Petri nets with user-defined functions, colour expressions and other annotations, and thus $\mathcal{PN}^{\mathcal{C}}$ permit to model complex systems in a compact and tidy fashion [Liu12].

Like standard Petri nets, $\mathcal{PN}^{\mathcal{C}}$ consist of places, transitions and arcs. Moreover, $\mathcal{PN}^{\mathcal{C}}$ are enriched by a set of discrete data types (called colour sets) and a set of expressions that are used to define the initial marking, arc inscriptions, and guards. Each place gets assigned a colour set and may contain distinguishable tokens, represented as a multiset expression over the assigned colour set. A multiset is a set, which may contain an element several times. Thus, a place may hold several tokens of the same colour. For instance, let a colour set be $S = \{a, b, c\}$, then the expression 1‘a++2‘b++4‘c is a multiset expression over S which contains 1 occurrence of element a, 2 occurrences of element b and 4 occurrences of element c [AHL21a]. Each transition gets a guard, which is a Boolean expression over variables, constants or functions working with the defined colour sets. The guard of a transition has to be evaluated to true for enabling the transition. Coloured Petri nets are formally defined as follows.

Definition 12 (Coloured Petri net [Liu12])
A *coloured Petri net* is a 8-tuple $N = \langle P, T, A, \Sigma, c, g, f, m_0 \rangle$, where:

- P is a finite, non-empty set of places.
- T is a finite, non-empty set of transitions.
- $P \cap T = \emptyset$
- $A \subseteq (P \times T) \cup (T \times P)$ is a finite set of directed arcs.
- Σ is a finite, non-empty set of colour sets.
- $C : P \to \Sigma$ is a colour function that assigns to each place $p \in P$ a colour set $C(p) \in \Sigma$.
- $g : T \to EXP$ is a guard function that assigns to each transition $t \in T$ a guard expression of the Boolean type.
- $f : A \to EXP$ 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 adjacent to the arc a.

- $m_0 : P \rightarrow EXP$ is an initialization function that assigns to each place $p \in P$ an initialization expression of a multiset type $C(p)_{MS}$.

Please note that extending this definition by the special arcs introduced for $\mathcal{XPN}$ yields coloured extended Petri nets ($\mathcal{XPN}^{\mathcal{C}}$).

Having a coloured Petri net $N = \langle P, T, A, \Sigma, c, g, f, m_0 \rangle$, each coloured place corresponds to a set of (uncoloured) place instances $I_P(p)$, in which each place instance $p(c)$ represents one colour c from the colour set $C(p)$ which is associated with the coloured place $p \in P$. The set of place instances of all coloured places is denoted by I_P. The formal definition of a place instance is given as follows.

Definition 13 (Place instance)
A place instance $p(c)$ is a pair (p,c) , where $p \in P$ and $c \in C(p)$.

For each transition $t \in T$, each expression (transition guard $g(t)$ and its adjacent arcs) needs to be evaluated. For each involved variable $Var(t)$, a binding algorithm has to be applied in order to get all valid bindings $B(t)$ [JKW07]. Each valid binding $b \in B(t)$ represents one (uncoloured) transition instance $t(b)$. The set of all transition instances of a transition t is denoted as $I_T(t)$ and the set of all transition instances of all coloured transitions is detonated as I_T. A transition instance is formally defined as follows.

Definition 14 (Transition instance)
A transition instance $t(b)$ is a pair (t, b), where $t \in T$ and $b \in B(t)$.

Definition 15 (Transition instance enabling)
A transition instance $t(b) \in I_t$ is enabled in a marking m, denoted by $m[t(b)\rangle$, if and only if the following conditions hold:

- $g(t)\langle b \rangle$,
- $\forall p \in {}^\bullet t, m(p) \geq f(p,t)\langle b \rangle$,

which means a transition $t(b)$ is enabled if the guard $g(t)$ induced by a transition's guard and those that may occur on the adjacent arc expressions $f(p,t)$ are evaluated to true. Then, the enabled transition instance $t(b)$ can fire only if pre-place(s) have enough tokens of the given colours.

Definition 16 (Transition instance firing)
A transition instance $t(b) \in I_t$ enabled in a marking m, may fire and reach a new marking m', denoted by $m[t(b)\rangle m'$, with

$m'(p) = m(p) + f(t,p)\langle b \rangle - f(p,t)\langle b \rangle, \forall p \in P$

Let us illustrate these definitions by extending the Lotka Volterra system to a coloured version. As we have seen, the Lotka Volterra system basically comprises two species: prey and predator, in which predators consume preys. We extend the Lotka Volterra system to a food chain system, in which the $species_n$ survives by consuming the $species_{n-1}$. Figure 2.14 gives the coloured Petri net of the food chain system. The set of coloured places are: *Prey* and *Predator*. All places get assigned the colour set *CS* (finite colour set) consisting of three colours $1, 2$ and 3. The place *Prey* is initialised with N token of the colour 1 which is specified either by using the colour expression $N`1$ or by using the colour expression $N`all()$ (which initialises the place *Prey* with N token of all colours) and then assigning the guard $[x = 1]x$ as arc expression which is the case in our example, while the place *Predator* is initialised with M token of each colour which is specified by using the colour expression $M`all()$ which means that all the colours in the colour set *CS* are set to the same coefficient (here it is M).

Places and transitions are connected to each other by using standard arcs which are decorated with arc inscriptions, the arc expression $Births`x$ specifies that Births-many tokens of the colour determined by variable x will be added from the pre-place to the post-place by firing either the transition *consume_sp* or the transition *reproduce_prey*.

The transition *reproduce_prey* is restricted by using the guard (boolean expression) $x = 1$. The colour expression $[x = 1]x$ is required to allow only the colour 1 to be passed over the arc connecting the coloured place *Prey* to the coloured transition *consume_sp*. The colour expression $[x > 1](x) + +[x > 1](-x) + +[x = 1](x)$ is mandatory to preserve the connection in a chain style which means that if the colour value of the variable x is greater than 1, then the colour determined by the variable x together with its predecessor colour will be obtained, otherwise the colour value 1 will be obtained. Please note that the former expression is equivalent to the expression $[x > 1](x + +(-x)) + +[x = 1]x$, but due to some implementation issues in IDD unfolding we deliberately used the former colour expression.

Please note that changing the colour set *CS* adjusts the Petri net model to a different set of predators in the food chain model. Moreover, this also will change the size of the unfolded model without touching the structure of the coloured model. To learn more about the syntax of the colour expressions (EXP). For further details of $\mathcal{PN}^{\mathcal{C}}$ as supported by Snoopy, see [Liu12, LHR12b].

Figure 2.15 gives the equivalent standard Petri net of the food chain produced by unfolding the coloured Petri net model.

2.6 Coloured Quantitative Petri Nets

Likewise, as we have just seen for the uncoloured world, coloured quantitative Petri nets ($\mathcal{QPN}^{\mathcal{C}}$) can be introduced exactly in the same way. For this, each transition

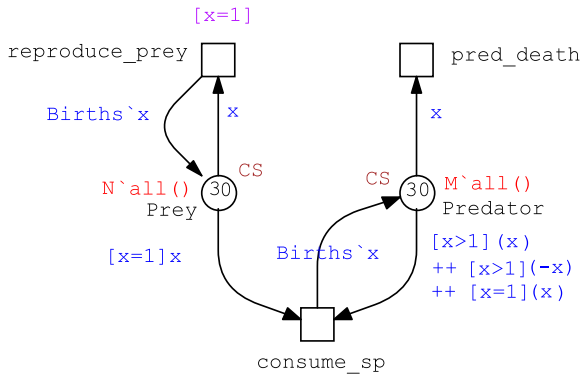


Figure 2.14: Coloured Petri net of the extended version of the Lotka Volterra Petri system (food chain). The colour definitions are: constants: $SIZE = 3; N = M = 10$; colour sets: $int\ CS = \{1 - SIZE\}$; variables: x: CS.

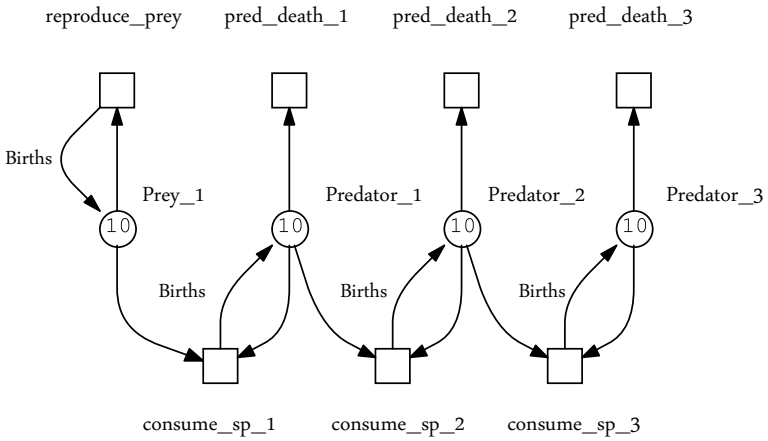


Figure 2.15: Unfolded Petri net of the coloured food chain model given in Figure 2.14.

instance t_c of each coloured transition $t \in T$ is associated with a rate function which can be - as usual - interpreted either in the stochastic or continuous way, giving rise to coloured stochastic Petri nets ($\mathcal{SPN}^{\mathcal{C}}$), coloured continuous Petri nets ($\mathcal{CPN}^{\mathcal{C}}$), and coloured hybrid Petri nets ($\mathcal{HPN}^{\mathcal{C}}$) [Liu12]. The formal definition of coloured quantitative Petri nets is given as follows.

Definition 17 (Coloured Quantitative Petri net [Liu12])
A coloured quantitative Petri net is a 9-tuple $N = \langle P, T, A, \Sigma, c, g, f, V, m_0 \rangle$, where:

- $\langle P, T, A, \Sigma, c, g, f, m_0 \rangle$ is a coloured Petri net.
- $V : I_T \rightarrow H$ is a function which assigns a firing rate function $h_{t(b)}$ to each transition instance $t(b) \in I_T(t)$, $\forall t \in T$, whereby $H = \{h_{t(b)} | h_{t(b)} : \mathbb{R}_0^{+|^\bullet t(b)|} \rightarrow \mathbb{R}_0^+, t \in T\}$ is the set of all firing rate functions, and $V(t(b)) = h_{t(b)}$, $\forall t(b)\ \forall t \in T$.

Please note, firing rates can be colour-dependent, which means the instances of a given coloured transition may enjoy different rate functions. This feature is crucial for modelling and simulating biochemical systems.

Like $\mathcal{SPN}$, an $\mathcal{SPN}^{\mathcal{C}}$ comprises a set of discrete places and stochastic transitions (each associated with a possibly colour-dependent stochastic firing rate). Figure 2.16 shows the coloured stochastic Petri nets of the food chain model, in which the transition *consume_sp* is associated with a colour-dependent rate function which assigns the mass-action kinetics with parameter $kc1$ to the transition instance corresponding to the colour $x = 1$, the mass-action kinetics with parameter $kc2$ to the transition instance corresponding to the colour $x = 2$ and the mass-action kinetics with parameter $kc3$ to the transition instance corresponding to the colour $x = 3$.

$\mathcal{CPN}^{\mathcal{C}}$ are the coloured version of continuous Petri nets, each coloured place gets assigned a continuous number of coloured tokens. Similar to $\mathcal{CPN}$, each coloured transition gets assigned a continuous firing rate function, also here the rate function is possibly colour-dependent. Figure 2.17 shows the coloured continuous Petri net of the food chain model.

It is worth mentioning that the models shown in Figures 2.16 and 2.17 share the same $\mathcal{PN}^{\mathcal{C}}$ structure; the difference is in the interpretation of the rate functions as either stochastic or deterministic rates.

$\mathcal{HPN}^{\mathcal{C}}$ combines the modelling capabilities of both $\mathcal{SPN}^{\mathcal{C}}$ and $\mathcal{CPN}^{\mathcal{C}}$ in one model. Figure 2.18 shows the food chain model as coloured hybrid Petri net. We further extend our coloured model given in Figure 2.17 by two stochastic transitions: *r_prey* and *r_pred*, firing any of these two transitions, will reset the system to its initial state, compare the mechanism discussed for the model given in Figure 2.12. Assuming the place *Prey* reaches one token of the colour 1, this will prevent enabling the transition *r_prey* because of the connected inhibitor arc, which is decorated with the colour expression $1'1$. A reset transition can only fire, when the connected pre-place reaches zero tokens of the colour induced by the variable x. Upon firing the transition, e.g. *r_pred*, the place *Prey* will get N tokens of the colour 1 and the place *Predator* will get M tokens of all colours.

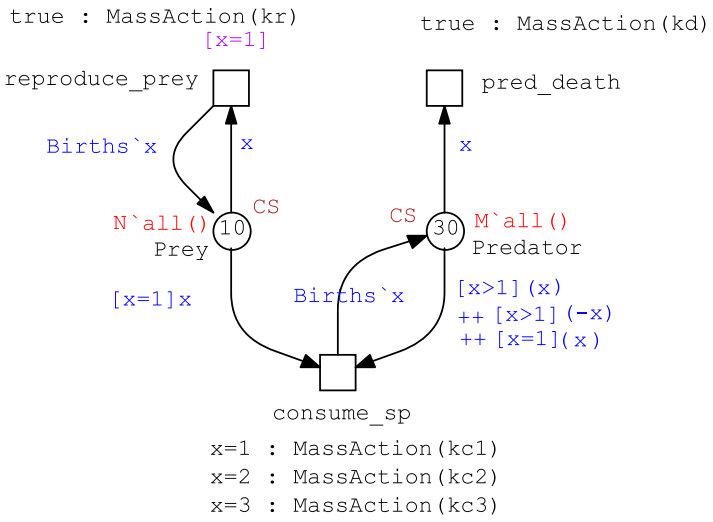


Figure 2.16: An $\mathcal{SPN}^{\mathcal{C}}$ of the food chain. The transition *consume_sp* gets assigned a colour-dependent stochastic rate function with mass-action kinetics. This means the transition instance corresponding to the colour $x = 1$ will enjoy the kinetic parameter $kc1$, the transition instance corresponding to the colour $x = 2$ will acquire the kinetic parameter $kc2$ and the transition instance corresponding to the colour $x = 3$ will get the kinetic parameter $kc3$.

The simulation of $\mathcal{SPN}^{\mathcal{C}}$, $\mathcal{CPN}^{\mathcal{C}}$ and $\mathcal{HPN}^{\mathcal{C}}$ is always done on the uncoloured level. For this purpose, coloured Petri nets are automatically unfolded to their corresponding uncoloured counterparts. Figure 2.19a shows stochastic simulation traces of the $\mathcal{SPN}^{\mathcal{C}}$ food chain model given in Figure 2.16 . Figure 2.19b gives the continuous simulation traces of the $\mathcal{CPN}^{\mathcal{C}}$ food chain model given in Figure 2.17. Figure 2.20 presents the hybrid simulation traces of the $\mathcal{HPN}^{\mathcal{C}}$ food chain model given in Figure 2.18.

2.7 Unfolding Coloured Petri Nets

As we have seen, coloured Petri nets offer a compact way for modelling complex systems, in which the structure of a sub-system can be regularly repeated. Thus coloured Petri nets are an excellent choice for modelling such systems. Having a (scalable) coloured Petri net model, the size of the uncoloured model can be adjusted by changing the colour sets, e.g. changing the colour set *CS* in the food chain model will change the number of predators. Studying the behaviour of coloured Petri nets calls for un-

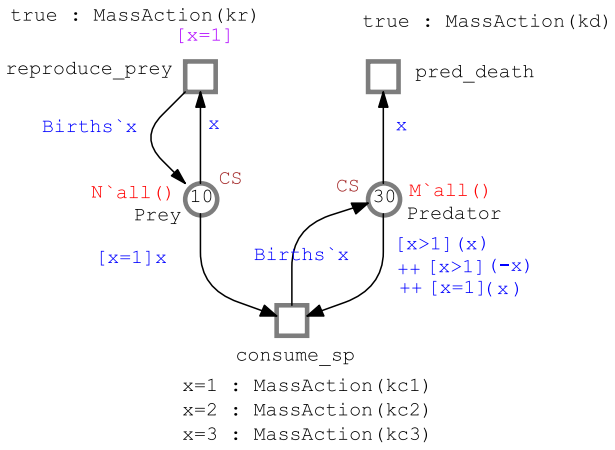


Figure 2.17: A $\mathcal{CPN}^{\mathcal{C}}$ of the food chain. The transition *consume_sp* gets assigned a colour-dependent continuous rate function with mass-action kinetics with parameters $kc1$, $kc2$ and $kc3$. This means the transition instance corresponding to the colour $x = 1$ will enjoy the kinetic parameter $kc1$, the transition instance corresponding to the colour $x = 2$ will enjoy the kinetic parameter $kc2$ and the transition instance corresponding to the colour $x = 3$ will enjoy the kinetic parameter $kc3$.

folding a coloured Petri net into its equivalent standard Petri net (unfolded Petri net); then all analysis and simulation techniques supported by standard stochastic, continuous and hybrid Petri nets can be utilised. The algorithm generating the unfolded Petri net from a coloured Petri net is called Unfolding.

2.7.1 Equivalent Standard Petri Nets

A coloured Petri net has a corresponding equivalent standard Petri net if it builds on finite colour sets [Jen92]. Definition 18 [Liu12] gives the formal definition of an equivalent standard Petri net.

Definition 18 (Unfolded Petri net [Liu12])
Let $N = \langle P, T, F, \sum, C, g, f, m_0 \rangle$ be a coloured Petri net. Its unfolded Petri net $N^* = \langle P^*, T^*, F^*, f^*, m_0^* \rangle$ is defined by:

1. $P^* = I_P$.
2. $T^* = I_T$.

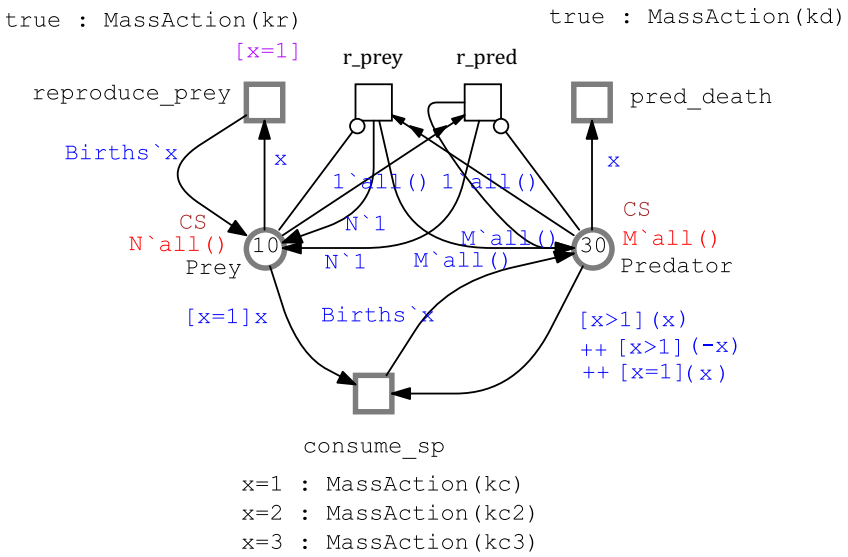
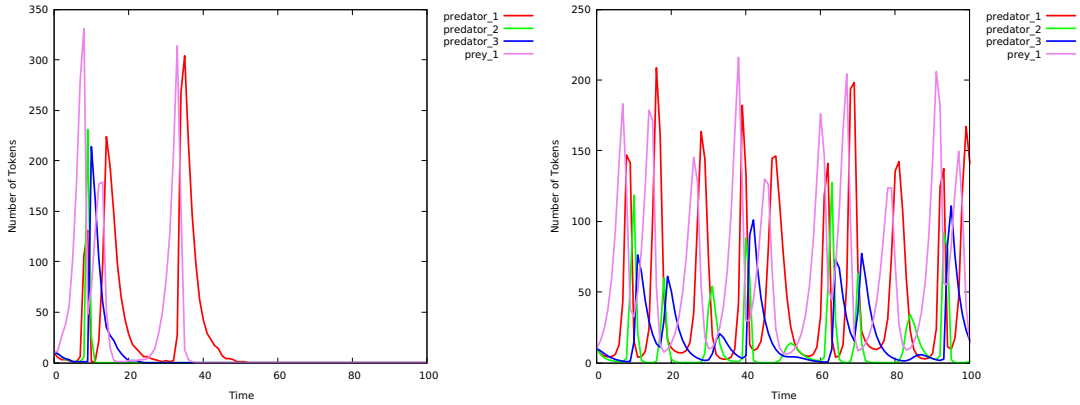


Figure 2.18: An $\mathcal{HPN}^{\mathcal{C}}$ of the food chain. It combines the $\mathcal{HPN}$ in Figure 2.12 with the colour definitions of the model shown in Figure 2.16.

3. $F^* = \{(p(c), t(b)) \in P^* \times T^* | (f(p,t)\langle b \rangle)\langle c \rangle > 0\} \cup$
   $\{(t(b), p(c)) \in T^* \times P^* | (f(t,p)\langle b \rangle)\langle c \rangle > 0\}$.
4. $\forall (p(c), t(b)) \in F^* : f^*(p(c), t(b)) = (f(p,t)\langle b \rangle)\langle c \rangle$,
   $\forall (t(b), p(c)) \in F^* : f^*(t(b), p(c)) = (f(t,p)\langle b \rangle)\langle c \rangle$.
5. $\forall p(c) \in P^* : m_0^*(p(c)) = m_0(p)\langle c \rangle$.

The explanations of this definition are as follows [Liu12].

1. Each place instance (each colour) in the place instance set I_P of the coloured Petri net N corresponds to a place in the Petri net N^*. That is, the coloured tokens in the coloured Petri net are now distinguished by different places in its corresponding Petri net.
2. Each transition instance (each binding) in the transition instance set I_T of the coloured Petri net N corresponds to a transition in the Petri net N^*. This means that each binding of the coloured Petri net is instantiated as a transition in its corresponding Petri net.
3. If the occurrence of t with binding b removes at least one token of colour c from p, denoted by $((f(p,t)\langle b \rangle)\langle c \rangle > 0$, then an arc that connects $p(c)$ and $t(b)$ exists



(a) Stochastic simulation traces of the model given in Figure 2.16 (single run).

(b) Deterministic simulation traces of the model given in Figure 2.17.

Figure 2.19: Stochastic and deterministic simulation traces of the coloured food chain models. The kinetic parameters $kr, kc, kc2, kc3$ and kd are getting assigned $0.5, 0.01, 0.02, 0.03$ and 0.4, respectively.

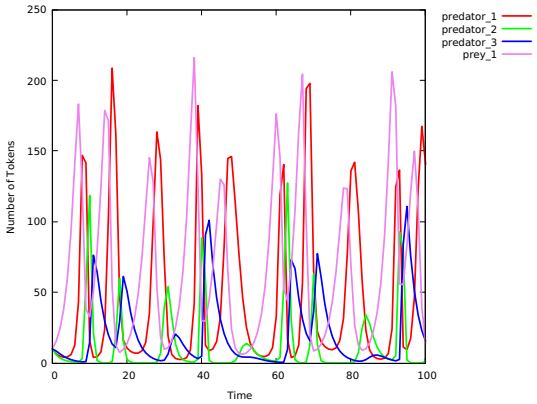


Figure 2.20: Simulation traces of the $\mathcal{HPN}^{\mathcal{C}}$ food chain model shown in Figure 2.18. The kinetic parameters $kr, kc, kc2, kc3$ and kd are getting assigned $0.5, 0.01, 0.02, 0.03$ and 0.4, respectively.

for the Petri net, whose weight is the number of tokens of the colour c, denoted by $((f(p,t)\langle b\rangle)\langle c\rangle$. Analogously, if the occurrence of t with the binding b adds at least one token of the colour c to p, denoted by $((f(t,p)\langle b\rangle)\langle c\rangle > 0$, then an arc that connects $t(b)$ and $p(c)$ exists for the Petri net, whose weight is the number of tokens with colour c, denoted by $((f(t,p)\langle b\rangle)\langle c\rangle$.

4. If the initial marking of the coloured Petri net N contains tokens of the colour c on the place p, then the place $p(c)$ of the Petri net N^* has initial tokens, whose coefficient is the number of tokens of colour c, denoted by $m_0(p)\langle c\rangle$.

Please note that this definition does not involve special arcs and time information, if $\mathcal{PN}^{\mathcal{C}}$ contains special arcs, we only need to set coloured arcs and their corresponding unfolded arcs to the same arc types. For $\mathcal{PN}^{\mathcal{C}}$ with time information, we simply add time information to the transition instances (unfolded transitions).

2.7.2 Unfolding Algorithms

Our platform supports three unfolding algorithms: *Generic*, *Gecode* and *Idd-based* unfolding. All our unfolding engines proceed basically in three steps [SRL+20].

1. Unfolding of coloured places generates for each coloured place as many unfolded places as we have colours in the place's colour set, which is also reflected in the applied naming convention for the generated unfolded places. If the initial marking of a coloured place p contains n tokens of the colour c, then the unfolded place p_c has initially n (black) tokens.

2. Unfolding of coloured transitions generates an unfolded transition (transition instance) for every variable binding and connects this unfolded transition with those unfolded places which correspond to the binding. The naming convention for the generated unfolded transition reflects the variable binding.

3. Deleting isolated unfolded places which are never used yielding isolated places. These isolated places will never influence the net behaviour, even if initially holding tokens; thus they can be safely removed.

These three steps are sketched in Algorithm 2.4. Our unfolding engines differ from each other by the method they use for solving the constraint satisfaction problems *(CSP)* induced by guards which may occur over transitions, arcs or even initial marking. A finite domain CSP can be expressed in the following form. Given are a set of variables, together with a finite set of possible values that can be assigned to each variable, and a list of constraints (boolean expressions). We have now to find all value *combinations* of the variables that satisfy the *CSP* [Tsa93, BPS99].

Remarkably, all CSP's defined by a given $\mathcal{PN}^{\mathcal{C}}$ can be solved independently. In the following, we sketch briefly how each individual engine deals with the CSP problem on hand.

Generic Unfolding The generic unfolding algorithm [LHY12] applies patterns (templates) and basically uses a similar pattern matching mechanism as CPN tools [CK04]. The patterns represent an expression with variables (identifiers) which can be matched

with arguments to give the values of the variables. Let us take our running example (food chain), the arc connecting the place *Predator* to the transition *consume_sp* has the following expression as guard: $[x > 1](x) + +[x > 1](-x) + +[x = 1](x)$ representing three conditions (patterns) separated by the symbol ++ which is interpreted here as boolean OR, and the variable x is defined on the colour set CS. In order to resolve the given guard, the binding of the variable x has to be computed, e.g. all the patterns on the outgoing arc from the place *Predator* involve one variable (x). Thus, we can match the coloured tokens on the place *Predator* (initialised with M tokens of every colour of the colour set CS comprising the colours 1, 2 and 3) with the given pattern yielding three possible bindings of the variable x, which is defined on the colour set CS. The binding list is:

$b1 = \langle x = 1 \rangle$,
$b2 = \langle x = 2 \rangle$,
$b3 = \langle x = 3 \rangle$.

Then, the trivial guard has to be checked against the binding list causing all the bindings to be considered since they satisfy the given guard.

Gecode Unfolding For scaleable coloured models, the generic unfolding algorithm suffers from an annoying increase of the unfolding runtime, particularly for large scaling factors. For the purpose of reducing the unfolding runtime, constraint satisfaction problems induced by, e.g. guarded transitions are solved by means of the constraint solver library *Gecode* [Gec]. The *Gecode* unfolding accelerates the unfolding, but it has a notable increase of the runtime for very large models.

Interval Decision Diagrams-based Unfolding IDDs have been proposed in [LR95]. They belong to the symbolic data structures and can be seen as a generalisation of the popular Binary decision diagrams (BDD) which are helpful to encode Boolean functions. IDDs are Directed acyclic graphs (DAGs) with two types of nodes - terminal and non-terminal ones. There are two terminal nodes represented as boxes, labelled with 0 and 1, and the non-terminal nodes are represented as circles, labelled with variables occurring in the interval logic function to be encoded [SRL$^+$20]. Non-terminal nodes may have an arbitrary number of outgoing arcs labelled with intervals of natural numbers in the form $[a, b)$. To encode the CSP using IDD, the domain of each individual variable is represented as IDD, then the set of all paths going from the root to the terminal node 1 describes all solutions of the given CSP; one path encodes more than one solution.

An IDD is called reduced if the following conditions hold:

1. The interval partitions labelling the outgoing arcs of each non-terminal node are

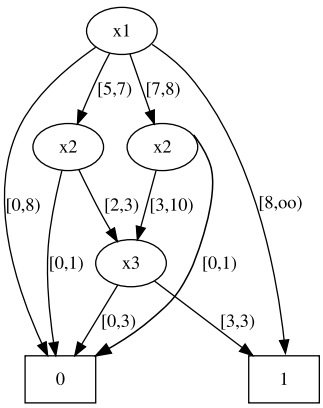
reduced.

2. Each non-terminal node has at least two different children.

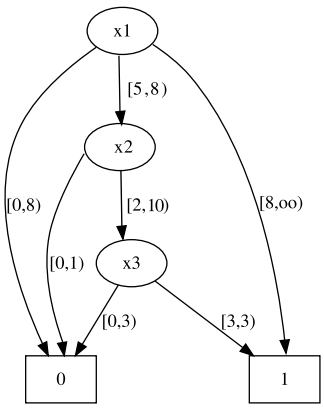
3. There exist no two nodes with isomorphic sub-graphs.

In comparison with the former unfolding approaches, IDD-based unfolding often accelerates the unfolding procedure of scaleable models with large scaling factors. For the entire unfolding algorithm making use of IDD please see [SRL$^+$20].

Figure 2.21 gives an IDD for the boolean expression: $(x1 \geq 8) \vee (x1 \in [5,8) \wedge x2 \in [2,10) \wedge x3 = 3)$. Figure 2.21a presents the non-reduced diagram, while Figure 2.21b depicts the reduced one. Figure 2.22 gives the IDD of the expression $[x > 1](x)++[x > 1](-x)++[x = 1](x)$ which is used in our running example (food chain model) as arc expression.



(a) Non-reduced IDD with three solution paths.



(b) Reduced IDD (two solution paths) by merging the labels of the outgoing arcs of the nodes x_1 and x_2.

Figure 2.21: An interval decision diagram as an example for the purpose of illustrating the IDD principle. First, variables have to be totally ordered, they occur in same order and at most once along each path from the root node to one of the two terminal nodes. All non-valid paths go to the terminal node 0, while the valid ones go to the terminal node 1. The possible solutions are (accorrding to the non-reduced IDD (a)) :$x1 = 5 - 6, x2 = 2, x3 = 3; x1 = 7, x2 = 3 - 9, x3 = 3$ and $x1; \geq 8$.

Algorithm 2.4: Unfolding a coloured Petri net [Liu12].

Input: a coloured Petri net $N = \langle P, T, F, \Sigma, C, g, f, m_0 \rangle$
Output: an unfolded Petri net $N^* = \langle P^*, T^*, F^*, f^*, m_0^* \rangle$

```
1: for each place p ∈ P do
2:    for each colour c ∈ C(p) do
3:       create place instance p(c) and initialise it with the number of tokens of the
         colour c;
4:    end for
5: end for
6: for each transition t ∈ T do
7:    collect involved variables from the guard and adjacent arcs;
8:    compute variable bindings B;
9:    for each binding b ∈ B do
10:      create corresponding transition instance t(b);
11:      for each arc of t do
12:         evaluate its expression EXP;
13:         for each colour c in the binding b of the evaluated expression do
14:            create corresponding arc, its weight determined by the number of
               tokens of the colour c;
15:         end for
16:      end for
17:   end for
18: end for
19: for each unfolded place p(c) ∈ P* do
20:   if it is not connected to any transition instance then
21:      remove p(c); /* isolated place */
22:   end if
23: end for
```

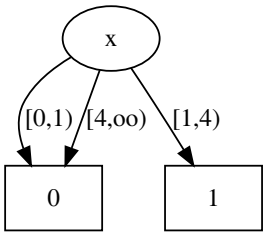


Figure 2.22: IDD diagram of the expression [x>1](x)++[x>1](-x)++[x=1](x) which is used as arc expression in the food chain model given, e.g. in Figure 2.16. The possible solutions are: $x = 1, x = 2$ or $x = 3$.

2.8 Closing Remarks

In this chapter, we recalled various types of Petri nets, qualitative and quantitative ones alike. For each net class, we presented its formal definition together with modelling features, and specifically the semantic of quantitative Petri nets comprising continuous, stochastic and hybrid Petri nets. We also presented their simulation algorithms together with simulation traces of our running example (Lotka Volterra) in different modelling paradigms.

We also have seen how to model and simulate systems with repeated structures using coloured Petri nets, for which unfolding is a crucial step for reusing the analysis techniques of the uncoloured world. We briefly sketched the unfolding algorithms supported in our platform. In this chapter, we used the Lotka Volterra system as a running example to illustrate all presented Petri net classes, We made some extensions to this example to explain some modelling features. The basic version of this model represents the interaction between two kinds of species (modelled as two places). We extended the basic Lotka Volterra system to the food chain version (coloured model) which comprises one prey and many predators (different kinds of predators, each adopts the prey role for the next predator in the modelled chain) so that the repeated sub-systems are well-suitable to be folded using coloured Petri nets.

Starting from the quantitative (coloured) Petri nets we are going to introduce the fuzzy versions of these net classes in the next Chapter for addressing the uncertainties which may be associated with the kinetic parameters.

3 Fuzzy Petri nets

3.1 Introduction

Based on Chapter 2 we go one step further and introduce new classes of Petri nets. These new classes are called fuzzy Petri nets ($\mathcal{FPN}$); they are useful for modelling and simulating systems with uncertain kinetic parameters, e.g. biological systems.

In many biological systems, some kinetic parameters may be uncertain due to incomplete, vague or missing kinetic data (often called fuzzy uncertainty), or naturally vary, e.g., between different individuals, experimental conditions, etc. (often called variability). Fuzzy sets capture kinetic parameters with fuzzy uncertainty or variability by associating each of those parameters with a fuzzy number instead of a crisp real value. By running fuzzy simulation, we obtain an uncertain band for each output according to the fuzzy parameters. Furthermore, we obtain more accurate information about the effect of parameter uncertainty on the output variables by reconstructing membership functions over time.

This chapter is organized as follows. We first review previous work concerning the combination of fuzzy logic with Petri nets in Section 3.2. In Section 3.3 we present some required definitions which are related to fuzzy logic concept including fuzzy sets and fuzzy numbers. Then, we introduce fuzzy quantitative Petri nets and their formal definitions; we illustrate these definitions by means of our running example (Lotka Volterra) in Section 3.4. In Section 3.5, we introduce coloured fuzzy quantitative Petri nets together with their formal definitions; we use the food chain model to illustrate these definitions. In Section 3.6, we present the export relationships among fuzzy and non-fuzzy Petri nets. Then, we introduce fuzzy simulation algorithms and the required sampling strategies in Sections 3.7 and 3.8, respectively. Afterwards, we explain the implementation principle of $\mathcal{FPN}$ in Snoopy in Section 3.9. Last but not least, we sketch some biological case studies which have been modelled and simulated using our framework together with some experimental results in Sections 3.10 and 3.11, respectively.

3.2 Related Works

Fuzzification has been used for dealing with uncertainties in, e.g. biological systems. Uncertainty here means that some parts of the model are not precisely known [GRHS00].

Uncertainties [KDS09] in biological systems generally fall into two categories [HH94]: aleatoric and epistemic. Aleatoric uncertainty usually stems from the inherent randomness in the behaviour of the system under examination. In biology, for example, noise in gene expression induces uncertainty in the model output. Since the noise stems from physical principles, this uncertainty can not be avoided and needs to be addressed by stochastic analysis. Epistemic uncertainty usually results from the lack of knowledge of the biological system to be studied due to such limitations as insufficient understanding of the underlying mechanisms, incomplete measurement data for some components or measurement errors from some data. This kind of uncertainty is appropriately dealt with by many fuzzy approaches [LHG20]. It is worth mentioning that when we deal with insufficient understanding of the mechanisms or when we can not obtain sufficient data of a system, we usually encounter structural uncertainty, i.e. it is hard to determine the exact structure of the model to be built.

The main benefits of fuzzy approaches are based on the generality of function estimators, clarity, modularity and easy handling of uncertainty. Contrary, the main limitations that restrict the use of these systems are the high computational costs and memory requirements [TAAC15].

Fuzzy logic, as a fundamental component of the fuzzy approach, is a combination of various mathematical principles for representing knowledge depending on a gradual degree of membership instead of a crisp membership available in Boolean logic. Fuzzy logic consists of several sets that can be used to map a certain input to an output, a process referred to as fuzzy inference. For instance, inferring gene regulatory networks (GRN) produces hypotheses about the presence or absence of interactions among genes, hypotheses that can later be tested by laboratory experiments [TAAC15].

As Petri nets are a powerful tool for modelling biological systems, the authors of [LHG20] reviewed some popular approaches to combine Petri nets with fuzzy logic and their applications for dealing with uncertainties in systems biology. These approaches are the following three:

Basic fuzzy Petri nets Standard Petri nets are combined with fuzzy logic to represent a set of fuzzy rules [V.R06, LYLT17]. These rules have the same form as the *IF-THEN* and *IF-THEN-ELSE* statements describing the required system response as a function of several linguistic variables. Such kind of modelling yields basic fuzzy Petri nets (BFPN). This class of Petri nets can address the uncertainty of the model structure, when the system under study suffers from insufficient prior knowledge or measurement data to capture its accurate structure. Petri net places represent propositions, e.g. *gene1 is low*, whereas transitions represent fuzzy rules over these propositions, e.g. *if*

gene1 is low and gene2 is low, then gene3 is high. The connecting arcs represent the direction of the reasoning. In a BFPN, there is no conflict, as there is no "resource" concept and the proposition (of a place) may be shared by different rules at the same time. This means that all transitions sharing a pre-place can fire concurrently. Each place p, which represents a proposition, gets a truth degree (or membership degree) in the closed interval [0,1], obtained via fuzzification of a given crisp value of a species like g1. A transition is said to be enabled if its pre-places have tokens and their values are greater than or equal to a threshold. The reasoning process of an FPN is performed by firing transitions (fuzzy rules) and updating the truth degree of the places at each reasoning step [LYLT17]. For this purpose, a reasoning algorithm has to be utilised. The fuzzy inference system (FIS) [MNP11] is responsible for reasoning over the system. The main elements of the FIS are: fuzzifier, inference engine and defuzzifier. Crisp input values are converted via the fuzzifier into fuzzy values, which are then fed into the inference engine. The inference engine performs the reasoning based on, e.g. Mamdani inference method [Mam77] to produce fuzzy output values. Here a fuzzy reasoning algorithm is utilised, see, e.g. [V.R06]. The defuzzifier then converts the fuzzy output values into crisp ones. This approach can equally be applied on coloured Petri nets, which are useful to represent a large-scale system as a compact model by encoding similar components of the system as colours [LC18], see Section 2.5.

Fuzzy quantitative Petri nets (FQPN) combine fuzzy rules with quantitative Petri nets [Win13]. The aim is to complement uncertain modelling capabilities for existing quantitative Petri nets. FQPN offer a semi-quantitative approach for modelling a complex biological system, where some components can be built as uncertain fuzzy models, if some kinetic data are not available or insufficiently precise, and the other as certain ones, if kinetic data are sufficiently known. There are two kinds of FQPNs which have been proposed for systems biology.

1. Fuzzy hybrid functional Petri nets (FHFPN) [Win13], which are considered as an instance of hybrid functional Petri nets (HFPN) [MTA$^+$03] by extending HFPN with fuzzy logic. An HFPN has two kinds of places (transitions): discrete and continuous, and allows the weight of each arc to be a function over its pre-places. Thus, an HFPN model can be divided into two parts, discrete and continuous. The former can model discrete quantities like molecular numbers or concentration levels and how they transit from one state to another, and the latter can model continuous quantities like the concentration of species, and how they evolve continuously. Unlike HFPN, in an FHFPN, the weight of an arc is allowed to be a FIS, besides other forms of functions that are allowed in HFPN.

2. Fuzzy continuous Petri nets (FCPN) [BMZM18, LSHG19], which extend $\mathcal{CPN}$ by fuzzy logic. $\mathcal{CPN}$ offer a graphical way to represent systems of ordinary differ-

ential equations (ODEs). For $\mathcal{CPN}$ modelling and their semantic, please check Section 2.4.2.

In an FCPN, some transitions are allowed to be fuzzy ones and the others are continuous ones that are equivalent to a set of ODE, as a CPN is. Each fuzzy transition is considered as an FIS, which calculates a concentration change per time step of a specific species in a fuzzy way.

Quantitative Petri nets with fuzzy kinetic parameters Standard quantitative Petri nets produce crisp simulation values/traces for specific input which is determined by the initial state of the model and parameter values. However, there are the following two scenarios that need to be modelled [LHG20]:

- For a biological model, if some of its kinetic parameters are unavailable or not precisely known, quantitative (rather than qualitative) analysis of the model may give an uncertain band of output, in which the true trace would lie in, according to the uncertain kinetic parameters.

- For a biological model where some of its parameters vary due to environmental effects or other factors and stochastic methods are not appropriate, it may make more sense to obtain an uncertain band of output, according to the variability of the parameters.

Quantitative Petri nets with fuzzy kinetic parameters are the perfect choice for dealing with such scenarios. They extend the standard quantitative Petri nets by fuzzy kinetic parameters.

Among all these combinations, we are interested in quantitative Petri nets with fuzzy kinetic parameters (third group). Figure 3.1 presents the general approach of combining quantitative Petri nets with fuzzy kinetic parameters represented as triangular fuzzy numbers [ZMC14]. First, the quantitative Petri net model, e.g. a $\mathcal{CPN}$, of the biological network has to be developed, as usual. Then, fuzzy kinetic parameters have to be assigned to some transitions, for which kinetic parameter(s) is uncertain. Then, the fuzzy simulation algorithm has to be performed which gives for each output variable as a result the following:

- a fuzzy band describing the uncertainties associated with the input. A fuzzy band is constructed by making use of all generated simulation traces expressing the uncertainty which is associated with the input fuzzy number (uncertain parameter), see the blue region in Figure 3.1,

- membership function developing over time which gives more accurate information about the associated uncertainties.

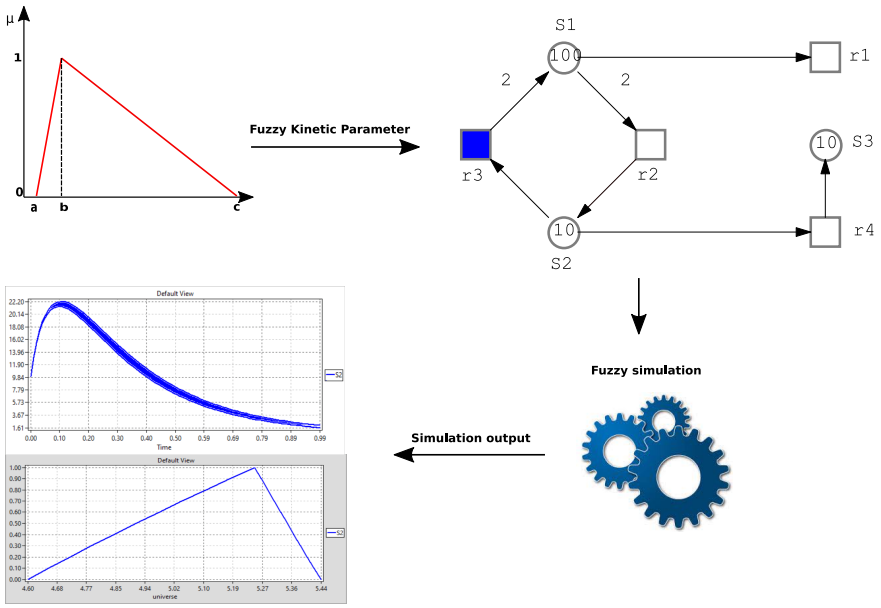


Figure 3.1: A general diagram of fuzzy quantitative Petri nets. Triangular fuzzy numbers are used as kinetic parameters in one or more rate functions. The blue transition (r_3) has a fuzzy kinetic parameter in its rate function, while all other transitions have crisp kinetic parameters in their rate functions. The fuzzy model is fed into the fuzzy simulator which generates fuzzy bands and timed membership functions of the variables (places) of interest as result (here shown for $S2$). Note that one membership function at a certain time point is shown.

In the following, we are going to sketch the related work for quantitative Petri nets with fuzzy kinetic parameters. Fuzzy stochastic Petri nets have been introduced in [LHY16] for exploring the uncertainties of output variables resulting from the uncertainties associated with (input) kinetic parameters. They address both the randomness and fuzziness of biological systems. While stochastic modelling is able to capture the randomness and fine grain behaviour of biological systems which are not appropriately described by deterministic methods, fuzzy sets are able to address uncertainties associated with kinetic parameters.

Fuzzy approaches differ from parameter estimation in the following way. Parameter estimation means to tune parameters and find crisp values to fit the simulation results to in vivo/vitro experiment observations, i.e., removing parameter uncertainties, while fuzzification means to derive the uncertainties of outputs which are caused by uncertain input parameters, i.e., keeping parameter uncertainties. For detailed information about parameter estimation with Petri nets framework, see [SSW06, KTC+06].

In [SRAJ18], the authors used fuzzy stochastic Petri nets for studying the effect of parametric uncertainty on some variables in the Tumor-Immune System. Moreover, triangular fuzzy numbers have been used as fuzzy kinetic parameters of the system's reaction rates.

Fuzzy continuous Petri nets have been introduced in [LCHS18] by combining continuous Petri nets with fuzzy logic. Similar to $\mathcal{FSPN}$, kinetic parameters of continuous rate functions can be represented by means of fuzzy sets if they can not be precisely estimated.

In all these researches, modelling and simulation of $\mathcal{FPN}$ have been done using Matlab. There is no doubt that Matlab is a powerful tool and can be used for this purpose. But it needs a lot of experience and it is not easy to be used especially by non-informaticians, e.g. biologists.

The research presented in this thesis aims at incorporating quantitative Petri nets with fuzzy kinetic parameters into our powerful and graphical modelling and simulation tool Snoopy; so that modellers, e.g. biologists, can more easily develop and simulate their models. Moreover, we consider fuzzy hybrid Petri nets ($\mathcal{FHPN}$) [AHL19] as a combination of both fuzzy stochastic Petri nets ($\mathcal{FSPN}$) and fuzzy continuous Petri nets ($\mathcal{FCPN}$) which was not considered previously. The number of fuzzy kinetic parameters has an influence on the total number of simulations (which have to be performed) and thus on the simulation time and memory load, as we will see in the next sections. This calls for efficient sampling strategies to be considered. Moreover, for the sake of reducing memory load, we reduce the fuzzy bands of the output variables by considering only the minimum and maximum traces over time. Furthermore, we produce for each output variable their membership functions over time, which is not considered in former related work. Finally, we extend coloured quantitative Petri nets by fuzzy kinetic parameters which is a very powerful extension for modelling and simulating systems with repeated structures and kinetic parameters. Here we reuse the same analysis and simulation techniques that are supported by uncoloured fuzzy Petri nets by unfolding a coloured fuzzy Petri net model into its uncoloured fuzzy counterpart, which is done in the background.

The fuzzy approach does not aim to capture sensitivity (i.e., which parameters have a minor/major influence on the output), but is meant to address any lack of knowledge with respect to kinetic parameters, which we would have to assume as given for any sensitivity analysis in the first place. What we obtain by our fuzzy kinetic parameters are - besides the output band - membership functions of any output variable, while sensitivity analysis does not give such kind of measures. See [MHRK08, LCHS18] for a detailed discussion of the fuzzy approach and sensitivity analysis.

The fuzzy approach as supported by this thesis is characterised by the following:

- input: a (coloured) fuzzy quantitative Petri net model, whose transition rates may either enjoy:

1. crisp kinetic parameters, when the values of the kinetic parameters are sufficiently well known, or
2. fuzzy kinetic parameters when the kinetic parameters are not precisely known. We represent each fuzzy kinetic parameter using a triangular fuzzy number, for which uncertainty is mapped into three points, each could be interpreted differently.

- output: is described by two kinds of measures:
  1. A fuzzy (uncertain) band of each variable (place) of interest describing the variability associated with uncertain input parameters. The fuzzy band usually comprises a set of traces describing the variability according to the input parameters (uncertain parameters). In our approach, we represent each band using two curves representing the minimum and maximum traces over time.
  2. Timed membership functions of each variable which precisely describe the associated uncertainties. They capture knowledge about uncertainties caused by input fuzzy kinetic parameters at each time point of simulation, by mapping the interpretation of each point of the input fuzzy kinetic parameter to those forming the output timed membership function at a certain time point.

3.3 Fuzzy Logic

Fuzzy logic was initially proposed to deal with imprecise knowledge by simulating human thinking. It uses a degree of belonging defined by a membership function to describe an element, and thus can represent different kinds of uncertainties. The fundamental concept of fuzzy logic is the fuzzy set [Zad65]. A fuzzy set $\tilde{\xi}$ is defined on a universal set $\mathbb{X}$ by its membership function μ

$$\mu_{\tilde{\xi}} : \mathbb{X} \to [0, 1], \tag{3.1}$$

which means it only takes real values in the closed (unit) interval $[0, 1]$, thus specifying a membership degree for each element belonging to the universal set. In contrast, in traditional (crisp) sets, the membership function only takes the two values $\{0, 1\}$.

A fuzzy number is a special (convex and normalised) fuzzy set with the universal set $\mathbb{X}$ given by the set of real numbers. Commonly used fuzzy numbers include triangular, trapezoidal and Gaussian fuzzy numbers. Among the known types of fuzzy numbers, triangular fuzzy numbers (TFN for short) have been applied in many application fields, such as risk analysis, decision-making, and an evaluation type, where a triangular fuzzy number is used to express the users' comprehensive evaluation of items [ZMC14].

A triangular fuzzy number, denoted by $\tilde{\xi} = (a, b, c)$, $a \leq b \leq c$, is defined as (see Figure 3.2):

$$\mu_{\tilde{\xi}}(x) = \begin{cases} 0 & \text{if } a > x, \\ \frac{x-a}{b-a} & \text{if } a < x \leq b, \\ \frac{c-x}{c-b} & \text{if } b < x \leq c, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

The α-cut of a fuzzy set at a given membership degree $\alpha \in [0, 1]$ (formally called α level), consists of a crisp subset of $\mathbb{X}$, in which each element has a membership degree greater than or equal to the given α level.

$$\tilde{\xi}_\alpha = \{x | \mu_{\tilde{\xi}}(x) \geq \alpha, x \in \mathbb{X}, \alpha \in [0, 1]\}. \tag{3.3}$$

The α-cut of a TFN for any $\alpha \in [0, 1]$ is written as

$$\tilde{\xi}_\alpha = [a + \alpha(b - a), c - \alpha(c - b)]. \tag{3.4}$$

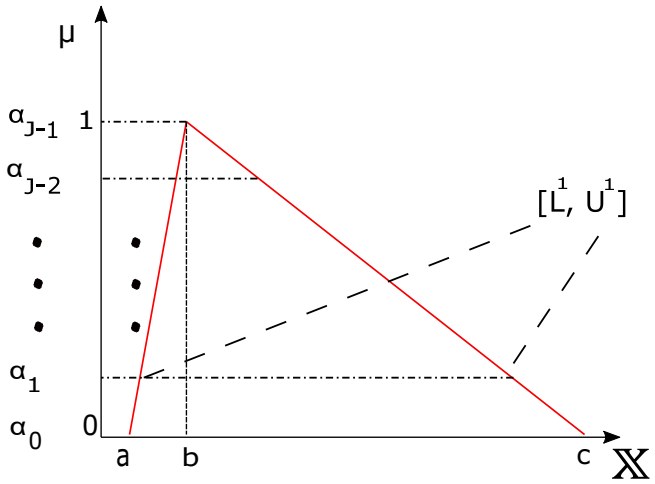


Figure 3.2: A triangular fuzzy number (TFN) defined by the triple (a, b, c), with $0 \leq a \leq b \leq c$, where a can be read as the pessimistic value, b as the most possible value, and c as the optimistic value; and its α-cuts, each defining an α level.

Figure 3.2 gives a triangular fuzzy number and its α-cuts. Note that uncertainty can be turned into fuzzy numbers in the following way: we first obtain a rough estimate for the interval of parameter values, and then extract the pessimistic value (a), the most possible value (b) and the optimistic value (c); all of these three values (a,b,c) define a triangular fuzzy number.

The extension principle [Zad65] offers a general procedure for extending crisp domains to fuzzy domains. Assume $f : \mathbb{X}^n \to \mathbb{Y}$, and $\tilde{A}$ is a fuzzy set on $\mathbb{X}$ such that

$$\tilde{A} = \mu_{\tilde{A}}(x_1)/(x_1) + \mu_{\tilde{A}}(x_2)/(x_2) + ... + \mu_{\tilde{A}}(x_n)/(x_n). \tag{3.5}$$

The notation $\tilde{A} = \mu_{\tilde{A}}(x_i)/(x_i)$ means that x_i has the membership value of $\mu_{\tilde{A}}(x_i)$[Zim10]. Then, applying the extension principle yields the following fuzzy set

$$\tilde{B} = f(\tilde{A}) = \mu_{\tilde{A}}(y_1)/(y_1) + \mu_{\tilde{A}}(y_2)/(y_2) + ... + \mu_{\tilde{A}}(y_n)/(y_n). \tag{3.6}$$

where $x_i \in \mathbb{X}$, $y_i \in \mathbb{Y}$, $y_i = f(x_i)$, $i = 1, 2, ..., n$. By applying the extension principle on a model described by $\mathbf{y} = f(\mathbf{p}, \mathbf{x}, t)$, the model outputs can be expressed as fuzzy sets when some or all elements of input parameters are expressed as fuzzy numbers. Note that $\mathbf{p}$, $\mathbf{x}$ and $\mathbf{y}$ denote the model parameters, inputs and outputs, respectively.

3.4 Fuzzy Quantitative Petri nets

Fuzzy logic has been combined with many methods, such as Petri nets, artificial networks, differential equations to take advantage of the synergy between graphical representation capability and uncertainty handling capability [LHG20].

Fuzzy Petri nets ($\mathcal{FPN}$) [AHL19] are an extension of standard quantitative Petri nets by fuzzy kinetic parameters, meaning that kinetic parameters of rate functions can be defined either as crisp values or fuzzy numbers. Depending on the rate function type, this yields three quantitative Petri net classes: fuzzy stochastic Petri nets ($\mathcal{FSPN}$) [LHY16], fuzzy continuous Petri nets ($\mathcal{FCPN}$) [LCHS18], and fuzzy hybrid Petri nets ($\mathcal{FHPN}$) [AHL19].

$\mathcal{FSPN}$ are an extension of $\mathcal{SPN}$ by associating each transition $t \in T$ with a stochastic rate function, whose kinetic parameter can be seen as a crisp value or a fuzzy number.

In $\mathcal{FCPN}$, continuous Petri nets have been combined with fuzzy kinetic parameters, in which $\mathcal{CPN}$ is described as a set of ODEs, but have some inaccurate or missing kinetic parameters. Because of the existence of fuzzy uncertainties caused by insufficient data as in, e.g. biological systems, we combine fuzzy methods with $\mathcal{CPN}$ to accomplish a trustworthy modelling of such a Petri net class [LCHS18]. This allows continuous rates to enjoy fuzzy kinetic parameters represented as fuzzy numbers.

Combining both $\mathcal{FSPN}$ and $\mathcal{FCPN}$ yields $\mathcal{FHPN}$ complementing the fuzzy Petri net family, in which continuous/stochastic rates are allowed to enjoy fuzzy kinetic parameters and crisp values (as usual).

In the following, the set of all fuzzy numbers is denoted by the symbol Γ. Please note that the set of fuzzy numbers Γ includes real numbers in $\mathbb{R}_0^+$, as a triangular fuzzy number can be seen as a real number by assigning to the points forming the triangular fuzzy number the same value, e.g. (a=b=c). The formal definition of fuzzy quantitative Petri nets is given as follows:

Definition 19 (Fuzzy quantitative Petri nets)
A *fuzzy quantitative Petri net* is a 6-tuple $N = \langle P, T, A, F, V, m_0 \rangle$, where:

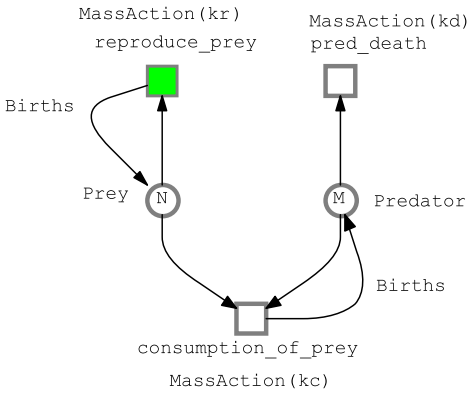
- $\langle P, T, A, F, m_0 \rangle$ is a quantitative Petri net; see Chapter 2
- $V : T \to H$ is a function which assigns a firing rate function h_t to each transition t, $\forall t \in T$, whereby $H = \{h_t | h_t : \Gamma^{|\bullet t|} \to \Gamma, t \in T\}$ is the set of all firing rate functions, and $V(t) = h_t, \forall t \in T$, which means that a kinetic parameter is described by either a fuzzy number or a real (crisp) number in Γ.

We are going to illustrate fuzzy quantitative Petri nets using $\mathcal{FCPN}$. Figure 3.3 gives fuzzy continuous Petri net of the Lotka Voltera systems, for which the kinetic

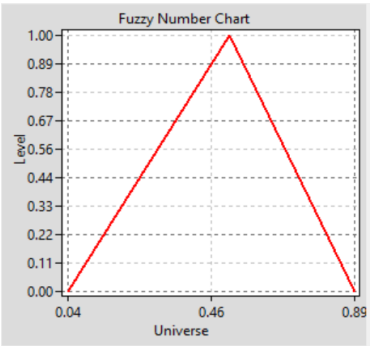
parameter kr of the transition *reproduce_prey* is defined as a triangular fuzzy number, whereas all other kinetic parameters remain crisp, see Table 3.1.

Table 3.1: Lotka Volterra system $\mathcal{FCPN}$- rate functions of the transitions, all of them follow mass/action kinetics pattern.

Transition	Rate function	Kinetic constant k_i
reproduce_prey	$k_r \cdot Prey$	(0.04,0.46,0.89)
consumption_of_prey	$k_c \cdot (Prey + Predator)$	0.1
pred_death	$k_d \cdot Predator$	0.4



(a) $\mathcal{FCPN}$ model



(b) The fuzzy kinetic parameter kr represented as TFN.

Figure 3.3: The $\mathcal{FCPN}$ model of Lotka Volterra system. The transition *reproduce_prey* gets assigned the fuzzy kinetic parameter kr, its membership function is shown in sub-figure (b).

-

3.5 Coloured Fuzzy Quantitative Petri nets

As we have seen, $\mathcal{PN}^{\mathcal{C}}$ combine the expressive power of standard Petri nets with those of programming languages. They are useful for modelling large-scale systems as compact models by encoding components of such systems as colours. Moreover, in coloured Petri nets unfolding mechanism plays a crucial role for obtaining an uncoloured standard Petri net from their coloured counterpart. This permits using all analysis features which are available for standard Petri nets. For more detailed information about colouring and unfolding, we strongly recommend to review Chapter 2.

Coloured fuzzy Petri nets ($\mathcal{FPN}^{\mathcal{C}}$) [AHL21a] are an extension of coloured quantitative Petri nets ($\mathcal{QPN}^{\mathcal{C}}$) by fuzzy kinetic parameters, in which kinetic parameters can be either crisp values (as usual), or fuzzy kinetic parameter - represented as triangular fuzzy numbers. Depending on the rate function type, we obtain three different Petri net classes: coloured fuzzy stochastic Petri nets ($\mathcal{FSPN}^{\mathcal{C}}$), coloured fuzzy continuous Petri nets ($\mathcal{FCPN}^{\mathcal{C}}$) and coloured fuzzy hybrid Petri nets ($\mathcal{FHPN}^{\mathcal{C}}$).

Combining $\mathcal{SPN}^{\mathcal{C}}$ with fuzzy kinetic parameters yields $\mathcal{FSPN}^{\mathcal{C}}$, in which kinetic parameters of rate functions can be either crisp values or fuzzy numbers. $\mathcal{FCPN}^{\mathcal{C}}$ combine coloured continuous Petri nets with fuzzy kinetic parameters. While $\mathcal{CPN}^{\mathcal{C}}$ describe a set of ODEs, fuzzy kinetic parameters describe parametric uncertainties. Similar to coloured hybrid Petri nets, combining both $\mathcal{FCPN}^{\mathcal{C}}$ and $\mathcal{FSPN}^{\mathcal{C}}$ yields $\mathcal{FHPN}^{\mathcal{C}}$. The formal definition of coloured fuzzy quantitative Petri nets is given as follows.

Definition 20 (Coloured fuzzy quantitative Petri net)
A *coloured fuzzy quantitative Petri net* is a 9-tuple $N = < P, T, A, \Sigma, c, g, f, V, m_0 >$, where:

- $< P, T, A, \Sigma, c, g, f, m_0 >$ is a coloured Petri net; see Chapter 2.
- $v : T \to H_c$ is a function which assigns a firing rate function h_{tc} to each transition instance t_c, $\forall t_c \in T_c \; \forall t \in T$, whereby $H_c = \{h_{tc} | h_{tc} : \Gamma^{|\bullet t_c|} \to \Gamma, t_c \in T_c\}$ and $v(t_c) = h_{tc}$, $\forall t_c \in T_c \; \forall t \in T$, which means that a kinetic parameter is described by either a fuzzy number or a real number in Γ.

We chose to illustrate this definition using the food chain modelled as $\mathcal{FSPN}^{\mathcal{C}}$. Table 3.2 gives the rate function of each coloured transition, whereas Figure 3.4 presents the coloured fuzzy stochastic model of the system.

Table 3.2: Food chain $\mathcal{FSPN}^{\mathcal{C}}$- the transitions' firing rate functions.

Transition	Rate function	Kinetic parameter
reproduce_prey	$k_r \cdot Prey$	$k_r = 0.5$
consume_sp	$k_c \cdot Prey \cdot Predator$	$k_c = (0.10, 0.42, 0.43)$
pred_death	$k\_block \cdot Predator$	$k_d = 0.4$

3.6 Export Relationships

Fuzzy Petri nets inherit all features from their non-fuzzy counterparts. This means that it is possible to obtain one $\mathcal{FPN}$ net class from a non-fuzzy Petri net class,

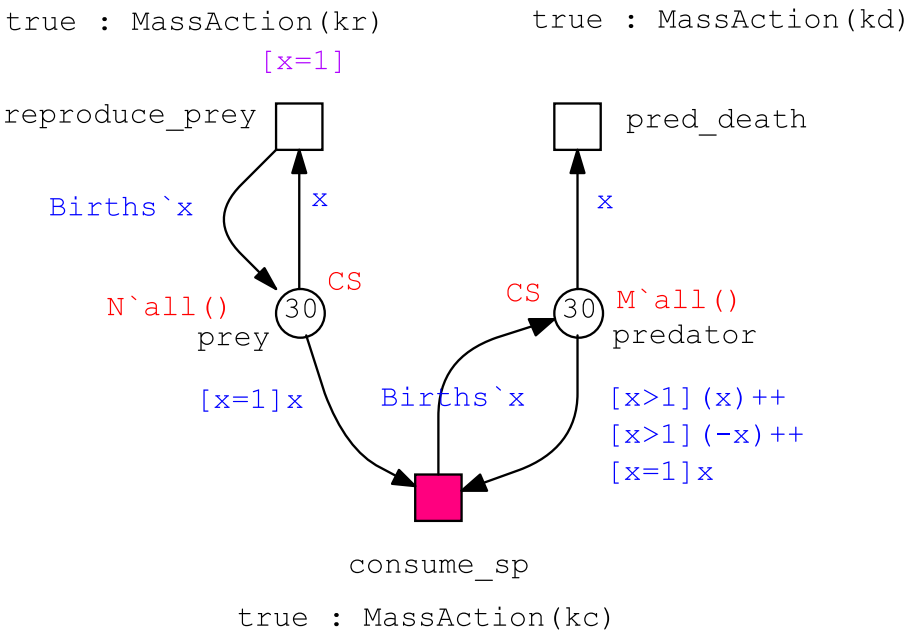


Figure 3.4: The $\mathcal{FSPN}^{\mathcal{C}}$ model of the food chain; The colour-related definitions: *constants:* $SIZE = 3; N = M = 10$*; Births = 2; colorset int CS = {1- SIZE};* and *variable x: CS*. The transition *consum_sp* is marked in red indicating that it has a fuzzy kinetic parameter in its rate function.

where all kinetic parameters are crisp. Moreover, obtaining an $\mathcal{FPN}$ net class from another $\mathcal{FPN}$ net class is simply done by keeping fuzzy kinetic parameters as they are. For the coloured world, generating uncoloured $\mathcal{FPN}$ from the coloured one is done by unfolding which is a required additional step.

Obtaining non-fuzzy Petri nets from their fuzzy counterparts will convert fuzzy kinetic parameters into crisp ones (real values), by keeping the middle value of each fuzzy kinetic parameter (point b). Additionally, $\mathcal{FPN}^{\mathcal{C}}$ can be obtained from $\mathcal{QPN}$ by folding the uncoloured Petri net; where all kinetic parameters are crisp (by default). Figure 3.5 gives the export relations among (coloured) fuzzy and non-fuzzy Petri net classes.

Exporting a Petri net model to the ANDL format (an exchange format between Snoopy and Charlie [HSW15]) is useful for, e.g. performing structural analysis of the given $\mathcal{FPN}$ model using Charlie. Thus, uncoloured fuzzy Petri nets can be directly exported to the *ANDL* format. For the coloured fuzzy Petri nets, they have to be unfolded before exporting. Please note that fuzzy Petri nets are always exported to

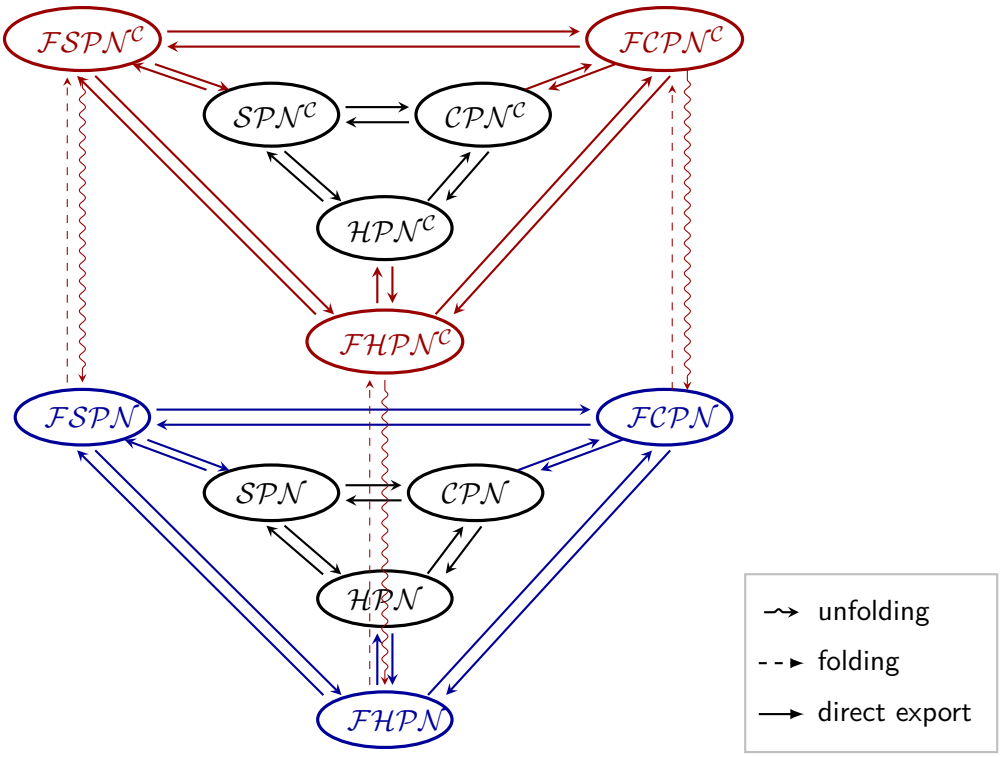


Figure 3.5: Export relation between some of Snoopy's Petri net classes. Fuzzy nets differ from their crisp counterparts by additional pre-defined data types, supporting fuzzy numbers, which can be used as kinetic parameters. The Snoopy extensions presented in [AHL19] are coloured in blue, while the latest addition of net classes supported by Snoopy and their export relation are coloured in red [AHL21a]. Note that for clarity three folding/unfolding relations are not shown in the figure ($\mathcal{SPN}$– $\mathcal{SPN}^{\mathcal{C}}$, $\mathcal{CPN}$– $\mathcal{CPN}^{\mathcal{C}}$, $\mathcal{HPN}$– $\mathcal{HPN}^{\mathcal{C}}$).

their non-fuzzy counterparts (in the *ANDL* files). Coloured fuzzy Petri nets can also be exported to the *CANDL* (Coloured Abstract Net Description Language) format, which is a textual format for the coloured Petri nets supported by Snoopy. Please note that exporting coloured fuzzy Petri nets to the *CANDL* format will convert the given model to its non-fuzzy counterpart by converting the fuzzy kinetic parameters to crisp parameters and converting the Petri net class to the non-fuzzy counterpart. For example, exporting an $\mathcal{FCPN}^{\mathcal{C}}$ model to *CANDL*, will set the target net class to $\mathcal{CPN}^{\mathcal{C}}$, and all fuzzy kinetic parameters will be crisp in the target CANDL file.

3.7 Fuzzy Simulation

Simulation is always done on the uncoloured level. For this purpose, coloured fuzzy Petri nets are automatically unfolded to their corresponding uncoloured counterparts. Please note that fuzzy kinetic parameters do not have any influence on the unfolding step.

Unfolding our running example $\mathcal{FSPN}^{\mathcal{C}}$ shown in Figure 3.4 generates the $\mathcal{FSPN}$ shown in Figure 3.6.

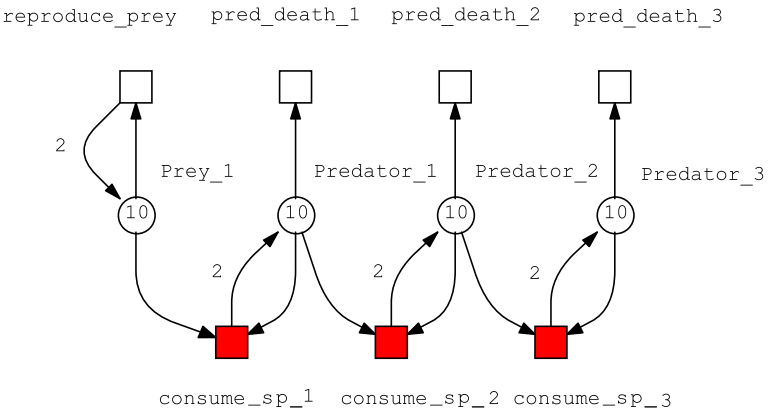


Figure 3.6: The unfolded $\mathcal{FSPN}$ of the food chain model, given as $\mathcal{FSPN}^{\mathcal{C}}$ in Figure 3.4. Please note that the transition instances marked in red have the fuzzy colour-independent kinetic parameter kc in their rate functions.

Simulation The general idea for simulation and analysis of $\mathcal{FPN}$ follows Zadeh's extension principle [Zad65], according to which a fuzzy number is represented as a union of its α-cuts, typically equally spread over the continuous interval $[0, 1]$.

The whole procedure is sketched in pseudo-code notation in Algorithm 3.1. The algorithm starts off with iterating over all α levels, whereby all fuzzy parameters are decomposed into α-cuts. This is mainly done by calling Algorithm 3.2 (line 3) which takes an α level and a fuzzy parameter and returns the corresponding α-cut by applying the Formula 3.4. After that, crisp values are obtained (line 4) by discretising the α-cut. Then we draw samples at each α level and run – depending on the given model class – stochastic, continuous or hybrid simulations for each sample combination (lines 6–8). Please note that a sample combination means to combine samples taken from each involved fuzzy parameter at the same level, for instance, having two fuzzy numbers, then each sample combination comprises two values, each taken from each fuzzy parameter at a certain level. Finally, the output fuzzy band and membership functions over time of each output variable are obtained (lines 11–12).

To obtain the output bands of the time series data, we could simply print all simulation traces together into one plot. This would require to keep all simulation traces. Instead, to reduce the memory load, we determine for each output variable the minimum and maximum values of the traces as they evolve over time. The Algorithm 3.3 is responsible for computing the fuzzy band of a given variable.

Algorithm 3.4 presents how to compose the membership function of a given place at a certain time point t. This algorithm requires the set of result traces of a given variable p and the time point t, for which we would like to compute the membership function. First, the algorithm determines for each α-level (except for the level α=1) the result points (at the time point t) (lines 4-8). Then, we obtain for each level the minimum and maximum result points (lines 9-12). Finally, we draw lines between the computed points starting from the minimum points to the middle point and maximum points. Please see [LHG20] for more details.

Discretising each α-cut of the fuzzy number(s) independently into crisp values may produce redundant samples over all levels. This causes unnecessary simulation runs. To address this issue, more efficient discretising method need to be designed to eliminate redundant samples. Snoopy supports three sampling strategies, but the last one comes with four improvements (algorithms). In the following, K gives the number of fuzzy kinetic parameters, and J the number of α levels.

The $\mathcal{FPN}$ simulation settings comprise the same settings as for standard $\mathcal{QPN}/\mathcal{QPN}^{\mathcal{C}}$, but extended by the following settings:

1. **alpha levels** : specifies the number of α levels; the default value is 10.

2. **discretisation points**: specifies the number of sample points per each level; the default value is 10.

3. **sampling strategy**: the user can choose among seven options: *Basic sampling, Reduced Sampling* and *five LHS sampling algorithms*, outlined in Section 3.8.

3.8 Sampling Strategies

The sampling strategy plays a crucial role for fuzzy simulation. On one hand, it is used to discretise fuzzy numbers into crisp values. On the other hand, it determines the number of simulations which have to be performed. Thus, the more efficient the sampling strategy, the less the number of simulation runs needed. We support three sampling strategies: Basic Sampling, Reduced Sampling and Latin Hypercube Sampling (LHS).

Algorithm 3.1: $\mathcal{FPN}$ simulation algorithm.

Input: $\mathcal{FPN}$ with M variables (places, species) and K fuzzy kinetic parameters, J - number of α levels.

Output: Output bands & membership functions of the M variables over time

1: **for each** α level $\alpha_j, j = 0, 1, \ldots J-1$ **do**
2: **for** `each fuzzy kinetic parameter TFN` **do**
3: α-cut= $ObtainAlphaCut(\alpha_j$,TFN);
4: Sampling: discretise the α-cut and obtain crisp values;
5: **end for**
6: **for each** combination of values for the K fuzzy kinetic parameters **do**
7: Run stochastic/continuous/hybrid simulation;
8: **end for**
9: **end for**
10: **for** `each` variable $Y_m, m = 1, 2, \ldots M$ **do**
11: Use Algorithm 3.3 for obtaining fuzzy band of the corresponding variable Y_m;.
12: Compose all the α-cuts of Y_m to obtain its membership function over time;
13: **end for**

Algorithm 3.2: Compute the alpha cut set of the corresponding fuzzy number and a certain level.

Input: α level α_j and TFN.

Output: α-cut.

1: a = $TFN.GetLeft()$;
2: b = $TFN.GetMiddle()$;
3: c = $TFN.GetRight()$;
4: α-cut= [a+α (b - a), c - α (c - b)];
5: **return** α-cut;

Algorithm 3.3: Compute output fuzzy band.

Input: Simulation traces of an output variable p.
Output: The corresponding fuzzy band.

```
1: MinTrace[ ] = 0; // set the entire trace to 0
2: MaxTrace[ ] = 0; // set the entire trace to 0
3: for each Time Point T_j, j = Start, Start + 1, ... EndTime do
4:   for each Simulation Trace ST do
5:     if MinTrace[T_j] > ST[p][T_j] then
6:       MinTrace[T_j] = ST[p][T_j];
7:     end if
8:     if MaxTrace[T_j] < ST[p][T_j] then
9:       MaxTrace[T_j] = ST[p][T_j];
10:    end if
11:  end for
12: end for
```

3.8.1 Basic Sampling

Discretises each α-cut of the fuzzy number(s) independently into crisp values. This strategy discretises each α level with the same number of samples, except for $\alpha = 1$, and samples are equally spread over each α level. Thus, samples may occur several times at different α levels. The occurrence of redundant samples means that there will be simulation repetition, and thus the same simulation traces will be obtained by the redundant samples, compare Figure 3.7. The sample redundancy problem has an influence not only on the fuzzy simulation time, but also on the memory load due to the repetition of simulation traces. In this case, the total number of simulation runs is given by Equation 3.7. samples.

$$S = N^K \times J + 1, \tag{3.7}$$

with N being the number of samples per level, K is the number of fuzzy kinetic parameters and J is the number of α-levels.

We consider our running Lotka Volterra example ($\mathcal{FCPN}$). For the default settings, i.e., 10 levels and 10 samples per each level. We have $K = 1$, $J = 10$ and $N = 10$, which in turn means: to obtain the averaged result of, e.g., 100 stochastic runs, we have to perform in total 101 x 100 stochastic runs covering all samples over all levels.

3.8.2 Reduced Sampling

This strategy addresses the raised issue of redundant samples by basic sampling. It takes redundant samples into consideration by reusing the samples at $\alpha = 0$ for all

Algorithm 3.4: Compute membership function of an output variable at the time point t.

Input: Simulation traces of the output variable p (denoted as ST_p) and the time point t of simulation time.

Output: Timed membership function of the variable p at the time point t.

```
1: LevelTraces[ ][ ] = 0; /* required output points at all levels */
2: MinPoints[ ] = 0; /* minimum points of all levels */
3: MaxPoints[ ] = 0; /* maximum points of all levels */
4: for each level j ∈ J except j = 1 do
5:    for each simulation trace ST ∈ ST_p at the level j do
6:       LevelTraces[j] .add(ST[t]);
7:    end for
8: end for
9: for each vector v ∈ LevelTraces do
10:   MinPoints.add(min(v)); /* compute the minimum point and add it to the
      vector MinPoints */
11:   MaxPoints.add(max(v)); /* compute the maximum point and add it to the
      vector MaxPoints */
12: end for
13: Add the result point at the level j = 1 to the vector MinPoints;
14: Draw a line segment between each two minimum points starting from the first
    level;
15: Draw line segments between each two maximum points starting from the level
    j = 1;
```

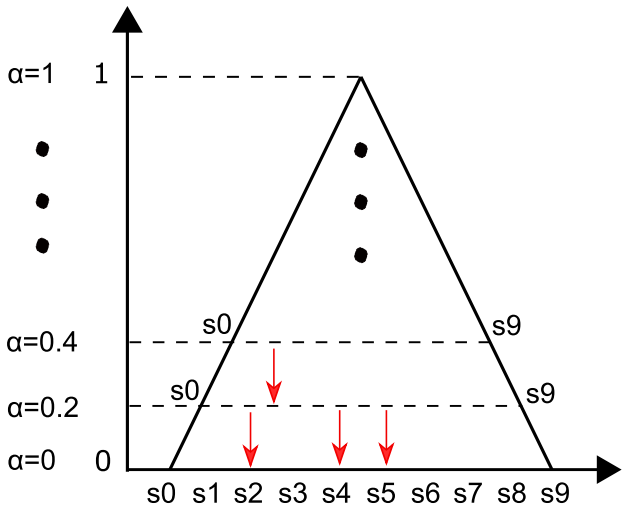


Figure 3.7: Basic sampling strategy. Equidistant samples (here 10) are independently taken on each level. For $\alpha = 1$, there is only one sample. The samples s_2, s_4 and s_5 at the level α=0.2 are examples where the redundancy issue occurs.

levels; compare Figure 3.8. Thus, the number of samples at $\alpha = 0$ should be larger than in the basic sampling strategy, to obtain a suitable resolution of the results. In this case, the total number of simulation runs is given by Equation 3.8.

$$S = N^K + (J - 1) \times 2 + 1, \tag{3.8}$$

with N being the number of samples at $\alpha = 0$, K is the number of fuzzy kinetic parameters and J is the number of α-levels.

Assuming Lotka Volterra with the same setting as we have for basic sampling, $K = 1, J = 10$ and $N = 10$ with averaged result of 100 stochastic runs, we have to perform in total 29 x 100 runs.

3.8.3 Latin Hypercube Sampling

Latin Hypercube Sampling (*LHS*) is a statistical method for generating a near-random sample of parameter values from a multidimensional distribution [MBC79]. Its name originates from the terms *Latin square* and *Latin hypercube* [Wik21a]. A Latin square is a matrix of symbols with the same number of rows and columns, where each symbol occurs exactly once in its rows/columns. It is called *Latin*, because Latin symbols (letters) were used. The second term *hypercube* means that the matrix can be generalized to an arbitrary number of dimensions.

When sampling a function of J variables (uncertain parameters), the range of each

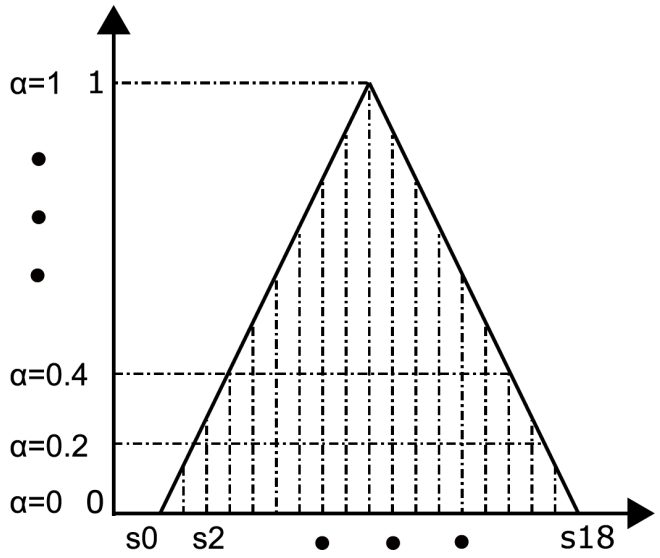


Figure 3.8: Reduced sampling strategy. Reuses the samples at $\alpha = 0$ (here 19) for all levels, if they fall into the corresponding cut, complemented by two samples at each level, determined by the cut with the triangular shape. For $\alpha = 1$, there is only one sample.

variable is divided into K equally probable intervals. K sample points are then placed to satisfy the Latin hypercube requirements [MBC79], i.e., each sample point occurs once in each row/column.

The basic principle of LHS matrix construction follows the basic algorithm (Random *LHS*). Let us assume we have one uncertain kinetic parameter ($K = 1$) and we need to generate four samples ($N = 4$). A random permutation of (1, 2, 3, 4) is generated for each column (fuzzy parameter). The numbers from 1 to 4 denote the row index of each sample, respectively. Let us consider the following permutation (3,1,2,4). This means picking up a uniform random number from each quarter whose index corresponds to the generated permutation. For instance, a random number between 0.5 and 0.75 is chosen for the index of the first sample (picked up from the third quarter as the first number in the generated permutation is 3), see Figure 3.9. In this way, we obtain a sample distribution satisfying the LHS requirement. Finally, the uncertain range of the fuzzy kinetic parameter is mapped to each column in the constructed matrix.

For our purpose, one sampling matrix is constructed for each α level, except the level $\alpha = 1$. Figure 3.10 illustrates the LHS sampling matrix with N samples and K fuzzy kinetic parameters for a certain level $j \in J$. Thus, the total number of simulation runs is given by Equation 3.9.

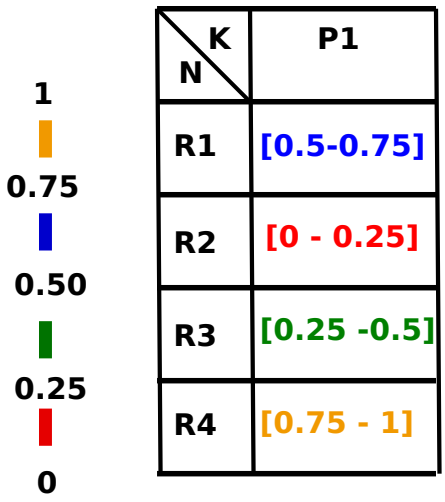


Figure 3.9: *LHS* construction for one uncertain parameter (P1) and four samples (R1 to R4). The range from 0 to 1 is equally divided into stripe quarters (depending on the number of samples). The sample locations are determined using a random permutation of (1, 2, 3, 4). The uniqueness here is characterised by two features random numbers and colours. Note that for one uncertain kinetic parameter we obtained one column (vector), but the general principle follows a square matrix (same number of rows and columns). Please also note the generated values have to be replaced with the uncertainty range induced by the given uncertain kinetic parameter $P1$.

$$S = N \times (J - 1) + 1, \tag{3.9}$$

with N being the number of samples (per each level), which will then be translated into the number of rows of the sampling parameter matrix, while its number of columns is determined by K (number of fuzzy kinetic parameters). This sampling scheme does not require more samples for more dimensions (variables). Using LHS we get now the minimal number of simulation runs; no matter how many fuzzy kinetic parameters do exist in the model.

Snoopy uses the public library **lhslib** [LIBa] for generating the sampling parameter matrix. Besides the standard LHS Random algorithm, the **lhslib** library supports four improved algorithms, which are included in Snoopy's $\mathcal{FPN}$simulator. In the following we briefly sketch these algorithms:

- **Improved Latin Hypercube Sampling** creates an LHS matrix from a set of uniform distributions to be used in creating an LHS matrix design. It attempts to optimize the samples with respect to an optimum euclidean distance between

design points [BR12]. The optimum distance D is obtained using Equation 3.10:

$$D = N/N^{\frac{1.0}{K}}, \tag{3.10}$$

Where n is the number of rows (of the LHS matrix) or sample points and K is the number of parameters.

- **Optimum Latin Hypercube Sampling** draws an LHS matrix from a set of uniform distributions to be used in creating a Latin Hypercube Design. This method uses the Columnwise Pairwise (CP) algorithm to generate an optimal design with respect to the S optimality criterion. The S-optimality seeks to maximize the mean distance from each design point to all the other points in the design, so the points are spread out as much as possible. For detailed information about the CP algorithm, please see [Sto05].
- **Latin Hypercube Sampling with a Genetic Algorithm** this algorithm attempts to optimize the samples with respect to the S optimality criterion through a genetic algorithm. For more detailed information about the algorithm, please see [Sto05].
- **Maximin Latin Hypercube Sampling** draws an LHS matrix from a set of uniform distributions to be used for creating a Latin Hypercube Design. This algorithm attempts to optimize the samples by maximizing the minimum distance between design points (maximin criteria) [Ste87].

Sample No	p_1	p_2	.	.	p_K
1	$s_{11} \in [a_1, b_1]$	$s_{12} \in [a_2, b_2]$	·	·	$s_{1K} \in [a_K, b_K]$
2	$s_{21} \in [a_1, b_1]$	$s_{22} \in [a_2, b_2]$			$s_{2K} \in [a_K, b_K]$
·			·	·	·
·					·
N	$s_{N1} \in [a_1, b_1]$	$s_{N2} \in [a_2, b_2]$	·	·	$s_{NK} \in [a_K, b_K]$

Figure 3.10: LHS sampling matrix with N samples and K fuzzy kinetic parameters for a given level $j \in J$. Each row represents one sample combination of all fuzzy kinetic parameters. Note that $[a_i, b_i]$ is the uncertain range of the parameter i at the level j; e.g. the range $[a_1, b_1]$ corresponds to $[s_0, s_9]$ in Figure 3.7.

Assuming two fuzzy kinetic parameters and one of the LHS strategies is chosen, with the number of samples per level $N = 10$ and number of levels $J = 10$, then the required number of stochastic simulation runs will be $10 \times 10 + 1$, averaging results over, e.g. 100 stochastic runs will obtain in total 101×100. Compared to the basic sampling

strategy, LHS obviously dramatically decreases the number of simulation runs required in total. However, there is some additional overhead due to the computation of the LHS matrix and the Random Number Generator (RNG) of the LHS library.

In the following, we present fuzzy simulation traces for our running example in two versions (plain and coloured).

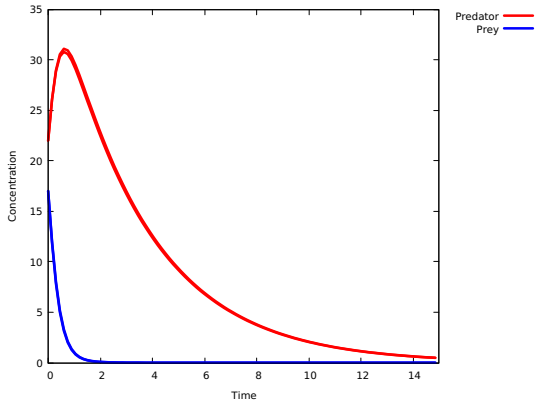
- Figure 3.11 gives the fuzzy simulation results (fuzzy bands and time membership functions) of the Lotka Volterra system (basic version as $\mathcal{FCPN}$) shown in Figure 3.3. Each fuzzy band comprises two curves: the minimum and the maximum traces over time. The two curves together describe the uncertainty caused by the variability of the input fuzzy kinetic parameter k_r. Sub-figure 3.11a gives a narrow fuzzy band of both variables *Prey* and *Predator* (even identical) as the used kinetic parameter k_r describes a narrow uncertain range (0.06, 0.09, 0.25), thus there is no big influence on the output, which means we obtain less uncertainty for the used kinetic parameter. Sub-figure 3.11b gives wider fuzzy bands describing larger uncertainties for the kinetic parameter k_r = (0.04, 0.52, 0.89). As we notice, the corresponding timed membership functions have a line shape, which means that the points forming membership functions a, b and c are almost identical. This is also reflected by the corresponding fuzzy band. For instance, if we consider Sub-figure 3.11d, the membership function of the variable *Prey* reports that $a = b = c = 0$ at time point 85, this can be seen from the fuzzy band of the variable *Prey* at the same time point. The same discussion is also applied for the membership function of the variable *Predator* which reports that $a = b = c = 2.94$.
- Figure 3.12 presents the fuzzy simulation traces of the coloured food chain model. Both traces show that increasing number of stochastic runs has an influence on the shape of fuzzy bands and membership functions of the variables.

3.9 Implementation Principle

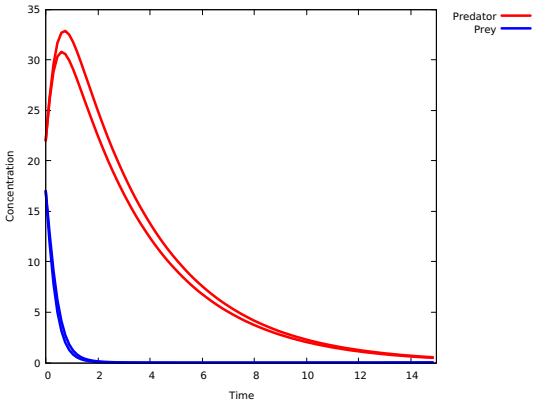
The implementation of $\mathcal{FPN}$ comprises two aspects: Modelling and Simulation. The modelling aspect is achieved by extending the already existing quantitative (coloured) Petri net classes by fuzzy kinetic parameters. In the following, we sketch class diagrams for some modelling and simulation aspects. For the purpose of simplifying the diagrams, we only give the class names without presenting the data members and the methods of each class.

Figure 3.13 presents the class hierarchy of fuzzy Petri nets as they are implemented in Snoopy. Note that all metadata for the modelling and simulation aspects, e.g. the place/transition information are kept in sophisticated data structures.

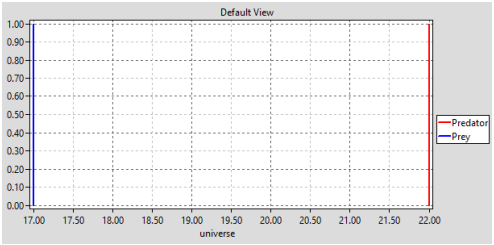
The simulation aspect comprises the following implementations:



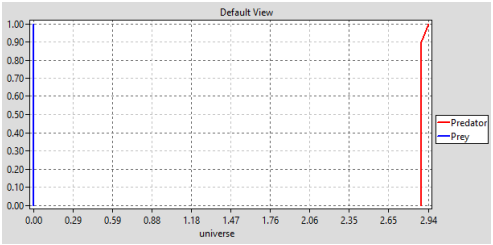
(a) The fuzzy bands - TFN with (0.06, 0.09, 0.25).



(b) The fuzzy bands - TFN with (0.04, 0.52, 0.89).



(c) Membership functions at time point 3.

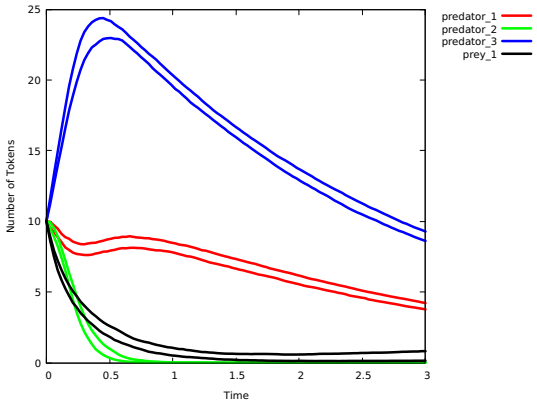


(d) Membership functions at time point 85.

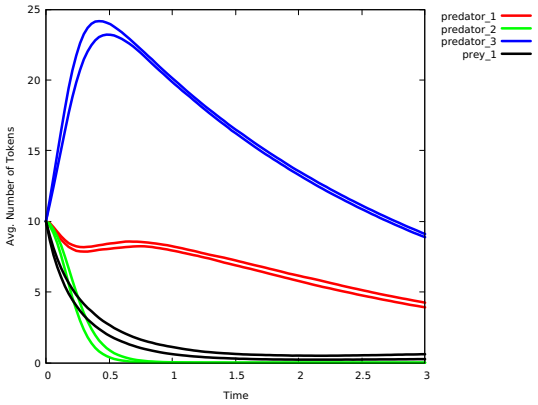
Figure 3.11: Fuzzy continuous simulation results of the Lotka Volterra system ($\mathcal{FCPN}$) shown in Figure 3.3. Constant values are: $N = 17$ and $M = 22$ and $Births = 2$. The chosen sampling strategy is *Basic Sampling*, the number of levels $J = 10$ and number of samples per each level is 10.

- Sampling algorithms (strategies), see Section 3.8.
- Fuzzy simulation, see Section 3.7.
- Computing the fuzzy bands/timed-membership functions of the output variables.
- Suitable data structures for keeping fuzzy bands together with timed-membership functions.
- Extending Snoopy's result viewer by drawing fuzzy bands and the corresponding timed-membership functions of the selected variables (places), see Figure 3.15.

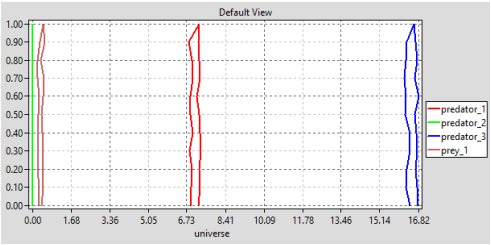
Figure 3.14 presents the class diagram of the $\mathcal{FPN}$ simulation engines. Note that the library *SPSIM* [sps] is used for performing the stochastic/continuous/hybrid sim-



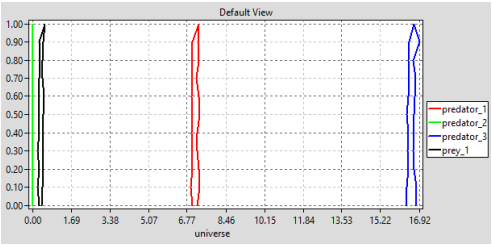
(a) averaged simulation results over 500 runs.



(b) averaged simulation results over 1000 runs.



(c) timed membership functions at t=49 (averaged 500 runs).



(d) timed membership functions at t=49 (averaged 1000 runs).

Figure 3.12: Fuzzy stochastic simulation results of the food chain system shown in Figure 3.4 and its unfolded $\mathcal{FSPN}$ given in Figure 3.6. Constant values are: $N = 10$ and $M = 10$ and $Births = 2$. The chosen sampling strategy is *Basic Sampling, the number of levels* $J = 10$ *and number of samples per each level is 10. The fuzzy parameter* k_c *gets the uncertainty range (0.10, 0.42, 0.43)*

ulation over the crisp kinetic parameters. The *SPSIM* library is part of our *PetriNuts* tool family.

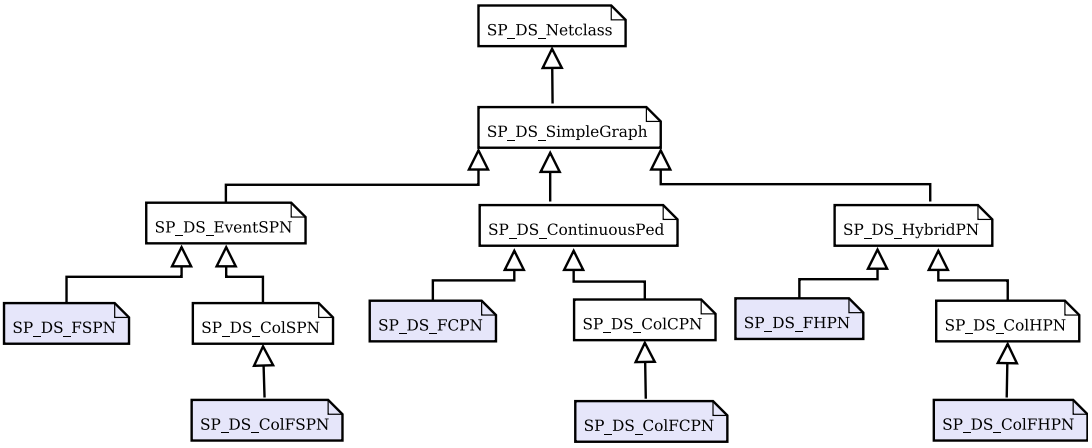


Figure 3.13: $\mathcal{FPN}$ class diagram. The highlighted classes represent the $\mathcal{FPN}$ extension in Snoopy, which are built by extending the unhighlighted ones.

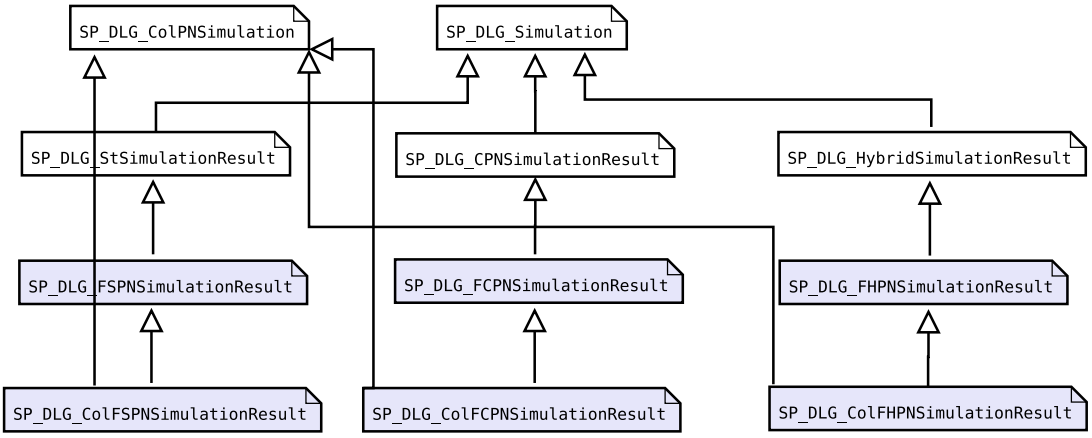


Figure 3.14: The class diagram of the $\mathcal{FPN}$ simulation engines. The highlighted classes represent the class extensions, which are built by extending the unhighlighted ones. Note that the parent class *SP_DLG_ColPNSimulation* keeps the unfolding information for coloured fuzzy Petri nets.

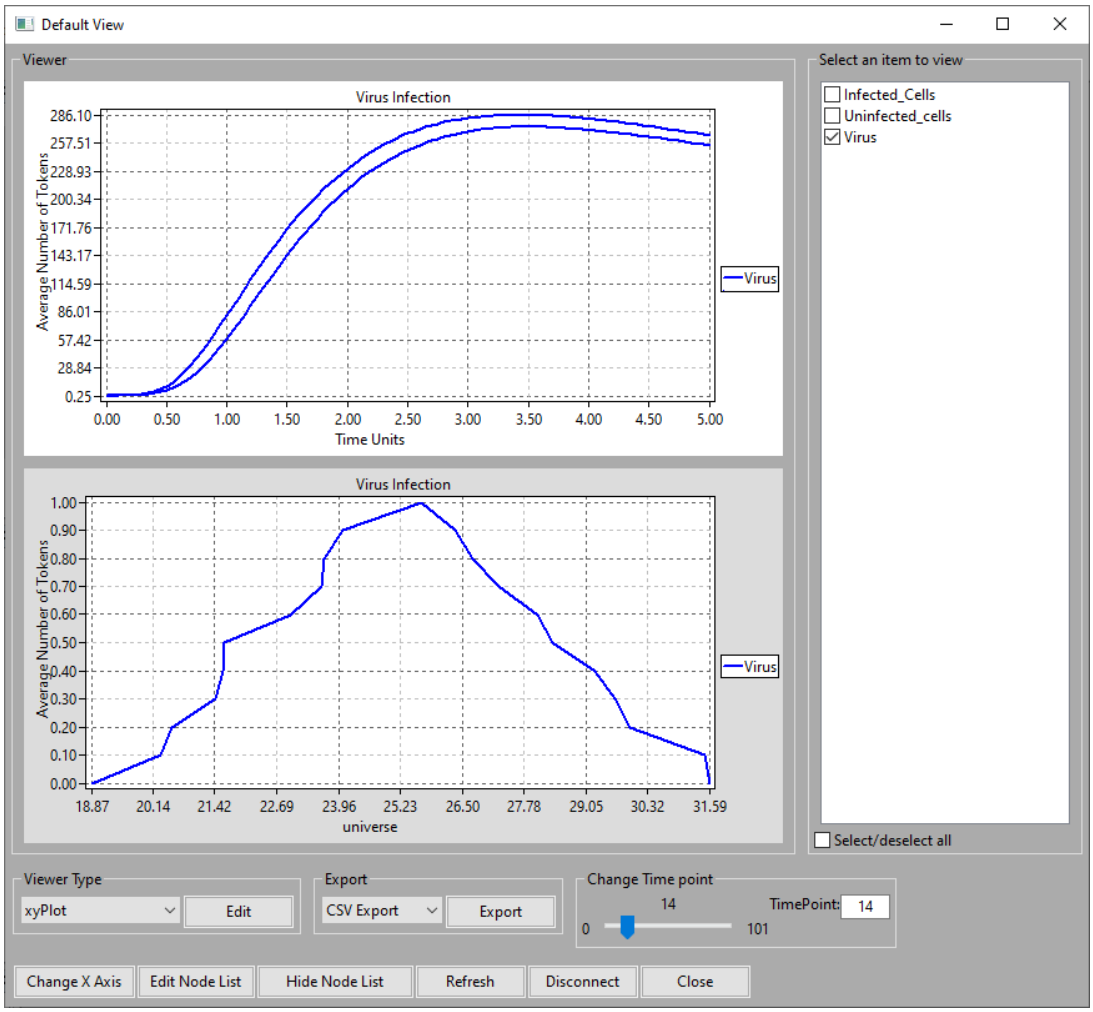


Figure 3.15: Snoopy's result viewer for all (coloured) Petri net classes belonging to the family of fuzzy quantitative Petri nets. The upper sub-window shows the fuzzy bands of the selected variables (places) which are constructed from the minimum and maximum traces over simulation time. The lower sub-window views timed membership functions of the selected variables. Please note that one can adjust the time point either by scrolling the scroll tool or feeding it directly in the related text field. Both fuzzy bands and timed membership functions of the selected places can be exported into an image format. Fuzzy bands of the selected places can also be exported into the *CSV* format.

3.10 Case Studies

In the following, we discuss six case studies ($\mathcal{FPN}$ and $\mathcal{FPN}^{\mathcal{C}}$), for which some of the kinetic parameters are assumed to be uncertain. For each case study, we present its fuzzy Petri net model together with a table of the rate functions. Then we give fuzzy simulation traces represented as fuzzy bands and timed-membership functions. Fuzzy bands give us an impression of the associated uncertainties, whereas timed-membership functions provide more accurate information about the associated uncertainties. Interested readers can also find more case studies in our technical report [AHL21b].

We would like to emphasise that the positions of the fuzzy kinetic parameters have been chosen in an arbitrary way, i.e. we chose arbitrary transitions to assign fuzzy kinetic parameters to the rate functions. The chosen values of the points forming fuzzy kinetic parameters (a, b and c) have been chosen in an arbitrary way as well.

3.10.1 Virus Infection

The virus infection model [LHY16] describes the infection of healthy cells by a virus. Cells grow or die. The virus may enter a healthy cell (UCell) and infect it (ICell). Then the virus starts the replication of itself and more viruses are released. Besides, infected cells may die and viruses may degrade. Figure 3.16 gives the $\mathcal{FSPN}$ model. The system starts with one virus and 100 healthy cells (uninfected cells). Please note that the virus replicates itself with 10 viruses as specified by the weight of the arc connecting the transition *virus_release* and the place *Virus*. Table 3.3 gives rate functions of the model transitions.

Table 3.3: Virus infection $\mathcal{FSPN}$- rate functions of transitions, all following mass/action kinetics.

Transition r	Rate function	Kinetic constant k
$ucell\_death$	$k\_ucelldeath \cdot Uninfected\_cells$	$k\_ucelldeath = 0.1$
$cell\_growth$	k_growth	$k\_growth = 1$
$infection$	$k\_infect \cdot Uninfected\_cells \cdot Virus$	k_infect =(0.2, 0.3, 0.5)
$icell\_death$	$k\_death \cdot Infected\_Cells$	$k\_death = 0.5$
$virus\_release$	$k\_release \cdot Infected\_Cells$	$k\_release$ =1
$degradation$	$k\_deg \cdot Virus$	k_deg =0.1

Figure 3.17 gives fuzzy bands of the variables (places) together with timed membership functions using different numbers of stochastic runs. We notice that the number of stochastic runs has an important rule in determining the shapes of fuzzy bands and timed membership functions. A high number of stochastic runs gives better shaped bands/membership function that we can trust due to the inherent random-

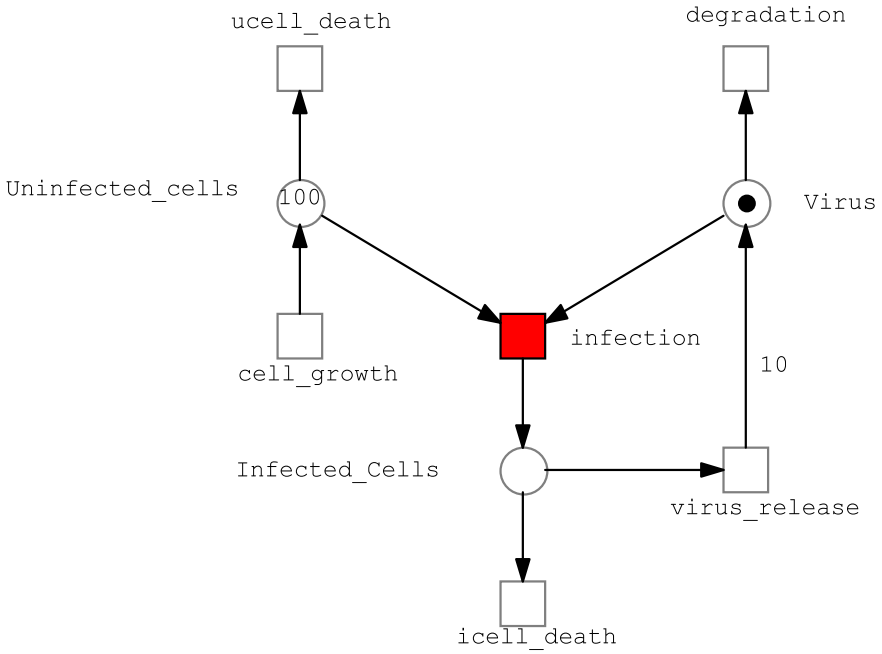
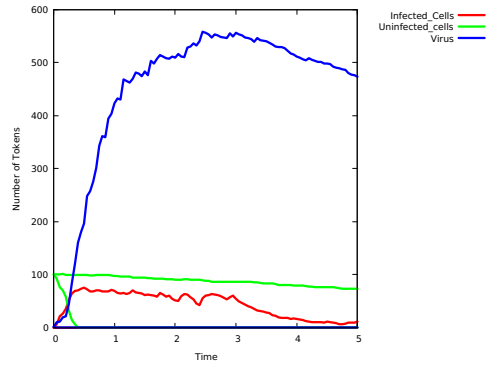
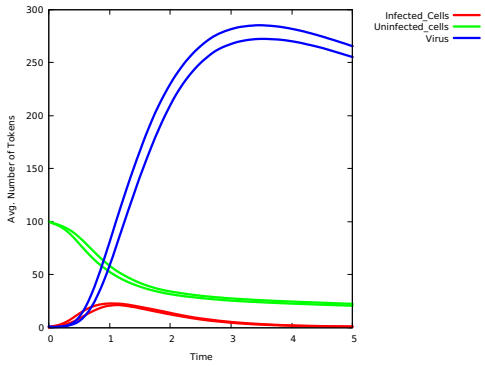


Figure 3.16: The $\mathcal{FSPN}$ model of the Virus Infection. The fuzzy kinetic parameter k_infect is assigned to the transitions *infection* (coloured in red).

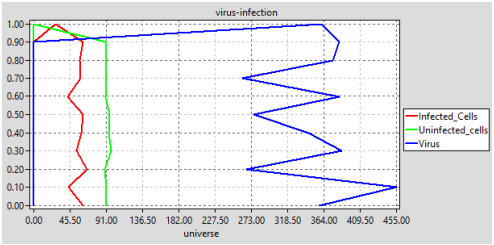
ness (stochasticity) induced by the system. Let us discuss the behaviour of the virus as an example according to its fuzzy band shown in Sub-figure 3.17b. The fuzzy band of the variable *Virus* describes the uncertainty range of the virus development (its population) over time. The population the virus gets increased overtime due to its replication. Sub-figure 3.17d shows the lower and upper bounds of the virus population at time point 16, which are 14 and 24 (approximately), respectively.



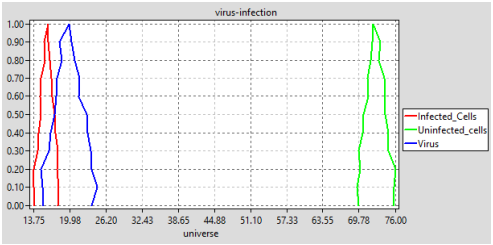
(a) Fuzzy band of variables (single stochastic run per each sample).



(b) Fuzzy band of variables (averaged over 10 000 runs).



(c) Membership functions did not formed using single run per each sample).



(d) Membership function of variables at time point t=16 (averaged over 10 000 runs).

Figure 3.17: Fuzzy stochastic simulation results of the virus infection ($\mathcal{FSPN}$). The number of α levels is 10 and the number of samples per each level is 10. The fuzzy kinetic parameter is k_infect with (0.2,0.3,0.5). The sampling strategy is Basic Sampling.

3.10.2 Decay Dimerization

This model consists of a degradation reaction (*r1*), one reversible dimerization reaction (modelled by *r2* and *r3*), and the transition *r4* describing the reaction which produces the species *S3* from the species *S2* [LCHS18]. The $\mathcal{FCPN}$ model is shown in Figure 3.18. Table 3.4 gives the rate functions of the transitions.

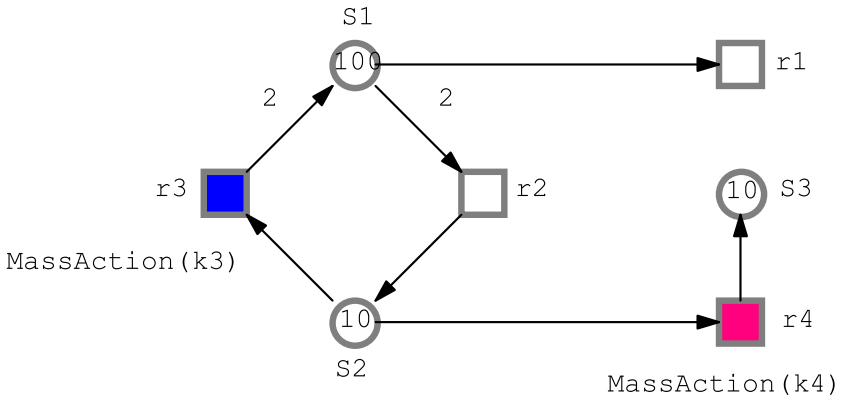


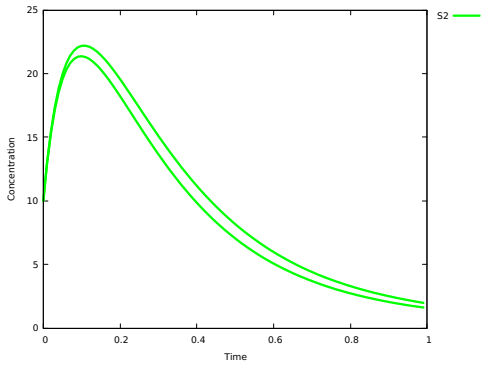
Figure 3.18: The $\mathcal{FCPN}$ model of the Decay Dimerization. The fuzzy kinetic parameters k_3 and k_4 are assigned to the transitions *r3* and *r4* (marked in blue and red, respectively).

Table 3.4: decay-dimerization network $\mathcal{FCPN}$- rate functions of transitions, all following mass/action kinetics.

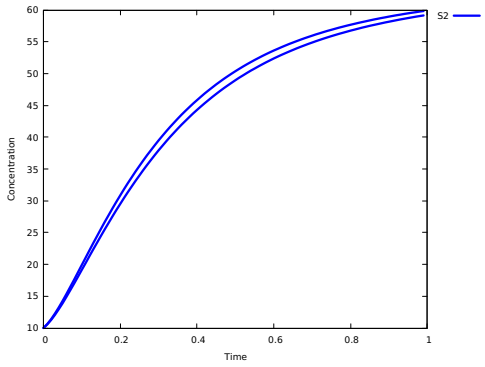
Transition r_i	Rate function	Kinetic constant k_i
r_1	$k_1 \cdot S1$	$k_1 = 0.2$
r_2	$k_2 \cdot (S1 + S1)$	$k_2 = 0.04$
r_3	$k_3 \cdot S2$	$k_3 =$(0.45, 0.5, 0.55)
r_4	$k_4 \cdot S2$	$k_4 =$ (4.9, 5.0, 5.4)

Figure 3.19 presents fuzzy simulation traces. Sub-figures 3.19a and 3.19b give the fuzzy bands of the variables (places) *S2* and *S3*, respectively. Each band describes the uncertainty range of the concentration of each variable over time. Sub-figures 3.19c and 3.19d present the membership functions of the variables *S2* and *S3*, respectively. Each provides precise information about the associated uncertainties, for instance, the membership function of the variable *S2* at time point 31 reports that the minimum

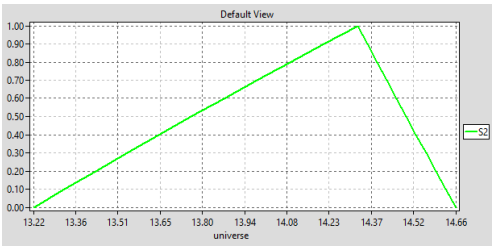
bound of the species concentration is 13.22, whereas the maximum bound is 14.66, and the corresponding values for the species $S3$ are 29.60 and 30.77, respectively.



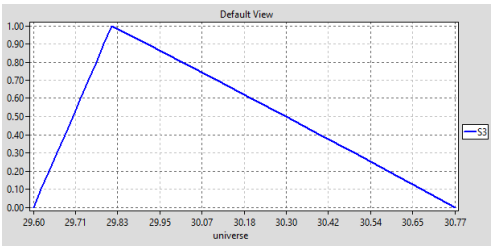
(a) Fuzzy band of the variable *S2*.



(b) Fuzzy band of the variable *S3*.



(c) Membership function of the variable *S2* at time point t= 31.



(d) Membership function of the variable *S3* at time point t= 20.

Figure 3.19: Fuzzy continuous simulation results of the Decay Dimerization . The number of α levels is 10, and the number of samples per each level is 10. The fuzzy kinetic parameters are k_3 =(0.45, 0.5, 0.55) and k_4 =(4.9, 5.0, 5.4). The *LHS* sampling method is chosen as sampling strategy.

3.10.3 Yeast Polarization

Yeast Polarization [DLPK10, DRGP11a, CBL17] comprises a set of biochemical reactions involving seven species describing the pheromone induced *G-protein* cycle in Saccharomyces cerevisiae [DRGP11b]. We model and simulate this system by means of $\mathcal{FHPN}$, see Figure 3.20. The places *R, L,* and *RL* represent the pheromone receptors, ligands, and receptor-ligand complexes, respectively. *The place G* represents the *G-protein*, and *Ga, Gd* and *G_bg* represent three separate units. The ligands *L* bind with the receptors *R* to form complexes *RL*. See [DLPK10] for more details.

This model [LHY16] consists of eight reactions. The continuous net elements are divided into three continuous transitions (*r1, r2* and *r6*) and three continuous places (*R*, *G_a* and *G_d*), while the rest of the net elements are either discrete places or stochastic transitions. In this case study, the reactions *r6* and *r8* have non-complete knowledge about the kinetic parameter; thus they got assigned fuzzy kinetic parameters represented as triangular fuzzy numbers. The entire reactions of this model are given in Table 3.5. Figure 3.20 gives the corresponding $\mathcal{FHPN}$ model.

Figure 3.21 presents the fuzzy bands and membership functions of two variables. Both fuzzy bands describe how the lower and upper bounds of selected variables develop over time according to the uncertainties caused by uncertain kinetic parameters. We notice that the bands of the selected places intersect between the period ranging from 16 and 30 (of simulation time), this is also reflected by the membership functions at time point 20. As our model has two fuzzy kinetic parameters, we chose the *LHS* sampling strategy to reduce the total number of required simulations to be performed, which will be 101 runs instead of 1001 runs (using basic sampling).

Table 3.5: Yeast polarization $\mathcal{FHPN}$- rate functions of transitions, all following mass/action kinetics.

Transition r_i	Rate function	Kinetic constant k_i
r_1	k_1	k_1 =0.38
r_2	$k_2 \cdot R$	k_2 = 0.04
r_3	$k_3 \cdot (L + R)$	k_3 =0.082
r_4	$k_4 \cdot RL$	k_4 =0.12
r_5	$k_5 \cdot (RL + G)$	k_5 = 0.12
r_6	$k_6 \cdot G_a$	k_6 =(0.08, 0.1, 0.12)
r_7	$k_7 \cdot (G_d + G_b g)$	k_7 =0.005
r_8	k_8	k_8 =(10, 13.21, 15)

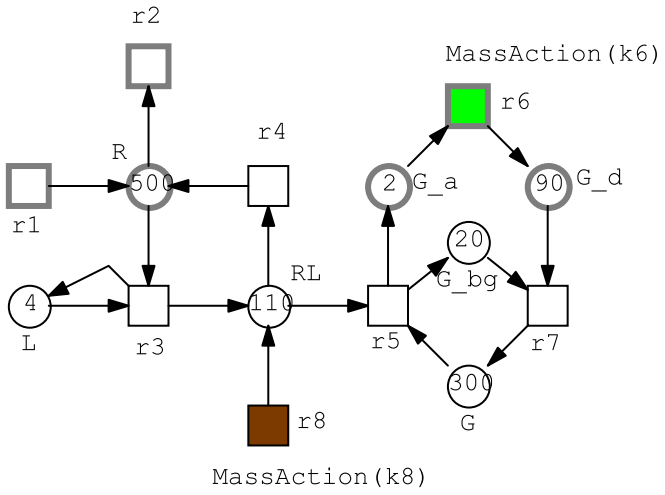
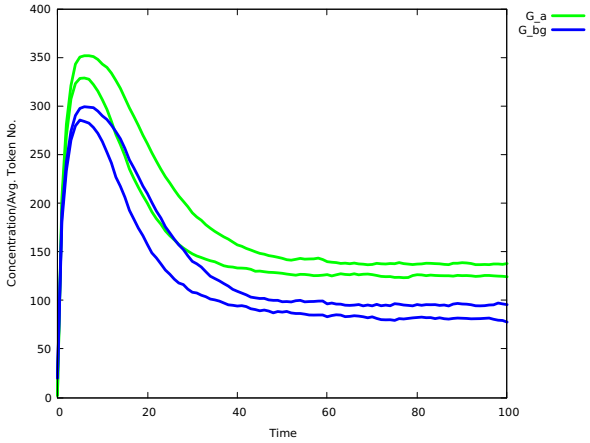
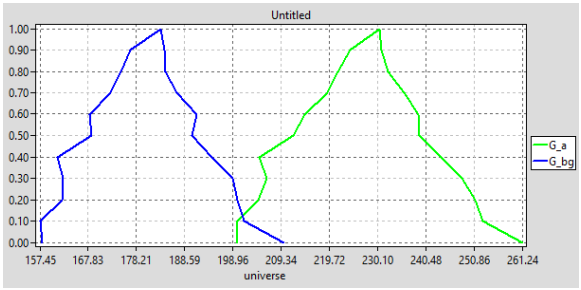


Figure 3.20: The $\mathcal{FHPN}$ model of the Yeast Polarization. The fuzzy kinetic parameters k_6 and k_8 are assigned to the transitions *r6* and *r8* (coloured in green and brown, respectively). The deterministic net elements are: the places R, G_a and G_d and the transitions $r1$, $r2$ and $r6$. The other remaining net elements are discrete places and stochastic transitions.



(a) Fuzzy bands of the variables G_a and G_bg.



(b) Membership functions at time point t= 20.

Figure 3.21: Fuzzy hybrid simulation results of the yeast polarization. Th number of α levels is 10 and the number of samples per each level is 10. *LHS* sampling method is chosen as sampling strategy. The number of stochastic runs (for the stochastic part) is 20 runs.

3.10.4 Repressilator

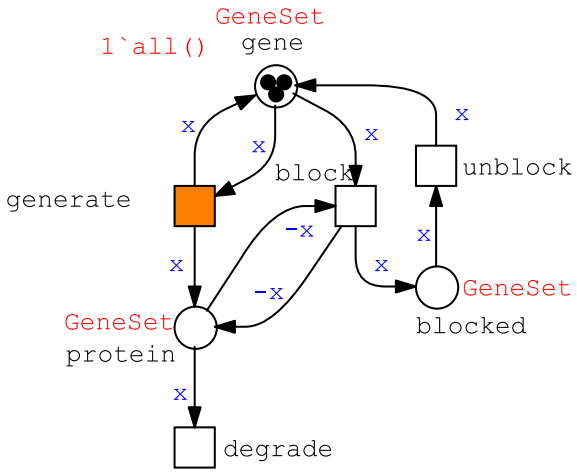
Repressilator is a synthetic genetic regulatory circuit known for its oscillation behaviour [EL00]. It comprises at least three genes, e.g. **a, b** and **c**, each blocking the next gene in a cyclic way [BCP08]. The coloured model comprises three places, each gets assigned the colour set *Geneset* encoding three colours **a, b** and **c**. The place *gene* is initialised with one token of each colour by using the colour expression 1‘*all*(). The set of transitions are *generate*, *block*, *unblock* and *degrade*. The transition *generate* gets a fuzzy kinetic parameters (*k_gene*) represented as TFN in its rate function, while the remaining transitions have crisp kinetic parameters in their rate functions; for more details about transitions' rate functions; see Table 3.6, and see Figure 3.22 for the entire $\mathcal{FSPN}^{\mathcal{C}}$ model.

Figure 3.23 presents fuzzy simulation results of some unfolded places using a different number of stochastic simulation runs. Due to the stochasticity induced by the system, the number of stochastic runs has an influence on the formed fuzzy band and timed-membership functions i.e. increasing number of stochastic runs gives better shapes of fuzzy bands and timed membership functions. Please note that averaging simulation traces over a high number of runs removes the oscillation behaviour from traces.

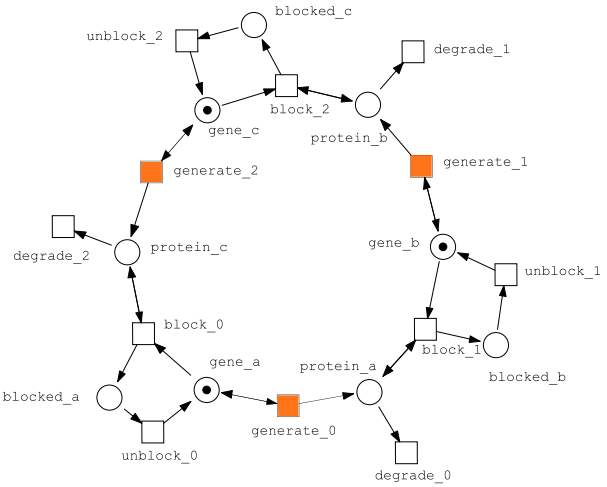
The output fuzzy bands of the unfolded places *a, b* and *c* report that the fuzzy kinetic parameter *k_gen* causes more uncertainties occurring between the minimum and maximum curves of each individual band, while the system evolves over time. Furthermore, the proteins *a, b* and *c* have the lower bound of 28.01 and their upper bounds are 85.37, 84 and 84, respectively, as reported by their timed membership functions at time point 32, see Sub-figure 3.23d.

Table 3.6: Repressilator $\mathcal{FSPN}^{\mathcal{C}}$- the transitions' firing rate functions.

Transition	Rate function	Kinetic parameter
generate	$k\_gen \cdot gene$	k_gen = (0.1, 0.1, 0.15)
degrade	$k\_deg \cdot protein$	k_deg = 0.001
blocked	$k\_block \cdot protein$	k_block = 1
unblocked	$k\_unblock \cdot blocked$	k_unblock = 0.0001

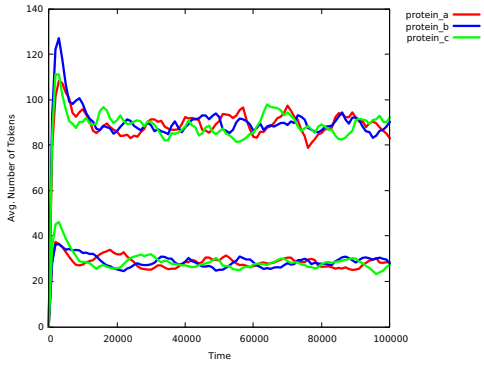


(a) The $\mathcal{FPN}^{\mathcal{C}}$ model of the Repressilator.

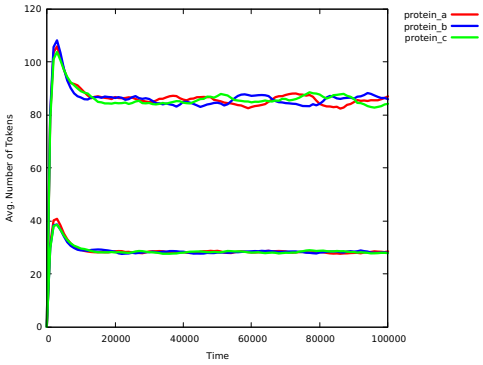


(b) The automatically unfolded $\mathcal{FSPN}$, layout automatically generated with the *FMMM* algorithm built-in Snoopy by help of the OGDF library [OGD].

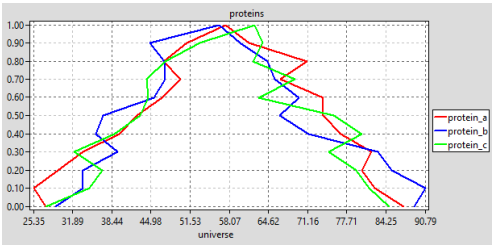
Figure 3.22: The $\mathcal{FSPN}^{\mathcal{C}}$ model of the Repressilator together with its unfolded version, adopted from [LH14]; The colour declarations: *colorset enum GeneSet = {a, b, c};* and *variable x: GeneSet*. The transition *generate* is marked with the orange colour indicating that it has a fuzzy kinetic parameter in its rate function.



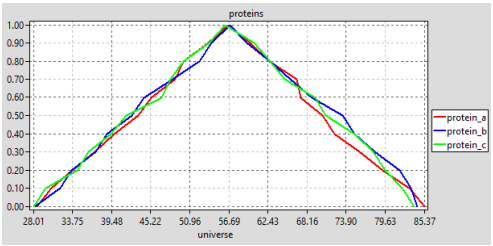
(a) Fuzzy band of the unfolded places (averaged over 500 runs).



(b) Fuzzy band of the unfolded places (averaged over 10000 runs).



(c) Membership function of the variables at time point t=32 (averaged over 500 runs).



(d) Membership function of the variables at time point t=32 (averaged over 10000 runs).

Figure 3.23: Fuzzy stochastic simulation results of the Repressilator. The number of α levels is 10 and the number of samples per each level is 10. The kinetic parameter is k_gen with (0.1, 0.1, 0.15). The used sampling strategy is basic sampling.

3.10.5 Membrane Systems

Membrane computing is a branch of molecular computing that aims to develop models and paradigms that are biologically motivated [Iba05]. Membrane systems (also known as P systems) are a very powerful computational modelling language, as they are able to model complex biological phenomena due to their modularity and their ability to enclose the evolution of different environments and different interrelated processes [DRQ+20].

Figure 3.24 presents the general structure of a membrane system. Each membrane (compartment) may contain other membranes and components (also called objects) that either reside in or translocate between these membranes. A membrane's object can be any reactant, i.e. molecule, gene, protein, etc. Furthermore, the *skin* membrane (outer membrane) is the one which separates the system from its environment [PR02]. An *elementary* membrane is the one which does not contain any further membrane. In membrane systems, there may be some membranes with the same label in the same level of hierarchy, i.e. two copies with the same label may exist in one compartment. Therefore, in order to differentiate each copy, we have to use integer numbers as unique identities [LH13]. Then each membrane will be encoded as a tuple, e.g. (ID, label). In Figure 3.24, the membranes (m1, l2) and (m2, l2) have the same label l2, thus the identifiers m1 and m2 have to be used to differentiate them. Note that membrane labels refer to the types of compartments (e.g., cell membrane and other membranes of cellular compartments such as chloroplasts, mitochondria, vacuoles etc.), whereas membrane identifiers refer to the copy number of the cell or cellular compartments.

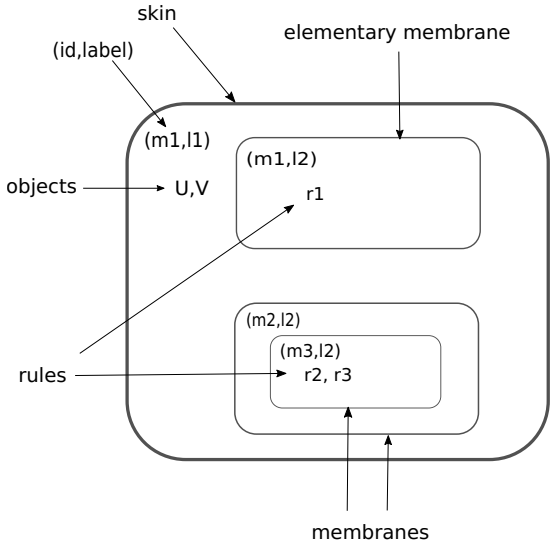


Figure 3.24: The general structure of a membrane system [AHF22].

A membrane structure μ is represented as a string of matching parentheses [LH13]. Each pair of matching square parentheses [] is called a membrane, e.g. [[]] denotes two nested membranes. Besides, each membrane is associated with a label, e.g. $[_1[_2]_2]_1$. For simplicity, we omit the label of the left parentheses, e.g. $[[]_2]_1$. The membrane's degree m is defined as the number of membranes in a given membrane system; e.g. $m = 2$ for the former example. Alternatively, the membrane structure μ with the degree $m \geq 1$ can be represented as a (rooted) tree, whereby the root node represents the skin membrane and the leaf nodes the elementary membranes.

Note that objects (components) are represented by symbols from a given alphabet. Tanslocation and/or transformation (interconversion) of the objects occurs according to certain rules which are called developmental rules. These rules are specific for a certain compartment or pairs of compartments. Note that developmental rules have many forms, determining either the transformation or the translocation of the membrane's objects. The membrane structure and the multisets of objects in the membranes define a configuration of a membrane system. The initial configuration is given by the membrane structure and the multisets of objects available in their membranes at the beginning of a computation.

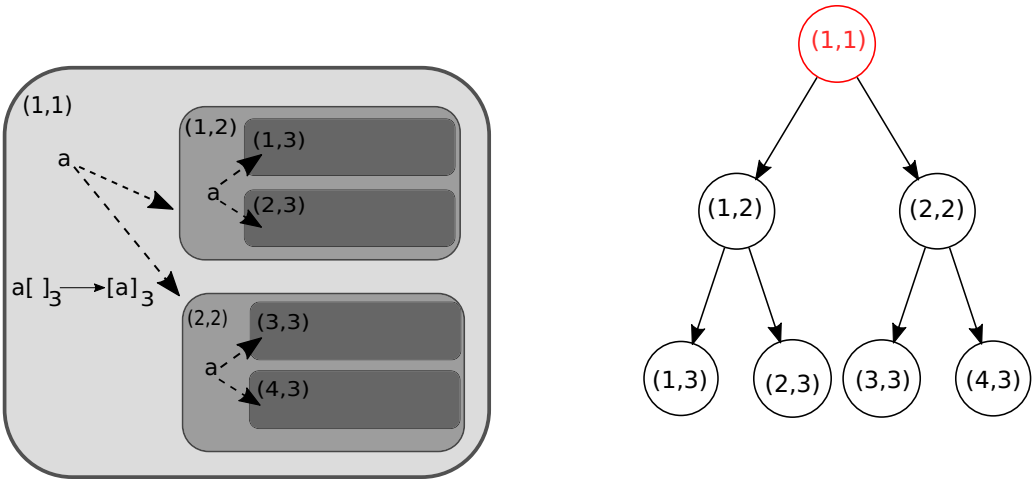
The system's rules include membrane creation, dissolution or division [LH13], according to which the membrane system changes its structure, e.g. increase or decrease the number of membranes. In this case, the membrane system is called a *dynamic membrane system*; otherwise, it is called a *static membrane system*, i.e. with *In-communication/Out-communicates rules* only permitting objects to be translocated within the system's hierarchy. In the following, we are going to model and analyse a static membrane system. Please note that *dynamic membrane systems* are subject to further work.

Each rule is associated with a stochastic kinetic parameter. Thus, a stochastic simulation algorithm can be utilised for simulating the system.

Figure 3.25 gives a fuzzy stochastic membrane system (static membrane system). The system comprises three membranes with two copies of the membrane with label 2 and four copies of the membrane with label 3. The skin membrane (1,1) is associated with the object *a*, while the other ones have no associated objects at the beginning of computation (initial state of the system). In this example, there is one *in-communication* rule which translocates the object *a* from membrane with label 1 to a randomly chosen membrane with label 3.

As a tree is a special case of a graph, we are going to exploit this idea to represent membrane systems with nested hierarchy. For this purpose, we introduce the *elemOf* (boolean) operation. The *elemOf* operation is a boolean operation motivated by the discrete space, represented by a directed graph, e.g. geographical map. Technically speaking, the *elemOf* takes two operands, the left-hand operand is a certain colour, while the right-hand one is a colour set, e.g. c *elemOf* CS. This operation returns *true* if the colour (in left-hand operand) belongs to the colour set (in right-hand operand);

otherwise, it returns *false*.



(a) A membrane system.

(b) The tree representing the structure of the given membrane system.

Figure 3.25: (a) Fuzzy stochastic membrane system. The membrane (1,1) (skin membrane) is associated with the object a, whereas the other membranes have no associated objects, the rule $a[]_3 \rightarrow [a]_3$ is an *in-communication* rule taking place in membrane (1,1), which will transfer the object a to an elementary membrane with label 3. Note that the object a has to be passed through the membrane with the label 2 to get into the membrane with label 3. (b) The structure of the membrane system represented as directed graph (tree). The red node refers to the root of the tree representing the skin membrane. The leaves of the tree represent the elementary membranes. The directed arcs refer to the direction of moving an object in the system.

To apply this modelling idea, we perform the following steps:

- We define an *enum* colour set encoding the vertices of the tree, e.g. *Nodes*. Each colour of this colour set encodes one vertex in the tree. Please note that a tree node corresponds to one membrane.
- A product colour set induced by performing the Cartesian product of the colour set *Nodes*, e.g. *Matrix*. Each element (tuple) of this colour set represents one possible connection (directed connection) between two graph vertices.
- To obtain the required connections between the graph vertices, we define a subset colour set, e.g. *Connections*, by constraining the product colour set *Matrix* using

a boolean expression, see Table 3.7.

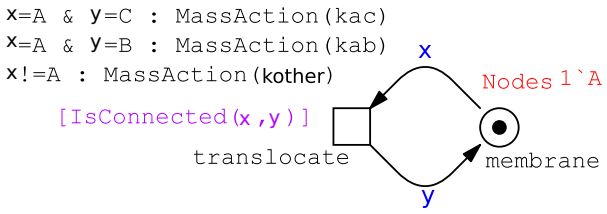
- To encode the system hierarchy, we add one coloured place, e.g. *membrane*. This place has to be defined on the colour set *Nodes*. Then, we define the initial marking of this place with one token of the colour A, e.g. 1‘A, as the membrane (1,1) has one associated object (here the a).

- To encode the rule together with the connectivity style, we add one coloured transition, e.g. *translocate*. This transition has to be connected using a loop (two opposite arcs) to the place *membrane*. Each arc is decorated with a variable (defined on the colour set *Nodes*). Then, We assign a guard to this transition to determine the directed connections between each two nodes, e.g. (a,b) *elemOf* Connections. Please note that we define this guard as a colour function, e.g. *IsConnected(a,b)*. As each coloured transition corresponds to a set of transition instances in the unfolded net, thanks to coloured-dependent rate, which allows to us to specify different rate functions to different transition instances. In this example, we are going to specify different rate functions to the transition instances representing the translocation of the object a from the membrane (1,1) (the skin membrane) to the membrane (1,2) and the translocation of the object a from the membrane (1,1) to the membrane (2,2).

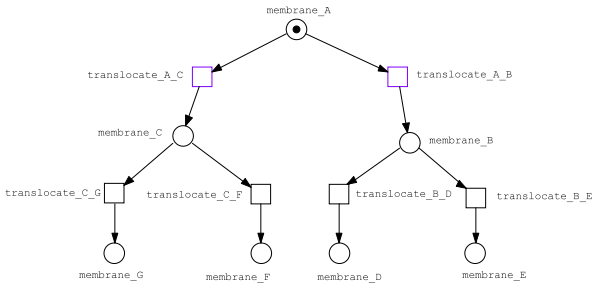
See Figure 3.26 for the coloured fuzzy stochastic Petri net model together with the unfolded version of this model. Please note that our modelling approach allows only to define one kind of object to be translocated in the system's hierarchy. We may overcome this by combining our modelling approach and the approach which is presented in [LH13].

Figure 3.27 gives the fuzzy simulation traces of the fuzzy stochastic membrane system shown in Figure 3.26a. As our model has two fuzzy kinetic parameters (*kab* and *kac*), we make use of the LHS sampling strategy to reduce the number of required runs. The discussion related to increasing the number of stochastic runs (which has been presented for the Repressilator) is also applied here as we have coloured fuzzy stochastic Petri net model.

The fuzzy bands sketched in Figure 3.27 describe the uncertain probability range, for which the object a may have the chance to stay in the membrane A (1,1) or to translocate to the membrane E (4,3). Figure 3.28 presents the timed-membership functions of the variable *membrane_A* at different time points. For instance, at time point 2 the lower bound of the probability that the object may stay at the membrane *membrane_A* is 0.14 and the upper bound value is 0.82.



(a) Coloured fuzzy stochastic Petri net model.



(b) The unfolded $\mathcal{FPN}$model.

Figure 3.26: (a) The coloured fuzzy stochastic Petri net ($\mathcal{FSPN}^{\mathcal{C}}$) model, see Table 3.7 for colour definitions. Note that we map each tuple representing (membrane id, label) into a letter encoded by enum type, e.g. the colour A corresponds the membrane whose id = 1 and label = 1, because colour ids are not allowed to have the symbol "_" (due to some inconsistencies). The parameters *kac* and *kab* are fuzzy kinetic parameters. (b) The unfolded fuzzy stochastic Petri net model, the transition instance *translocate_A_C* has the fuzzy kinetic parameter *kac* and the transition instance *translocate_A_B* has the fuzzy kinetic parameter *kab*.

Table 3.7: Declarations for the coloured model given in Figure 3.26a.

Type	Declaration
Constant	*TFN* k_{ab} = (0.1, 0.4, 0.9); //fuzzy kinetic parameter.
Constant	*TFN* k_{ac} = (0.1, 0.3, 0.61); //fuzzy kinetic parameter.
Constant	*double* k_{other} = 1.0; //crisp kinetic parameter.
Colorset	Nodes = *enum* with {A,B,C,D,E,F,G}; // graph vertices.
Colorset	Matrix = *product* with Nodes x Nodes; // All possible connections.
Colorset	Connections = *Matrix* with $(x = A\&(y = B\|y = C))\|(x = B\&(y = D\|y = E))\|(x = C\&(y = F\|y = G))$;// required connections between the graph nodes.
Variable	x :Nodes; y:Nodes;
ColorFunction	*bool* IsConnected(n Nodes, m Nodes) { ((n,m) *elemOf Connections*)}; // colour function checking a certain colour (n, m) whether it belongs to the colour set *Connections* or not.

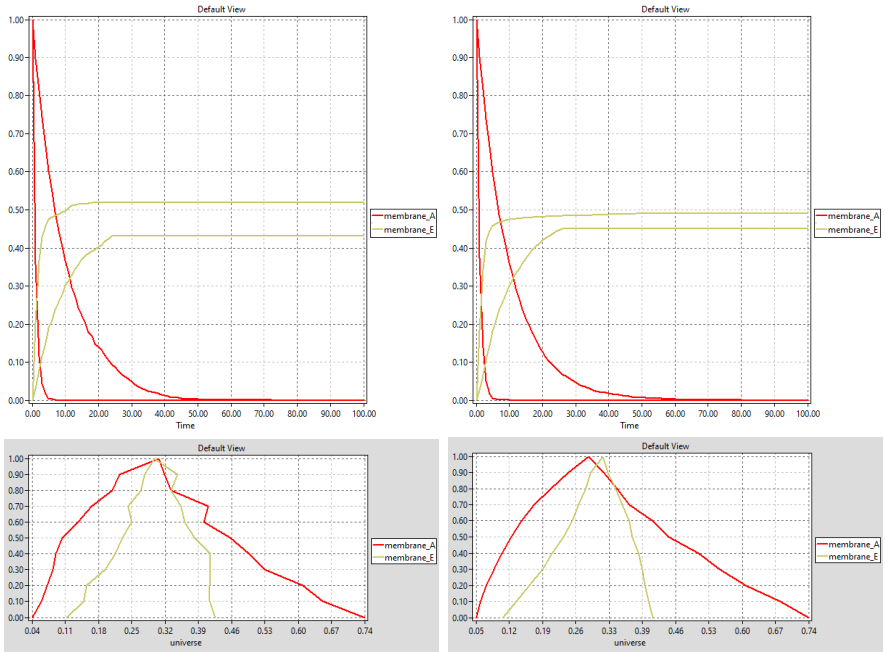


Figure 3.27: Fuzzy bands and timed-membership functions of the instance places *membrane_A* and *membrane_E* at time point $t = 3$. Fuzzy settings are: α-levels = 10; Number of sampling points/level = 10; sampling strategy = *LHS*; the number of stochastic runs: 1000 runs (first column), averaged over 10000 runs (second column).

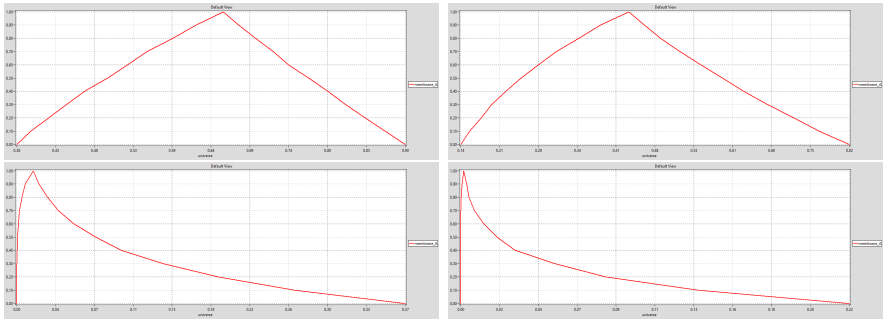


Figure 3.28: Timed membership functions of the place instance *membrane_A* at time points $t = 1, 2, 10$ and 15; from top left to bottom right. Fuzzy settings are: α-levels = 10; Number of sampling points/level = 14; sampling strategy = LHS; averaged over 6000 stochastic runs.

3.10.6 2D Diffusion

Here we consider an imaginary model [LAC21] to illustrate how to incorporate both stochastic and deterministic elements in one coloured model as $\mathcal{FHPN}^{\mathcal{C}}$. The scenario we consider is as follows. A gene has two states: active or inactive, which transit from one to the other with a given probability. When a gene is active, it can be transcribed into mRNA. Then, mRNA can either be degraded or translated into proteins, which may degrade or diffuse from one compartment to another [LBHY14]. We model this scenario in Figure 3.29. In this model, the gene and its state transitions are modelled using $\mathcal{SPN}$; whereas protein generation, degradation and diffusion are modelled using $\mathcal{CPN}$. The transition *translate* adopts the fuzzy kinetic parameter $k_{translate}$. We represent the space of the cell as a grid with 51 x 51 sub-volumes, each representing a compartment. Therefore, we define a tuple colour set with 2601 colours to represent 2601 compartments. Thus, we make use of coloured Petri nets to represent diffusion space. Table 3.9 presents the colour-related definitions and kinetic parameter values; also compare Table 3.8 for reaction rates.

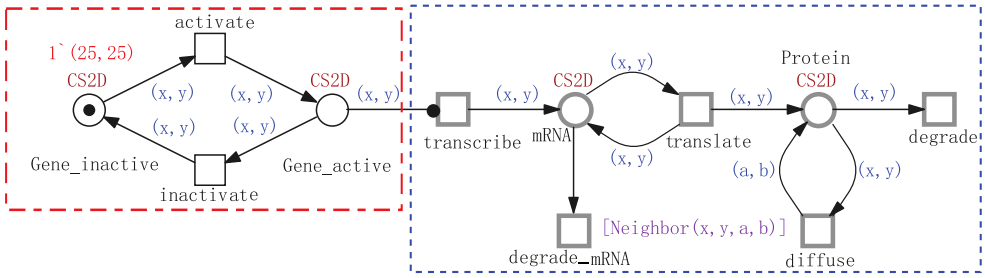


Figure 3.29: The $\mathcal{FHPN}^{\mathcal{C}}$ model of the Whole-cell modelling which includes diffusion. See Table 3.9 for the colour definitions.

Table 3.8: Rate functions of the $\mathcal{FHPN}^{\mathcal{C}}$ model.

Transition	Rate function	Kinetic parameter
activate	k1 × Gene_inactive	0.3
inactivate	k2 × Gene_active	0.1
transcribe	k3 × Gene	0.1
degrade_mRNA	k4 × mRNA	0.001
translate	k5 × mRNA	(0.5, 1.0, 1.5)
diffuse	k6 × Protein	1
degrade_protein	k7 × Protein	0.001

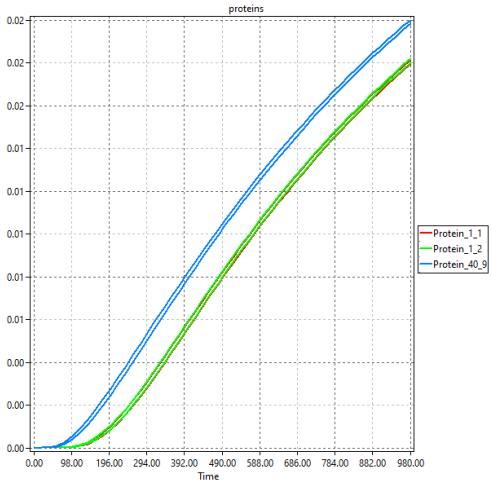
Figure 3.30 gives the fuzzy simulation traces of some variables. We notice that the fuzzy bands of the variables *Proteins_1_1* and *Proteins_1_2* are somewhat iden-

Table 3.9: Declarations for the model given in Figure 3.29.

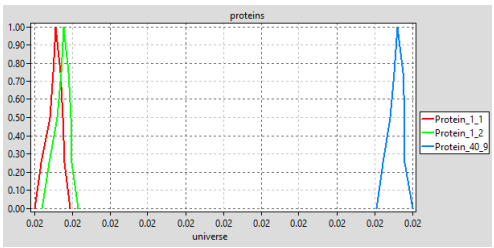
Type	Declaration
Constant	*int* $D1 = 51$; //an integer constant used as a scaling factor.
Constant	*int* $D2 = D1$; //an integer constant used as a scaling factor.
Colorset	XDim = *int* with 1 - $D1$; // space of X coordinates.
Colorset	YDim= *int* with 1 - $D2$; // space of Y coordinates.
Colorset	CS2D = *product* with XDim x YDim; // 2D Grid.
Variable	x :XDim; y:YDim;
ColorFunction	*bool* Neighbor(x XDim, y YDim, a XDim, b YDim) { (a=x | a = x+1 | a = x-1) & (b=y | b = y+1 | b = y-1) & ((a=x & b=y)) & ($a <= D1$ & $b <= D2$) & ($a >= 1$ & $b >= 1$) }; // colour function checking whether two points are neighbors or not.

tical; this is also reflected by the corresponding timed-membership functions; while Figure 3.31 presents the fuzzy band and corresponding timed-membership function of the unfolded place *mRNA_25_25*. It is worth mentioning that the size of the unfolded model is quite large as follows: 10,404 places, 33,205 transitions, 61,208 standard arcs and 2,601 read arcs. Thus, obtaining our fuzzy results consumed about half a day of runtime, and this would be much worse when we average over many stochastic runs.

Figure 3.32 and Figure 3.33 present the 2D minimum and maximum simulation plots of the fuzzy simulation results, which reflect the concentration of proteins as they evolve over space and time.

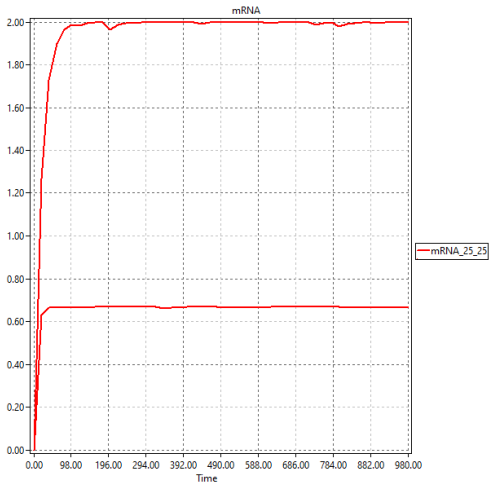


(a) Fuzzy bands of some unfolded variables.

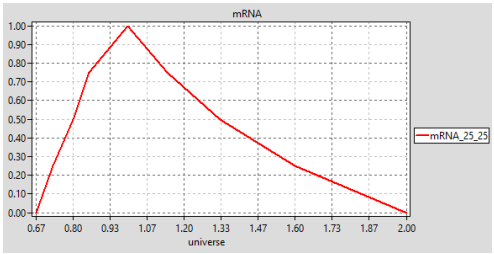


(b) Membership functions of some unfolded variables at time point t=47.

Figure 3.30: Coloured fuzzy hybrid simulation of the $\mathcal{FHPN}^{\mathcal{C}}$ model shown in Figure 3.29. The number of α-levels and number of sampling points per each level is 5.

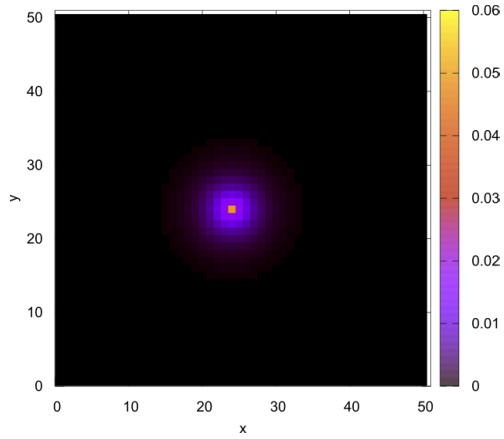


(a) Fuzzy band of the unfolded place $mRNA\_25\_25$.

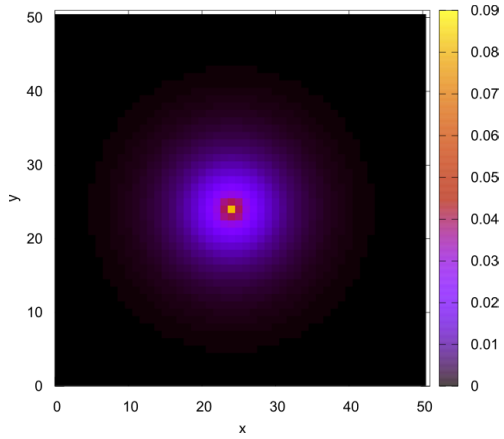


(b) Membership function of the unfolded place $mRNA\_25\_25$ at time point t=30.

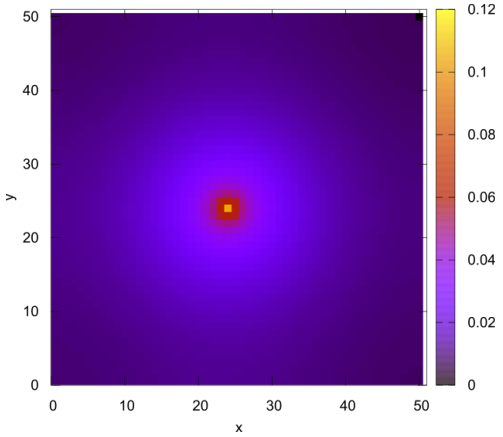
Figure 3.31: Coloured fuzzy hybrid simulation of the $\mathcal{FHPN}^{\mathcal{C}}$ model shown in Figure 3.29. The number of α-levels and number of sampling points per each level is 5.



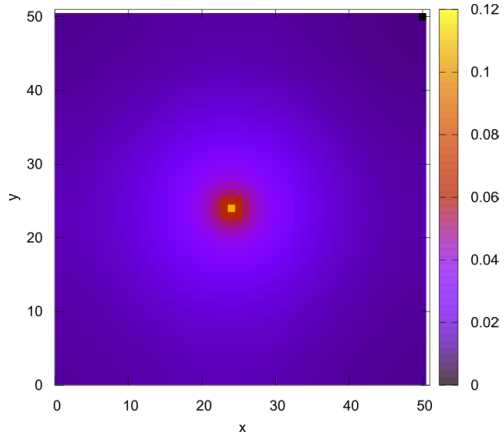
(a) 2D simulation plot of the protein concentration at each compartment at time point 20.



(b) 2D simulation plot of the protein concentration at each compartment at time point 100.

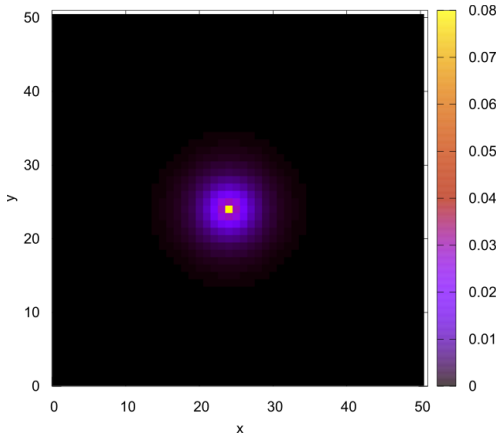


(c) 2D simulation plot of the protein concentration at each compartment at time point 700.

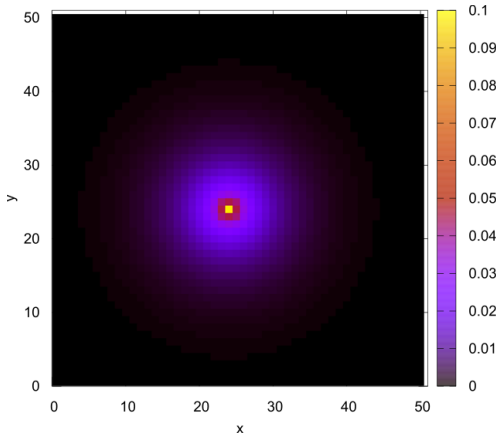


(d) 2D simulation plot of the protein concentration at each compartment at time point 1000.

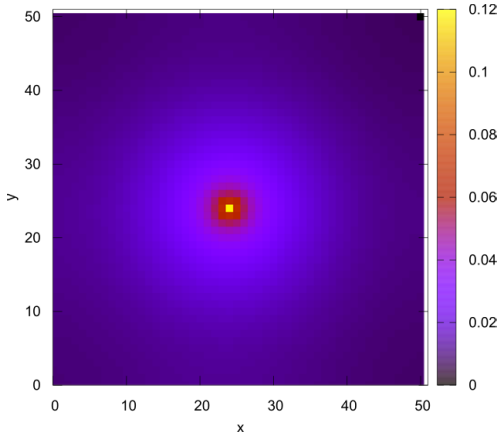
Figure 3.32: 2D simulation plot of minimum fuzzy traces of proteins at different points of the simulation time.



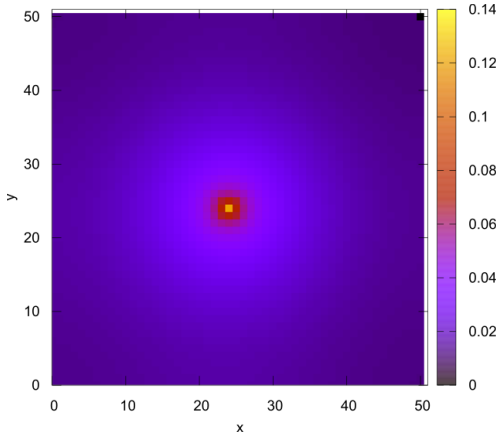
(a) 2D simulation plot of the protein concentration at at time point 20.



(b) 2D simulation plot of the protein concentration at time point 100.



(c) 2D simulation plot of the protein concentration at time point 700.



(d) 2D simulation plot of the protein concentration at time point 1000.

Figure 3.33: 2D simulation plot of the maximum fuzzy traces of proteins at different points of the simulation time.

3.11 Some Performance Results

In this section, we present some experimental results for the fuzzy simulation. We compare fuzzy simulation runtime for different fuzzy settings. Then we explore the accuracy of the obtained results using different simulation settings.

In the following, we sketch some performance measures, which describe how much burden to expect when performing the fuzzy simulation for both uncoloured fuzzy Petri nets and coloured ones. Table 3.10 presents simulation runtime for *Yeast Polarisation* (see Section 3.10.3) using different settings. In this model, fuzzy simulation has been averaged over 20 runs for the stochastic part of the model. Choosing the *LHS* as a sampling strategy, simulation always finishes first, even for a substantial number of both α-levels and samples/level. This result is expected as the *LHS* sampling strategy considerably reduces the number of simulation runs, when we have more than one fuzzy kinetic parameter in the model.

Table 3.10: Fuzzy simulation runtime for the Yeast Polarisation ($\mathcal{FHPN}$).

Input		simulation time		
Levels J	Samples N	Basic	Reduced	LHS
10	10	3.49 m	1.41 m	22 s
15	20	5.36 m	3.38 m	55 s
18	30	1.40 h	29 m	2.12 m
20	40	2.17 h	38 m	3.7 m
50	50	14h	12 h	25 m
75	75	◇	21 h	43m

* ◇ simulation did not finish within 24 hours. Done on PC, Intel(R) CPU 1.80GHz, RAM 32.00GB.

For coloured fuzzy Petri nets, we choose the 2D diffusion system. For more details about the size of the unfolded model and the kinetic data, please check Section 3.10.6. Table 3.11 gives the fuzzy simulation runtime for the three sampling strategies. We notice that the fuzzy simulation finishes first using the *Reduced sampling* strategy, because the model has only one fuzzy kinetic parameter which means simulation has to be done using the samples at the level $\alpha = 0$. However, fuzzy simulation lasts for the same time for the other two strategies, because both strategies give the same number of samples when the model has only one fuzzy kinetic parameter. Moreover, we notice that *LHS sampling* slightly gives less simulation runtime than the *basic sampling* which could confirm that the procedure of discretising the fuzzy number using *LHS sampling* is faster than the *basic sampling* due to some implementation aspects.

Overall, the number of involved fuzzy kinetic parameters together with the number of levels and number of samples at every level have an influence on the number of simulations to be performed, and thus on the total simulation time.

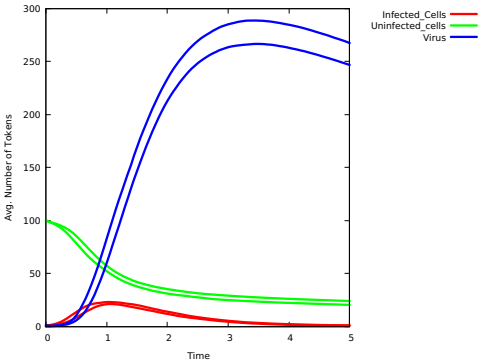
Figure 3.34 and Figure 3.35 show fuzzy simulation results for the virus infection

Table 3.11: Fuzzy simulation runtime for the 2D Diffusion system ($\mathcal{FSPN}^{\mathcal{C}}$).

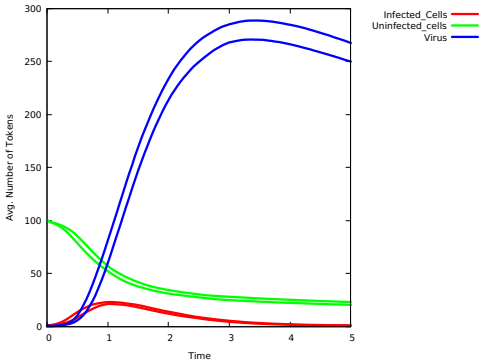
Input		simulation time		
Levels J	Samples N	Basic	Reduced	LHS
4	4	20.25 m	13.4 m	20.19 m
15	10	5.36 m	30.52 m	48.30 m
7	15	2.24 h	31.48 m	2.12 h
10	10	7.23 h	26.12 m	7.11 h
20	20	◇	26.12 m	◇

* ◇ simulation did not finish within 24 hours. Done on PC, Intel(R) CPU 1.80GHz, RAM 32.00GB.

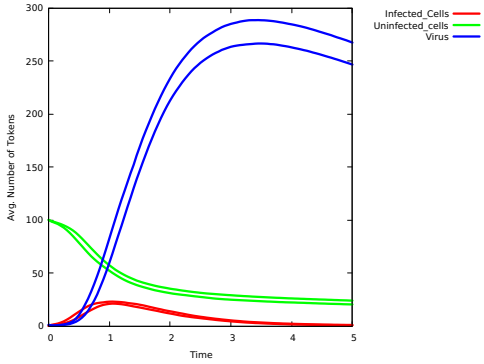
model with changing the sampling strategy. We notice that the three sampling strategies gave us somewhat identical shapes of fuzzy bands and membership functions. According to our experience, theses three strategies behave well. Which to choose depends on the number of fuzzy kinetic parameters that exist in the model on hand, as we have seen above.



(a) Fuzzy band of variables - basic sampling strategy.

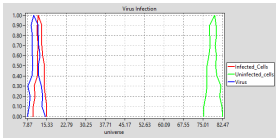


(b) Fuzzy band of variables - reduced sampling strategy.

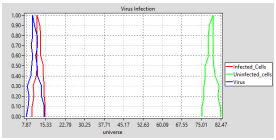


(c) Fuzzy band of variables - LHS sampling strategy.

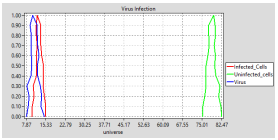
Figure 3.34: Fuzzy stochastic simulation results of the virus infection ($\mathcal{FSPN}$). The number of α levels is 10 and the number of samples per each level is 10. The fuzzy kinetic parameter is k_infect with (0.2,0.3,0.5).



(a) basic sampling.



(b) reduced sampling.



(c) LHS sampling

Figure 3.35: Membership functions of the variables for virus infection at time point 11. The fuzzy kinetic parameter is k_infect with (0.2,0.3,0.5).

3.12 Closing Remarks

In this chapter, we presented various classes of fuzzy quantitative Petri nets supported in Snoopy comprising coloured and uncoloured ones. These Petri net classes are very useful when kinetic parameters of the modelled system can not be estimated or measured precisely. Among those fuzzy Petri nets, coloured fuzzy Petri nets are particularly useful for modelling and simulating scalable models, e.g. biological models, involving several copies of a given component, while having at the same time uncertain kinetic parameters. Moreover, we illustrated three sampling techniques for obtaining crisp values of the fuzzy kinetic parameters. Among them, *LHS* is an efficient sampling method trying to minimise the required number of simulations. Moreover, we briefly sketched the implementation principle of fuzzy Petri nets in Snoopy. Finally, we presented some experimental results demonstrating how much burden could one expect when performing fuzzy simulation.

In our framework, we choose to represent biochemical networks as Petri nets. However, our approach can be equally applied to any related modelling approach for biochemical reaction networks involving fuzzy kinetic parameters.

Future work will include the generation of configuration scripts to delegate the time-consuming simulation step to the command-line tool Spike [CH19], which will run the simulations in a parallel manner and possibly remotely on a server. Finally, so far Snoopy supports only TFNs; we consider the option to support other types of fuzzy numbers, e.g. trapezoidal fuzzy numbers.

4 Some Implementation Aspects

4.1 Introduction

In this chapter, we sketch some implementation aspects, which are related to our Petri nets modelling and simulation tool Snoopy. The major contributions include harmonising coloured Petri nets in Snoopy [LHR12a] with the *CANDL* format [ACR+21] and with their uncoloured counterparts (in Snoopy) as well. Harmonising coloured Petri nets is important, as it overcomes previously existing inconsistencies due to some implementation issues, for example, some operators of colour expressions were supported by the *CANDL* format, but not by Snoopy. Thus, such a step is crucial to unify the usage of coloured Petri nets in all our tools. Moreover, we now support new features for dealing with user-defined declarations in both uncoloured and coloured Petri nets by exploiting the causal dependencies among Petri net declarations, e.g. cleaning unused declarations.

This chapter is organised as follows: In Section 4.2 we introduce the *CANDL* format for coloured Petri nets and we present previously existing inconsistencies, we then present how to deal with these inconsistencies. Moreover, we introduce new operations for colour expressions such as (--) and *elemOf* which increase the modelling power of coloured Petri nets. Furthermore, we present some algorithms reflecting the actual implementation of what has been done. Due to the lack of biological case studies for demonstrating some of these new operations, we illustrate their usage by some popular teaching examples for concurrent systems, e.g. the mutual exclusion problem. We then introduce declaration dependencies in Petri nets and how to represent these dependencies by means of dependency graphs in Section 4.3. After that, we present two application scenarios illustrating the usage of declaration dependencies for keeping the model on hand consistent (valid), for example, to avoid removing declarations which are already used in the model in an indirect way.

Finally, we present Snoopy's command-line feature and its advantages in Section 4.4.

4.2 Harmonisation of Coloured Petri Nets

As we have seen, coloured Petri nets have a set of user-defined declarations which may be annotated at different parts of the model, i.e. places, transitions and arcs. These annotations are crucial to obtain the desired model to be built. The coloured abstract net description language (CANDL) is a human and machine readable exchange format for different types of coloured Petri nets [ACR+21]. This format is used as a communication mean (intermediate language) between the family of *PetriNuts* tools [Pet], e.g. Snoopy, MARCIE [HRS13] and Spike [CH18] are able to communicate via *CANDL*.

Software harmonisation is a software engineering aspect. In our context, we mean by harmonisation to unify the definition of coloured Petri nets in Snoopy [HHL+12] as they are supported by the CANDL format. This includes unifying the usage of constants to be used in the same way as in uncoloured Petri nets, which has a big advantage of developing scaleable models.

We would like to draw the reader's attention to that we use American English for the naming convention of the key words, e.g. *colorset*, whereas we use the British counterparts in the text describing these key words, e.g. *colour set*.

4.2.1 Constants

Constants in coloured Petri nets can be characterised into two flavours; the first flavour is to use them as kinetic parameters, whereas the second one is to use them for colouring purposes; please note that the constants representing kinetic parameters of transitions' rate functions (quantitative coloured Petri nets) were previously dealt with as parameter nodes (ellipses), which means to add one constant, a new parameter node has to be added to the canvas. For large models, this makes the final model untidy and will have an influence on the size of the $\mathcal{PN}$ model file, as the graphics' information have to be kept as well. Thus, harmonising constants to be used in the same way as in uncoloured Petri nets is an important step.

A constant can be a specific value or a mathematical expression which may involve pre-defined constants. Each constant has to be associated with a named group. Each constant group can have one value set or many value sets. The default value set is the *Main* value set. Coloured Petri nets in Snoopy support by default four pre-defined groups. Table 4.1 sketches these groups together with their usage.

Each constant has a data type determining the type of the constant value; see Table 4.2 for the available data types in coloured Petri nets as they are supported by Snoopy.

Exporting our running example (food chain) to *CANDL* format gives the *CANDL* file shown in Listing 1. All constants have two value sets, each of which gets allocated to one of the following groups : *all, coloring, marking* and *parameters*. For example, the constant *SIZE* belonging to the group *coloring* is used to scale the model by using

Table 4.1: Pre-defined constant groups in Snoopy.

Group Name	Usage
all	constants belonging to the all group are used for all purposes
coloring	constants belonging to the coloring group are used for the purpose of scaling the model
marking	constants belonging to the marking group are used for initialising the marking of places
parameter	constants belonging to the parameter group are used as kinetic parameters

Table 4.2: Constant data types in Snoopy's $\mathcal{PN}^{\mathcal{C}}$.

$\mathcal{PN}$ class	Data types
Qualitative coloured Petri nets	*int* - *bool*
Quantitative coloured Petri nets	*int* - *bool* - *double*
Fuzzy Quantitative coloured Petri nets	*int* - *bool* - *double* - *TFN*

it in the definition of the colour set *CS*. Thus, the size of the unfolded model (number of unfolded Petri net elements) will be determined depending on the chosen value set of the *coloring* group. It is worth mentioning that exporting a coloured Petri net into the uncoloured counterpart will not export the constants belonging to the *coloring* group as they are only used for the purpose of scaleability.

Listing 1: Coloured food chain model in **CANDL** format.

```
colspn  [food_chain]
{
constants:
/*
Value sets
*/
  valuesets[Main:V_Set_0]

  /*
grouping of constants
*/
all:
  int Births = [2:3];
coloring:
```

```
15    int SIZE = [3:5];
16  marking:
17    int N = [10:20];
18    int M = [N:20];
19  parameter:
20    double kc1 = [.01:0.001];
21    double kc2 = [0.02:0.03];
22    double kc3 = [.03:0.01];
23    double kd = [.4:0.1];
24    double kr = [.5:0.1];
25
26  colorsets: /* colour sets*/
27    Dot = {dot};
28    CS = {1..SIZE};
29
30  variables:
31    CS : x; /*  variable defined on the colour set CS */
32
33  /*
34  discrete places
35  */
36  places:
37  discrete:
38    CS Prey = N`1;
39    CS Predator = M`all;
40
41  /*
42  stochastic transitions
43  */
44  transitions:
45    consume_sp
46      :
47      : [Predator + {Births`x}]
48       & [Predator - {[x>1](x++(-x))++[x=1]x}]
49       & [Prey - {[x=1]x}]
50      : /*  colour-dependent rates*/
51       [x=1] MassAction(kc1)
52      ++ [x=2] MassAction(kc2)
53      ++ [x=3] MassAction(kc3)
54      ;
```

```
55   reproduce_prey
56  {[x=1]} /* explicit transition guard */
57     :
58     : [Prey + {Births`x}] & [Prey - {x}]
59     : MassAction(kr)
60     ;
61   pred_death
62     :
63     : [Predator - {x}]
64     : MassAction(kd)
65     ;
66
67 }
```

When the given model has many defined groups and many associated value sets, it is important to determine efficiently the constant values for obtaining the desired results for unfolding, animating, simulating or even exporting the coloured model on hand. Algorithm 4.1 sketches the procedure's steps assigning the values of the corresponding selected value set to the constants. The procedure iterates over all defined groups and then it assigns for all constants belonging to the current group the corresponding value of the selected value set (lines 1- 6). Please note that the procedure assigns a value to a constant if the value set is not empty; otherwise, it assigns the default value set (Main value set, which has to be non-empty) to the constant (line 8).

Algorithm 4.1: Assigning constant values.

```
1:  procedure AssignConstantValues(v) /* v is the selected value sets*/
2:  for each defined group g do
3:    for each selected value set v  do
4:      for each constant c belonging to the group g  do
5:        if  the corresponding value of v denoted as v_0 is not empty then
6:          assign v_0 to the constant c;
7:        else
8:          assign Main value set to the constant c;
9:        end if
10:     end for
11:   end for
12: end for
13: end procedure
```

Animation and simulation of scaleable models which are scaled by one or more scaling factors (constants belonging to the *coloring* group) require re-unfolding the model on hand; when changing the latest selected value set of the *coloring* group. Algorithm 4.2 sketches the procedure's steps for assigning the constant values after changing the latest selected value sets when simulating or animating the given model. The procedure first calls the procedure *AssignConstantValues* (line 3). The model will be unfolded if the value set of the *coloring* group has changed (line 4). Finally, we animate or simulate the model (line 6).

Algorithm 4.2: Simulate/Animate a scaleable model when changing the selection of the groups' value sets.

```
1: procedure UpdateConstantValues(ṽ) /* ṽ is the newly selected value sets*/
2: AssignConstantValues(ṽ);
3: if the value set of the coloring group has changed then
4:    re-unfold the model;
5: end if
6: do animation/Simulation;
7: end procedure
```

4.2.2 Colour Expressions

Colour expressions are used as annotations in different positions of the given coloured Petri net model including places, transitions and arcs. Table 4.3 presents some examples of using colour expressions in the different positions of a coloured Petri net model. Due to some previous inconsistencies (related to the computation of the places' initial marking) in Snoopy, Snoopy now computes the initial marking by means of the *dssd_util* library [dss] which makes use of the IDD for this purpose. For more details about the allowed expressions, please check our CANDL report [ACR+21]. Furthermore, some parts of the colour expression syntax checker in Snoopy previously required manual parsing which is a time consuming and non-safe way for checking larger models. Thus, Snoopy now uses the the *dssd_util* syntax checker.

Another inconsistency between Snoopy and *CANDL* lay in the usage of some colour expression operators. Table 4.4 presents these operators and their usage together with some explanatory examples. Now both operators in each line can be used interchangeably.

Beyond that, Snoopy supports now new features/functionalities by extending the colour expression parser whose grammar and its associated rules were introduced in [LHR12a]. These features are crucial for dealing with some problems. In the following we present these new features together with examples demonstrating them. See Appendix A for the updated grammar of Snoopy's colour expressions parser.

Table 4.3: The usage of colour expressions in $\mathcal{PN}^{\mathcal{C}}$.

Position	Usage	Examples
Places	initialise all place instances with one token	1‘all()
	initialise a certain place instance with one token	$[x = N \& y = M]1‘(x, y)$
	initialise a set of place instances with one token	$1‘(x > 1)$
Transitions	guards	x=1
	colour-dependent rates	$[x = 1]MassAction(k1)++[x > 1]MassAction(k2)$
Arcs	arc inscriptions	2‘x++3‘y

Table 4.4: Harmonised boolean operators.

Operator	**Alternative**	**Description**	**Example**
&	&&	logical AND operators	$x = 1$ & & $y = 3$
|	||	logical OR operators	$x = 1 \parallel x = 3$
=	==	equal operators	$x == 1$
<>	! =	unequal operators	$x! = 1$

4.2.3 The elemOf Operation

The *elemOf* operation belongs to the boolean operations; it takes two operands and returns either *true* or *false*. It checks a certain colour whether it belongs to a certain colour set or not. It returns *true* if the colour (left-hand operand) belongs to the colour set (right-hand operand); otherwise, it returns *false*. Table 4.5 illustrates the usage of the *elemOf* operation.

Table 4.5: Examples of the *elemOf* operation.

example	**colour set**	**returned boolean value**
2 elemOf CS1	CS1 ={1..5}	true
8 elemOf CS2	CS2 ={1..5}	false
x elemOf CS1	CS1 ={1..5}	depends on the binding of the variable x
(x,y) elemOf CS3	CS3 =CS1 x CS2	depends on the bindings of the variables x and y

Assuming our running example (coloured food chain model), let us assume the first

prey gets consumed by its predator in a different rate than all other preys in the chain. Obviously, this problem can be solved by means of colour-dependent rates, as we have seen in the first chapter. We could also make use of the *elemOf* operation to formulate the given problem. For this purpose, we are going to define a subset colour set called *CS_PREY* containing the colour of the first prey (by constraining the colour set CS to the colour 1). We then add a colour-dependent rate function to the transition *consume_sp*, i.e. we assign the kinetic parameter $kc1$ to the rate function if the instance corresponds to the colour set *CS_PREY* (colour 1 corresponds to the first transition instance, e.g. *consume_sp_1*); otherwise, we assign the kinetic parameter $kc2$ to the rate functions of all other transition instances. Figure 4.1 gives the coloured stochastic Petri net model for the problem on hand.

Figure 4.2 gives related stochastic simulation traces of the place instances *Prey_1* and *Predator_1*. We notice that the *Predator_1* consumes the *Prey_1* and sooner or later the *Predator_1* becomes extinct over time (sub-system 1). For another case study utilising the operator *elemOf*, see Epidemic/Pandemic modelling [CGH21].

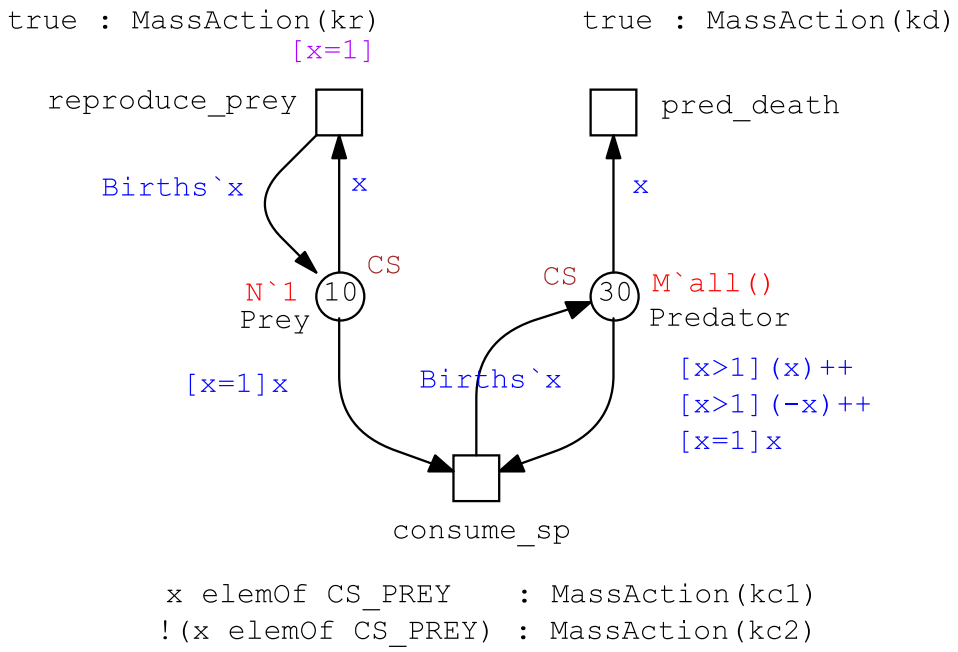
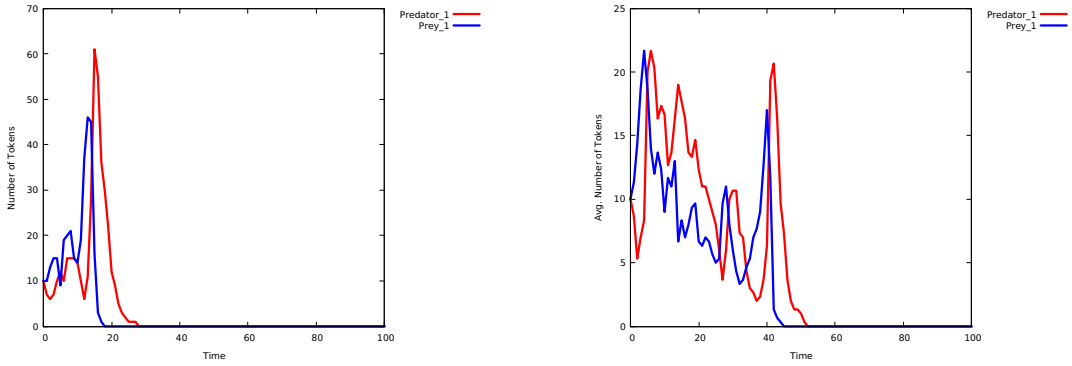


Figure 4.1: Using *elemOf* in colour-dependent rates. For the colour definitions and the used kinetic parameters, see Table 4.6.



(a) Single stochastic run.

(b) Averaged over 3 stochastic runs.

Figure 4.2: Stochastic simulation traces of the food chain model shown in Figure 4.1. For the constant values, see Table 4.6.

Table 4.6: Declarations for the model given in Figure 4.1.

Type	Declaration
Constant	*int* $SIZE = 3$; // an integer constant used as scaling factor.
Constant	*int* $N = 10$; // an integer constant used for initialising the places' marking
Constant	*int* $M =$ N;
Constant	*double* $kc1 = 0.03$; // kinetic parameter.
Constant	*double* $kc2 = 0.01$; // kinetic parameter.
Constant	*double* $kr = 0.1$; // kinetic parameter.
Constant	*double* $kd = 0.4$; // kinetic parameter.
Colorset	CS = *int* with 1 - $SIZE$; // space of preys/predators
Colorset	$CS\_PREY =$ CS with [x ==1]; // subset colour set with the first prey, determined by the boolean expression $x == 1$.
Variable	x :CS;

4.2.4 The Set Difference Operation

The set difference operation is applied in multi-set expressions to subtract a certain colour or a set of colours from a given colour set. The operator of this operation is detonated as (--) which performs the opposite operation of the multi-set addition operator (++). Let a colour set be S = {a, b, c}, Table 4.7 gives some examples of multi-set expressions with (--) operator over the colour set S.

Table 4.7: Examples of the multi-set difference operation.

Multi-set expression	**Result**
2‘a -- 1‘a	1 occurrence of a
2‘a --a	1 occurrence of a
1‘a ++ 2‘b -- 1‘b	1 occurrence of a and 1 occurrence of b
all() -- 1‘c	1 occurrence of a and 1 occurrence of b
all() -- (1‘a++1‘b)	1 occurrence of c

In the following, we present one case study (mutual exclusion problem), for which it is crucial to use the set difference operation. The mutual exclusion problem is one of the most well-known problems in concurrent systems, in which many processes compete to obtain access to the critical section. The critical section has to be protected from the concurrent access to a shared resource for writing purposes, i.e. keeping that shared resource consistent, when several processes are trying to change it. Figure 4.3 gives the scaleable coloured Petri model for the problem on hand. This model is scaled by the number of processes. For the colour definitions, please compare Table 4.8.

Table 4.8: Declarations for the model given in Figure 4.3.

Type	Declaration
Constant	*int* $N = 3$; // an integer constant used as scaling factor representing the number of processes
Colorset	Process = *int* with 1 - N; // integer colour set
Colorset	Bool = *bool* with {true, false}; // boolean colour set
Colorset	Flag = *product* with Process x Bool; // product colour set
Variable	x : Process;
Variable	y : Bool;

Each process has four states: *Idle*, *Waiting*, *AboutToEnter* and *Mutex*, each represented as a coloured place (defined on the colour set *Process*) with the place *Idle* initialised with one token of every colour. In order to recognise the process which already asked to get accessed to the critical section, we need to associate a boolean flag with each process. Thus we add a new place *Flag* (defined on the colour set *Flag*)

being initialised with *false* for all processes. Moreover, to avoid the situation that two waiting processes get access to the critical section at the same time, we need to check that the flags of all the other remaining processes are false. This can be achieved by using the colour expression (all()--x,y) and assigning the guard $y == false$ to the transition *wait*. We then set the flag (to be *true*) of the process which successfully passes the transition *wait*, which means that this process can safely enter the critical section. Finally, we unset the flag of the process as soon as it leaves the critical section.

In the following we are going to illustrate the model behaviour by playing the token game (model animation). In our model, the number of competing processes is $N = 3$, and the transition *askingForAccess* is enabled under the given initial marking; thus the variable x has the following three bindings; $x = 1$; $x = 2$; and $x = 3$; let us fire the transition *askingForAccess* by choosing the binding $x = 2$; this will remove one token of the colour 2 from the place *Idle* and add that coloured token to the place *Waiting*. Consequently, the transition *wait* is enabled; because there is one coloured token in its pre-place and the flags of the processes 1 and 3 are both *false*; thus, the process 2 can proceed by firing the transition *wait*. After that, the transition *setFlag* can obviously fire as it is enabled under the current marking, so firing this transition leads the process 2 to enter the critical section and to set the flag of the process 1 to *true* by *(2, true)*.

Now, let us assume that the process 1 (binding $x = 1$) would try to take its turn; so firing the transition *askingForAcces* will remove one token of the colour 1 from the place *Idle* and add it to the place *Waiting*. The transition *wait* is not enabled as the flag of the process 2 is *true*; which does not fulfil the associated guard. Assuming the process 2 finishes its work in the critical section, then the transition *resetFlag* can fire; thus firing this transition leads to reset the flag of the process 2 by *(2, false)*. At this moment, the process 1 can enter the critical section as the transition *wait* gets enabled; the flags of the other remaining processes were reset to *false*.

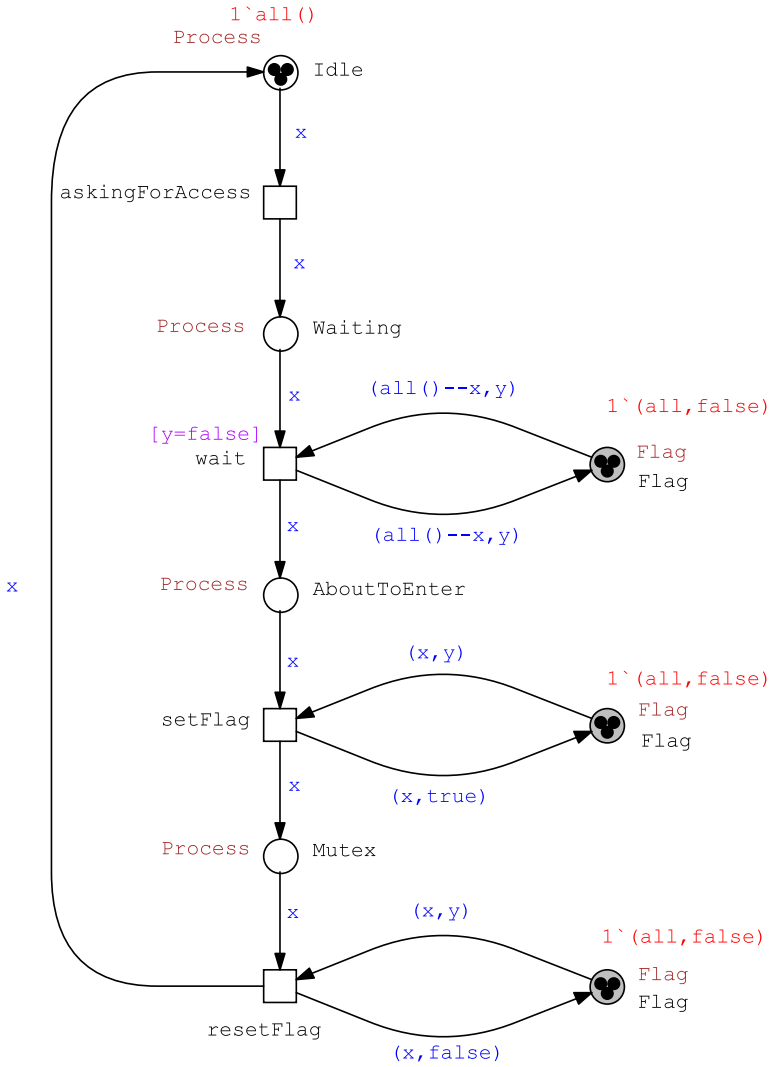


Figure 4.3: One possible solution of the mutual exclusion problem - many processes trying to get concurrently access to the critical section. See Table 4.8 for the colour definitions.

4.2.5 Colour Functions

Colour functions can be any arbitrary colour expressions which can be annotated in different positions in the coloured model making the coloured Petri net much more tidy and easy to read. They can be used:

- To determine the initial markings of places,
- As transition's guard,
- As arc expression.

Each colour function has a return type (colour set) determining the type of the returned value, parameters together with their colour types and an arbitrary colour expression as a function body. Snoopy now supports nested function calls (inside the function body) which implicitly substitutes a function call by its body. Algorithm 4.3 sketches the function which is responsible for the function body substitution.

Algorithm 4.3: Substitution of the function body.

```
1: function SubstituteFunctionBody(functionBody)
2: if there is no function call in functionBody then
3:    return functionBody;
4: end if
5: substitutedBody ← empty string; /* initialisation step */
6: for each function call f in functionBody do
7:    substitutedBody ← substitute f by its body; /* first level substitution*/
8: end for
9: substitutedBody ← SubstituteFunctionBody(substitutedBody); /* recursive call
   for more nested function calls if they do exist*/
10: return substitutedBody;
```

Furthermore, we now support colour functions to be used in a tuple expression (which was not possible), so that the returned colour value represents an element in the tuple expression. This includes the default colour function *all()* which returns all colours of the given colour set, see the mutual exclusion problem for an example (Figure 4.3). We are going to illustrate this feature using the GCD (Greatest Common Divisor) problem.

We define the function g which is formally given in Equation 4.1 [Rüd93]:

$$g(x,y) = \begin{cases} x & \text{if } x \leq y \\ y & \text{if } x > y \ \& \ x \ \% \ y = 0 \\ \text{x\% y} & \text{otherwise} \end{cases} \tag{4.1}$$

In the following, we present three scenarios for modelling the GCD problem by distributing the computations over many processes. The idea of distributing the GCD computation was initially introduced in [Mat89], whereas the equivalent $\mathcal{PN}^{\mathcal{C}}$ model was presented in [Rüd93].

First scenario Let us assume that we need to compute cooperatively the GCD for three given integer numbers a, b and c, by distributing the computation on three different processes. In this paradigm, the first process computes concurrently the function g (see Equation 4.1) on the numbers a and b, i.e. g(a,b), the second process first computes concurrently the function g on the numbers a and b, then it computes the function g for the output of its first computation and the third number c i.e. g(g(a,b),c). Finally, the third process computes concurrently the function g on the first and third numbers, i.e. g(a,c). We iterate all these steps until we obtain the GCD value. See Figure 4.4.

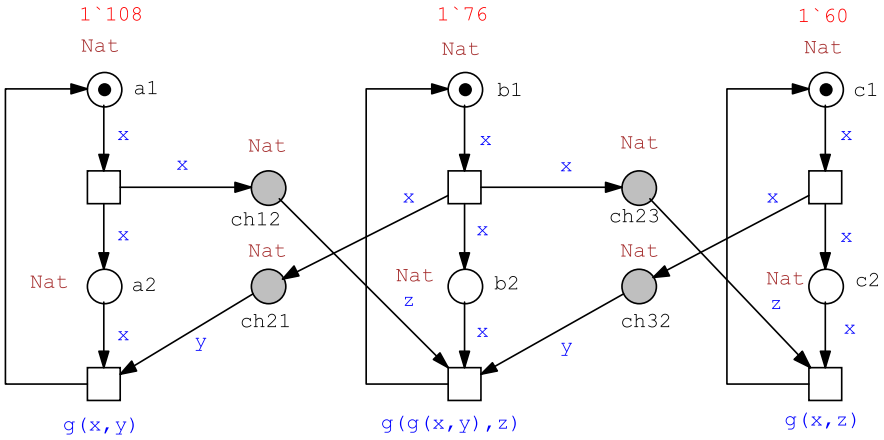


Figure 4.4: Basic coloured model of the GCD problem. We model the space of the integer numbers by using the integer colour set *Nat*. Then we initialise each process with one coloured token representing a, b and c. The three processes communicate with each other by means of the logical places, their names start with the prefix *ch* (channel). Thus these places work as communication channels between neighbouring processes. We then use the colour function g to compute the GCD formula. See Table 4.9, for the used colour definitions.

Second scenario In this scenario, processes are arranged in a cycle, each process computes the function $g(x, y)$, where x is the integer number which is assigned to the current process and y is the integer number which is delivered from the single neighbouring process. Thus, the first process computes $g(a, c)$, the second process

computes $g(b, a)$ and the third process computes $g(c, b)$. This communication paradigm gives a better scaleable communication style than the one which has been presented in the first scenario by exploiting a uniform neighbourhood relation between processes. See Figure 4.5.

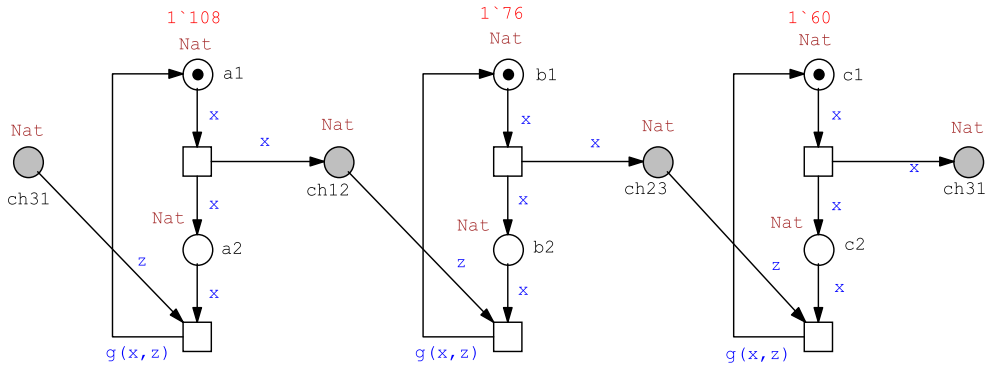


Figure 4.5: Coloured model of the GCD problem (second scenario). The cyclic computation pipe line exploits the uniform neighbourhood relation between two consecutive processes. Here we assume, the process 3 is the neighbour of the process 1. See Table 4.9, for the used colour definitions.

The two coloured models presented in the former scenarios are not scaleable in terms of the number of communicating processes. This means that we need to manually add Petri net blocks (representing processes), as many as the number of the integer numbers we would like to involve in the computation, and we have to add new places representing the channels between them. Figure 4.6 presents the scalable version of the GCD model.

4.2.6 Observers

Observers are mathematical expressions, which may involve some model variables (places) or transitions in order to observe them as the model evolves over time. As observers were already supported in quantitative Petri nets (uncoloured $\mathcal{PN}$), we now support them in coloured quantitative Petri nets in order to observe coloured places/transitions, place/transition instances or a combination of both (either coloured or uncoloured places/transitions). Each observer has a unique name, a type which determines the observer type and a mathematical function as observer body. It is worth mentioning that the observer body may involve constants (integer or double type). Table 4.10 lists the allowed observer types and a brief description for each one.

Algorithm 4.4 sketches the required steps for computing observers. The algorithm takes as input the following: simulation traces of places and transitions (coloured and

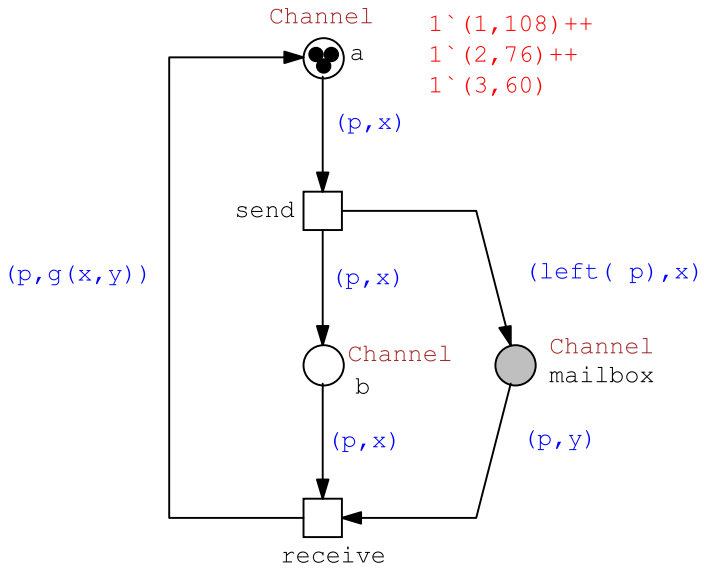


Figure 4.6: Scaleable coloured Petri net model for the GCD problem. We encode the integer numbers and the processes using integer colour sets, e.g. *Nat* and *process*, respectively. In order to associate each process with an integer number, we need to define the product colour set *Channel*, by applying the Cartesian product on the colour sets *Process* and *Nat*. The place *a* is initialised with one token of the colour *(1,108)*, one token of the colour *(2,76)* and one token of the colour *(3,60)*. These tokens represent the processes together with the integer numbers which are initially assigned to them. The place *mailbox* represents the channels, which will take care of the communication between neighbouring processes. So, we need to annotate the pre-arc of the place *mailbox* with the colour expression *(left(p),x)*, where the function left will return the left neighbour of the process *p*. Thus, the tuple expression *(left(p),x)* represents sending the current number to the left neighbour. See Table 4.9 for more details.

unfolded ones) and the defined observers, while it gives observer traces as output. We first initialise the output traces of the observers. We then iterate over all defined observers (line 2), and for each time point of the simulation time we evaluate the observers' body using the simulation traces of the involved places/transitions, and finally we assign the evaluated value to the corresponding output trace (line 4).

Table 4.11 gives some observer definitions for the coloured food chain shown in Figure 2.14. Figures 4.7 and 4.8 present the corresponding observer traces. In order to show the difference between simulation traces of some variables/transitions and their observer counterparts we give both simulation traces and observer traces side by side.

Table 4.9: Declarations for all versions of the GCD model. Please note that this table combines the colour definitions for the three GCD model versions; thus some definitions in this table are not used by all models.

Type	Declaration
Constant	*int* $N = 3$; // an integer constant used as scaling factor representing the number of processes.
Constant	*int* $MAX\_INT = 200$;
Colorset	Process = *int* with 1 - N; // integer colour set
Colorset	Nat = *int* with 1 - MAX_INT; // integer colour set
Colorset	Channel = *product* with Process x Nat; // product colour set.
Variable	x, y, z : Nat;
Variable	p : Process;
ColorFunction	*bool* g(y Nat, z Nat) { [y<=z] y ++ [y>z & y% z=0] z ++ [y>z & y%z <>0] y%z }
ColorFunction	*Process* left (q Process) { [q=1] N ++ [q<>1] q-1 };

Algorithm 4.4: Computation of observers.

Input: Simulation traces and the defined observers.
Output: Output traces of observers.

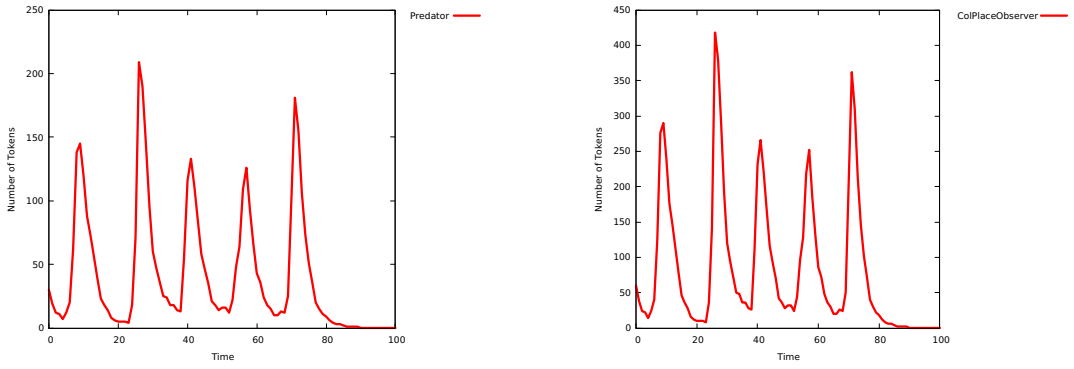
```
1: initialise output trace output;
2: for each defined observer do
3:    for each time point of the simulation time do
4:       output ← evaluate observer body using simulation traces of the involved
         variables/transitions;
5:    end for
6: end for
```

Table 4.10: Observers in coloured quantitative Petri nets.

Type	what for
Place	observes one or more coloured places
Transition	observes one or more coloured transitions
Place instance	observes one or more place instances
Transition instance	observes one or more transition instances
Mixed	observes a combination of any kind of observers including pre-defined mixed observers

Table 4.11: Some observer examples - coloured food chain model given in Figure 2.14. Please note that the constant ko is an arbitrary constant used in the observer body expression.

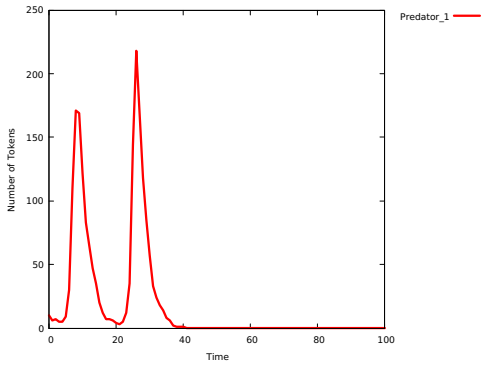
Observer ID	Type	Body	Figure
ColPlaceObs	Place	$Predator * ko$	4.7b
InstPlaceObs	Place instance	$(Predator\_1 + 1) * ko$	4.8b
InstTransObs	Transition instance	$pred\_death\_1 + pred\_death\_2 + pred\_death\_3$	4.8d
MixedObs	Mixed	$InstPlaceObs + ColPlaceObs$	4.8e
MixedTransObs	Mixed	$pred\_death_1 + pred\_death$	4.8f



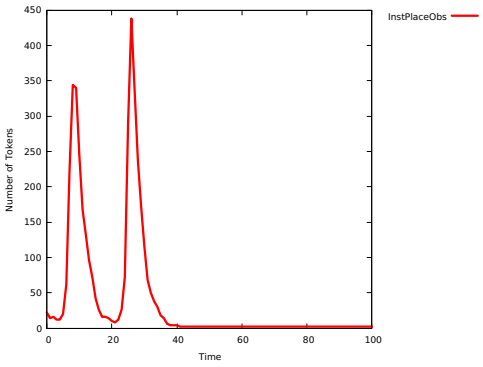
(a) Simulation trace.

(b) Observer trace, with $ko = 2$.

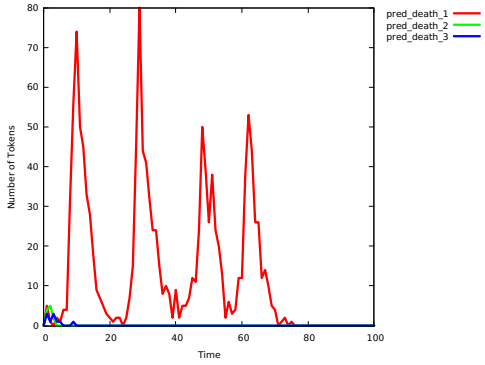
Figure 4.7: Simulation trace of the coloured place *Predator* and its corresponding coloured place observer (*ColPlaceObserver*).



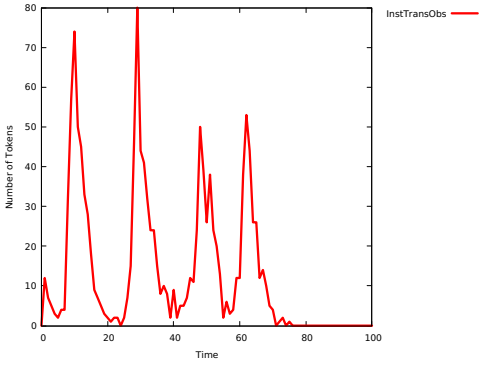
(a) Simulation trace of the instance place *Predator_1*.



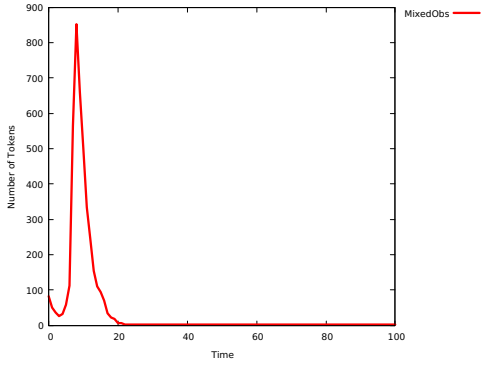
(b) Trace of the *InstancePlaceObserver*, with $ko = 2$.



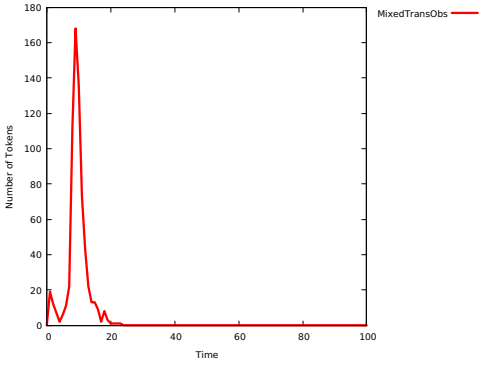
(c) Simulation traces of transition instances for the transition *Predator*.



(d) Trace of the *InstanceTransObserver*.



(e) Trace of the *MixedObserver*.



(f) Trace of the *MixedTransitionObserver*.

Figure 4.8: Some observer traces of the food chain model as they are defined in Table 4.11. The kinetic parameters kr=0.5, kc=0.01 and kd=0.4 are getting assigned to the rate functions of the transitions *reproduce_prey*, *consume_sp* and pred_death, respectively.

4.3 Declaration Dependencies

As we have seen, Petri nets basically comprise different types of structural elements which can involve a set of user-defined declarations. These declarations are used in different positions of the model on hand. For example, constants can be used as kinetic parameters for the transitions' rates in standard quantitative Petri nets and they can also be used to develop scaleable models in (quantitative) coloured Petri nets.

The diagram sketched in Figure 4.9 depicts the Petri net declarations for both standard and coloured Petri nets as they are supported by our framework. We obviously notice that coloured Petri nets outperform uncoloured Petri nets by their colour-related definitions which are colour sets, variables and colour functions.

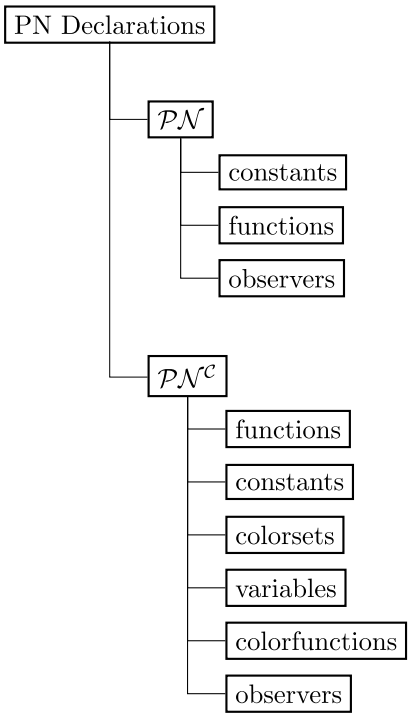


Figure 4.9: Petri net declarations diagram.

In our framework, one declaration can be used by other declarations depending on its kind, e.g. constants can be used by observers, but not vice versa. Thus, there exist dependencies among user-defined declarations. We can make use of these dependencies as we will see later to, e.g. avoid removing one used declaration by other declarations or by the structural elements of the model on hand, when it contains a large number

of declarations, and it is difficult to recognise which of these declarations is in use.

Dependency graphs are directed graphs representing causal dependencies between a set of elements in the form of a directed graph [FRS19, ELMS21]. They have many applications in, e.g. compilers and text extraction. In the following, we present the relation among user-defined declarations.

4.3.1 Declaration Dependencies in Uncoloured Petri Nets

Uncoloured Petri nets have three kinds of declarations: constants, functions and observers. Figure 4.10 presents the possible dependencies among these declarations.

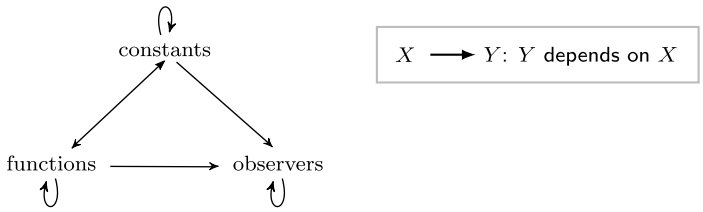


Figure 4.10: Dependency graph of user-defined declarations in uncoloured Petri nets. Please note that a function is a shortcut for a mathematical expression that a constant may depend on, i.e. the value of a constant is defined by a mathematical expression. Constants can take mathematical expressions directly as values.

Each of these declarations except observers can be used in the $\mathcal{PN}$ model for the following purposes:

- to initialise the places' marking,
- as arc weight, and
- as kinetic parameter of a transition's rate.

Algorithm 4.5 sketches the function *ComputeConstantDependencyTree* which computes the dependency graph of a given constant. Firstly, the algorithm assigns the constant identifier to the root as key. Then, for each function depending on this constant, it adds the dependency graph of this function as child, by calling the procedure *AddChild*, which takes the root node of the given constant and the root node of the function's dependency graph that will be obtained by calling the function *ComputeFunctionDependencyTree* (lines 3-5). After that, we repeat the same steps for observers (lines 6-8). Finally, we return the root node of the dependency graph, if the constant is not used by the other remaining constants (lines 9-11). Otherwise, we add the dependency graph for each constant depending on the given constant by recursively calling the function *ComputeConstantDependencyTree* (lines 12-17).

Algorithm 4.6 presents the function *ComputeFunctionDependencyTree*, which computes the dependency graph of an input function. Firstly, we assign the function identifier to the root node as key. We add the dependency graph of each constant if it depends on the given function (lines 3-5). Then, for each observer depending on this function, we add the dependency graph of that observer as child by calling the procedure *AddChild*, which in turn calls the function *ComputeObserverDependencyTree* (lines 1-13). We finally return the root node of the dependency graph, if there is no other function depending on the given function (lines 6-8). Otherwise, we add the dependency graph for each function depending on the given function as child by calling the function *AddChild* which in turn recursively calls the function *ComputeFunctionDependencyTree (lines 14-19)*.

Algorithm 4.5: Compute the dependency graph of a constant.

```
1: function ComputeConstantDependencyTree(Id_const, root)
2: root ← Id_const;
3: for each declared function f depending on constant Id_const do
4:    newFunChild = CreateNode(); /* creates a new node */
5:    AddChild(root, ComputeFunctionDependencyTree(f, newFunChild));
6: end for
7: for each declared observer ob depending on constant Id_const do
8:    newOBSChild = CreateNode(); /* creates a new node */
9:    AddChild(root, ComputeObserverDependencyTree(ob, newOBSChild));
10: end for
11: if there is no constant uses constant Id_const then
12:    return root;
13: end if
14: for each declared constant const except Id_const do
15:    if the constant const uses constant Id_const then
16:       newConstChild = CreateNode(); /* creates a new node */
17:       AddChild(root, ComputeConstantDependencyTree(const,newConstChild));
18:    end if
19: end for
20: return root;
21: end function;
```

Algorithm 4.7 sketches the function *ComputeObserverDependencyTree*, which computes the dependency graph for an observer. We first assign the observer identifier to the root node as key (line 2). We then return the root node, if the given observer is not used by the other remaining observers; otherwise, we add the dependency graph for each observer using the given observer as child, by calling the procedure *AddChild*,

Algorithm 4.6: Compute the dependency graph of a function.

```
1: function ComputeFunctionDependencyTree(Id_fun, root)
2: root ← Id_fun;
3: for each declared constant const depending on function Id_fun do
4:    newConstChild = CreateNode(); /* creates a new node */
5:    AddChild(root, ComputeConstantDependencyTree(const, newConstChild));
6: end for
7: for each declared observer ob depending on function Id_fun do
8:    newObsChild = CreateNode(); /* creates a new node */
9:    AddChild(root, ComputeObserverDependencyTree(ob, newObsChild));
10: end for
11: if there is no function uses Id_fun then
12:    return root;
13: end if
14: for each declared function f except Id_fun do
15:    if the function f uses function Id_fun then
16:       newFunChild = CreateNode(); /* creates a new node */
17:       AddChild(root, ComputeFunctionDependencyTree(f, newFunChild));
18:    end if
19: end for
20: return root;
21: end function;
```

which recursively calls the function *ComputeObserverDependencyTree* using a newly created node (lines 6-11). Please note that this algorithm does not iterate over constants and functions, as there are no forward dependencies form constants/functions to observers.

Algorithm 4.7: Compute the dependency graph of an observer.

```
1: function ComputeObserverDependencyTree(Id_obs, root)
2: root ← Id_obs;
3: if there is no observer uses observer Id_obs then
4:    return root;
5: end if
6: for each declared observer ob except Id_obs do
7:    if the observer ob uses observer Id_obs then
8:       newObsChild = CreateNode(); /* creates a new node */
9:       AddChild(root, ComputeObserverDependencyTree(ob, newObsChild));
10:   end if
11: end for
12: return root;
13: end function;
```

4.3.2 Declaration Dependencies in Coloured Petri Nets

Beside the declarations existing in uncoloured Petri nets, coloured Petri nets have colour-related declarations. The colour-related declarations include colour sets, variables and colour functions. Colour sets basically come in two flavours: simple and compound, where each of which has its own sub-types; compare Figure 4.11. One colour set may depend on pre-defined colour sets; which is a common case for, e.g. subset colour sets and product colour sets.

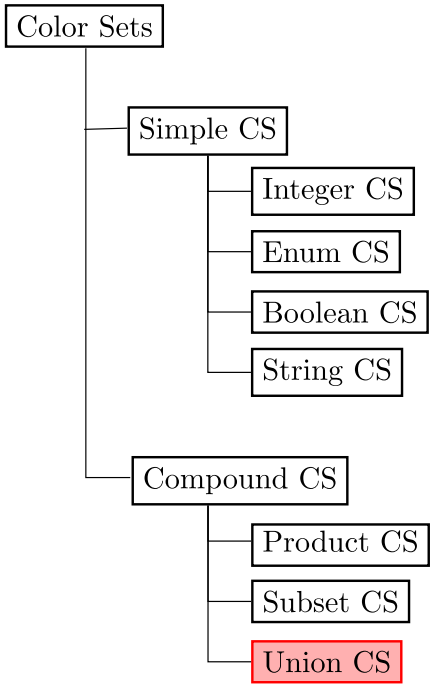


Figure 4.11: Colour sets diagram; illustrating the sub-types of simple and compound colour sets. Union colour set is marked with red to indicate that it is deprecated.

Figure 4.12 presents the dependency graph of declarations in coloured Petri nets. This graph extends the dependency graph of uncoloured Petri nets (shown in Figure 4.10) by colour sets, variables and colour functions.

Figure 4.13 gives the dependency graph of the constant *SIZE* for the coloured model shown in Figure 4.1.

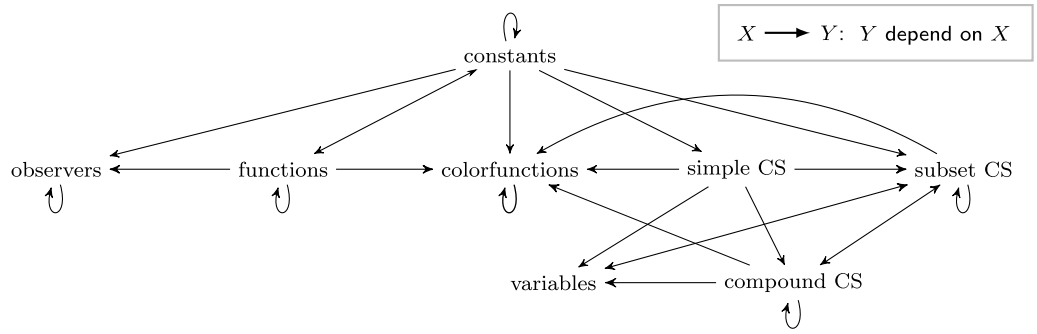


Figure 4.12: Dependency graph of user-defined declarations in coloured Petri nets.

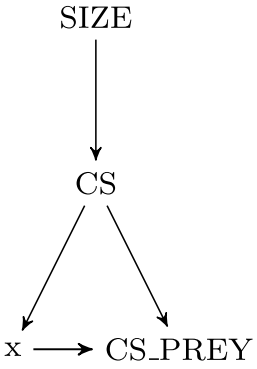


Figure 4.13: Dependency graph for the constant *SIZE*, compare Table 4.6 for the colour-related definitions of the model given in Figure 4.1.

Computing dependency graphs of constants in coloured Petri nets can basically be performed in the same way as sketched in Algorithm 4.5; but we need to iterate over colour sets and colour functions. Then we add the dependency graphs of each colour set and colour function as children, if they use the given constant. Algorithm 4.8 sketches the function *ComputeColorsetDependencyTree*, which computes the dependency graph of the given colour set (line 2). We first assign the colour set identifier to the root node as key. Then, we iterate over variables and colour functions as they directly depend on colour sets, and we add their dependency graphs as children if they use the given colour set (lines 3-13). We finally add the dependency tree of each colour set (except the given one) as child, if the latter use the given colour set (18-23); otherwise, we return the root node of the dependency graph of the given colour set (15-17).

The function *ComputeVarDependencyTree* computes the dependency tree of a variable. It only checks subset colour sets, if they use the given variable to constrain a global colour set (see colour related definitions of the coloured model shown in Figure 4.1). Then we add their dependency graphs as children to the root node of the

given variable. Similarly, the function *ComputeColourFunDependencyTree* computes the dependency tree of a colour function which checks the other remaining colour functions, if they use the given colour function. If so, we add their dependency graphs as children to the root node of that colour function.

Algorithm 4.8: Compute the dependency graph of a colour set.

```
1: function ComputeColorsetDependencyTree(Id_cs, root)
2: root ← Id_cs;
3: for each declared variable var do
4:    if the variable var uses colour set Id_cs then
5:       newVarChild = CreateNode(); /* creates a new node */
6:       AddChild(root, ComputeVarDependencyTree(var, newVarChild));
7:    end if
8: end for
9: for each declared colour function colfun do
10:   if the colour function colfun uses colour set Id_cs then
11:      newFunChild = CreateNode(); /* creates a new node */
12:      AddChild(root, ComputeColourFunDependencyTree(colfun,
         newFunChild));
13:   end if
14: end for
15: if there is no colour set uses the colour set Id_cs then
16:   return root;
17: end if
18: for each declared colour set except Id_cs do
19:   if the colour set cs uses colour set Id_cs then
20:      newCsChild = CreateNode(); /* creates a new node */
21:      AddChild(root, ComputeColorsetDependencyTree(cs, newCsChild));
22:   end if
23: end for
24: return root;
25: end function;
```

4.3.3 Applications of Declaration Dependencies

In the following, we present two applications for making use of the declaration dependencies.

Cleaning Unused Declarations

Users may define many declarations of the same or different types at early stages of the model design without using them in the final model. This would cause confusion for modellers, when they would further develop the model in the future. Thus polishing the model by cleaning its unused declarations would be of help for this case. Algorithm 4.9 sketches the boolean function *IsUsedDeclaration*. This function takes the root node of the dependency graph of a certain declaration, to take a decision about the usage of the declaration (in the model). This function returns true if one of the following two cases hold: the first case (2-4): the declaration is directly used in the model, e.g. by a place; the second case: there exists at least one other declaration (in the dependency graph) using it in an indirect way (lines 5-9). Otherwise, it will return false (line 10). For example, the dependency graph shown in Figure 4.13, the constant *SIZE* is used by the colour set *CS* which in turn is used directly in the model shown in Figure 4.1.

Algorithm 4.9: Determine whether a declaration is used or not in n $\mathcal{PN}$ model.

```
1:  function IsUsedDeclaration(root)
2:  if the key of the root is used in the model then
3:     return true; /* the declaration is directly used in the model */
4:  end if
5:  for each child node  do
6:     if  IsUsedDeclaration(child) then
7:        return true; /* stop here, it is enough to find one used declaration (child)
          to take a decision */
8:     end if
9:  end for
10: return false; /* the declaration is not used */
11: end function;
```

After finding unused declarations, a user can delete them, but it could happen that a modeller would like to keep some unused declarations for further use in the future. For this purpose, we support an option to individually select/unselect some of those declarations. Here we make use of the dependency graph to automatically find those dependencies which have to be kept. Algorithm 4.10 sketches the procedure *SelectDeclarationDependencies* which is responsible for selecting all dependencies of a certain declaration.

Algorithm 4.10: Automatically select the dependencies of a declaration.

```
1: procedure SelectDeclarationDependencies(key)
2: for each declaration dependency graph do
3:   if the key do exist as child then
4:     select the key of the root node as related dependency;
5:   end if
6: end for
7: end procedure;
```

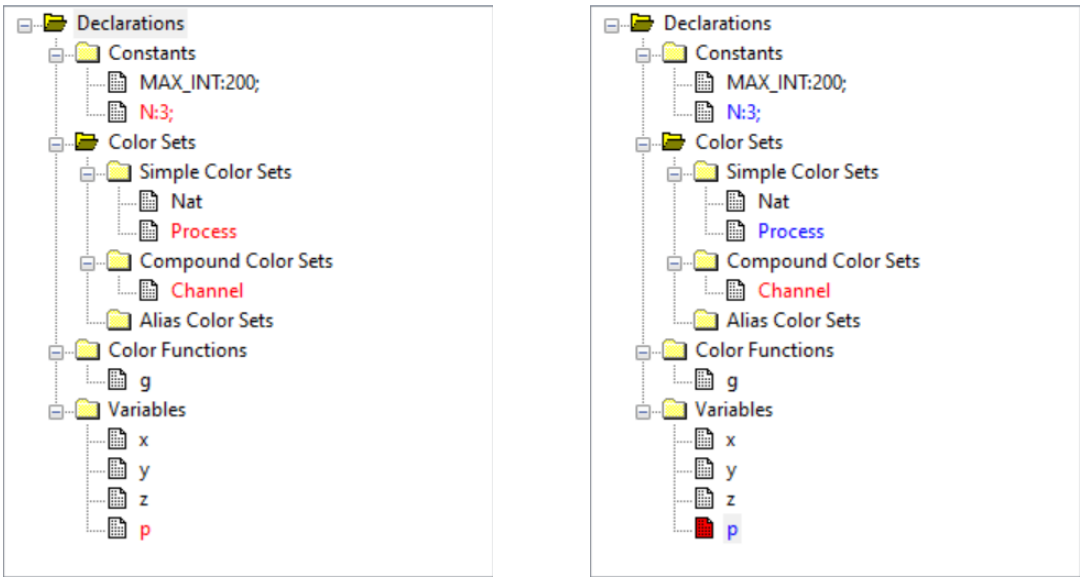
Assuming the coloured model shown in Figure 4.4, which contains all the declarations sketched in Table 4.9. In this model, as we see, there is no need to declare the following declarations (they are not in use): constant: N; Colorsets: Processes and Channel; variable: p. Figure 4.14a gives the result of performing Algorithm 4.9 on Snoopy's declaration graph, whereas Figure 4.14b presents the result of performing Algorithm 4.10 on the dependency graph for the given node *p*.

Selective Import ANDL/CANDL

Snoopy is able to read (coloured) Petri nets from ANDL/CANDL (text format) files, and to translate the model into its graphical representation. This will import the entire model including places, transitions, arcs and all defined declarations. For some modellers who frequently develop models relating to their major, e.g. biological models, they probably need to define the same declarations for each new model, which suggests to extend their modelling context. Thus, it would be helpful to have the most used declarations in a separate ANDL/CANDL file, and then what needs to be done is to import (from the declarations file in ANDL/CANDL format) only those declarations which are frequently used. For this purpose, we added the *selective import* feature.

By using the *selective import* feature, one can individually choose some declarations to be imported to the model on hand, but these individual declaration may depend on other unselected declarations (dependencies). This would make the model invalid as it lacks some missing dependencies. Thus, we make use of the dependency graph of declarations to automatically select the related dependencies. Figure 4.15 presents some screenshots for using the *selective import* feature in Snoopy. We first import the CANDL file which contains all declarations sketched in Table 4.9. In Sub-figure 4.15a, all declaration are unselected, In Sub-figure 4.15b, we select the variable *p* to be imported to our model, the other remaining Sub-figures give the automatically selected declarations due to the existing dependencies. The automatically selected declarations are obtained by performing Algorithm 4.10 on the input declaration *variable: p*.

Another useful scenario for the declaration dependencies is to unselect one declaration (meaning to keep it in the model) and all the other declarations that use this



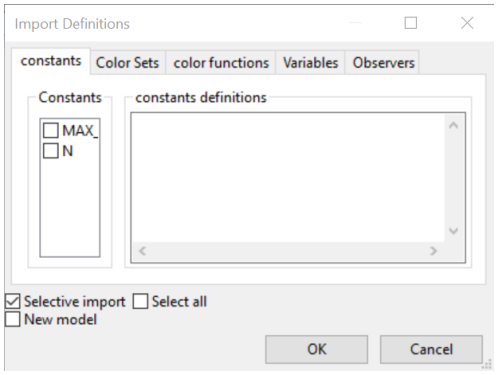
(a) Checking all declarations.

(b) Unselect the variable p.

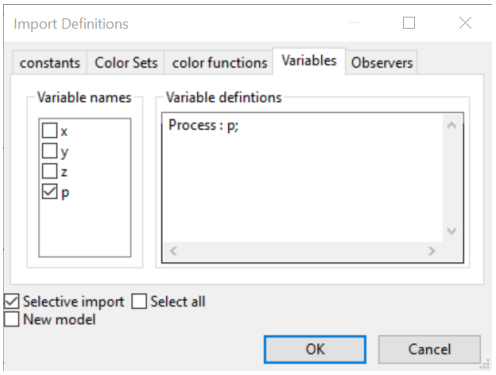
Figure 4.14: Checking unused declarations (coloured Petri nets) in Snoopy. Sub-figure (a) all unused declaration are marked in red. Sub-figure (b) Unselecting the variable p will automatically unselect (marked with blue) the colour set *Process* and the constant N due to the existing dependencies. Thus these declarations will be kept, while the colour set *Channel* will remain be selected to be deleted (marked with red), as there is no related dependency with the variable p.

declaration, because removing the former declaration and keeping those declarations which depend on it would make the model invalid. This calls for automatically unselecting all the declarations that use the unselected one. This can easily be achieved by iterating over the dependency graph of the unselected declaration and then marking all its child nodes as unselected. For example, unselecting the constant N will automatically unselect the colour sets *Process* and *Channel*, the colour function *left* and the variable p as all of them depend on the constant N which means all of them are children of the dependency graph of the constant N, see Figure 4.16.

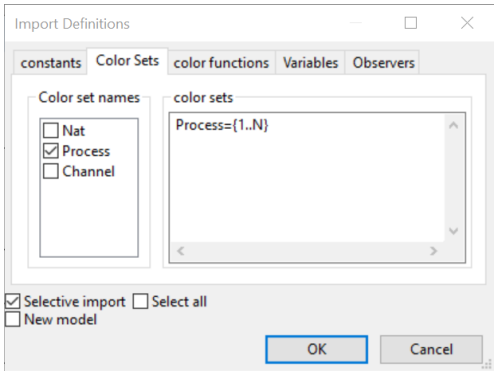
It is worth mentioning that the *selective import* feature requires an active Petri net document (an already opened $\mathcal{PN}$ document in the environment); otherwise, Snoopy will notify about this by means of a log message. Moreover, if some selected declarations to be imported do already exist in the target model, then Snoopy will offer an option to overwrite them.



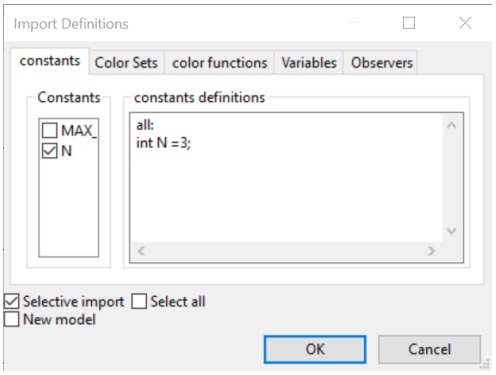
(a) All declarations are unselected.



(b) Select the variable p.

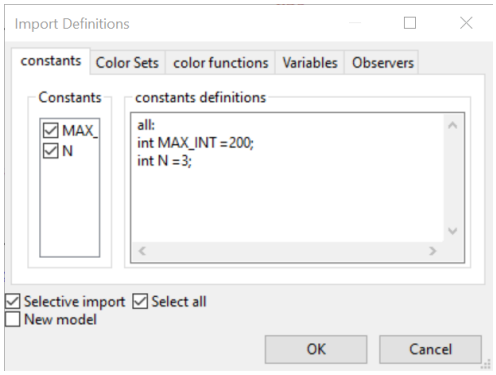


(c) The colour set *Process* has been automatically selected.

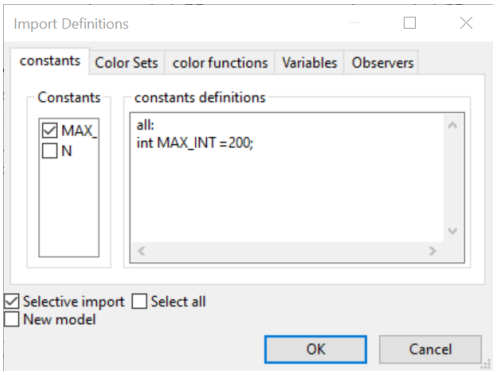


(d) The constant N has been automatically selected.

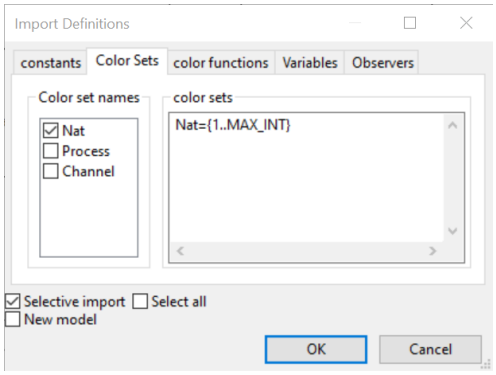
Figure 4.15: Selective import feature in Snoopy. Please note, the declarations in the *color functions* tab of the import definitions window will remain unselected. Declaration definitions are shown/disappear in the right-hand sub window as they are selected/unselected automatically (or manually by the user).



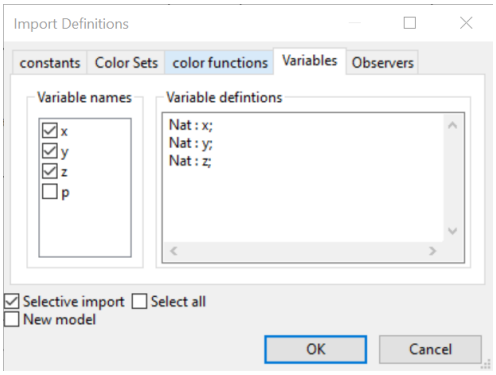
(a) All declarations are selected.



(b) Unselect the constant N.



(c) The colour sets *Process* and *Channels* have been automatically unselected.



(d) The variable p has been automatically unselected.

Figure 4.16: Selective import feature in Snoopy. All declarations were initially selected (by default). Please note, that the declarations in the *color functions* tab of the import definitions window will remain be selected.

4.4 Command-line Feature

Command-line tools may outperform those with a graphical user interface (GUI), in that the command-line interface (CLI) can be faster and more efficient than scrolling across GUI tabs and dialogues. This can be particularly useful when dealing with repetitive tasks. So we could write a simple script calling the tool (see Appendix A.3), and then let it run over all tasks automatically without any interaction with the tool itself. For this purpose, we now have an option to run Snoopy via the command-line prompt, with a few commands so far. Table 4.12 sketches the supported commands in Snoopy.

Table 4.12: Snoopy's commands - alphabetically ordered. The short-hand command has to be prefixed by -, whereas the alternative full command has to be prefixed by --.

Short command	Full command	Options	Description
c	close		close the tool
hw	help		show command line help
l	layout	1-3	do layout using one of these algorithms 1: *FMMM*; 2: *Planarization*; 3: *Sugiyama*
p	export	1	export the model to 1: *EPS*
s	save		save the model
v	version		show the tool's version

In the following, we present some command-line examples: the first example (line 1) prompts Snoopy to show the command-line help. The second example (line 2) prompts Snoopy to export the model *myPNModel.cpn* to *EPS*, with performing layout using the algorithm *Planarization*, saving the original model (after doing layout) and finally closing the tool. The third example (line 3) is equivalent to the example 2, but it uses the full form of the commands. The Example 4 (line 4) is also equivalent to example 2, but it combines all options together in one command.

Listing 2: Command-line examples. Please note that, the symbol $ refers to the command-line prompt.

```
1 $snoopy  -hw
2 $snoopy myPNModel.cpn -p 1 -l 2 -s -c
3 $snoopy myPNModel.cpn --export   1 --layout 2 --save --close
```

```
4 $snoopy myPNModel.cpn -p1l2sc
```

The implementation of this feature is basically done by the help of the *wxWidgets* library [LIBb], which provides a pre-defined parsing technique for adding a new command together with its options.

4.5 Closing Remarks

In this chapter, we presented some implementation aspects. The major work focuses on harmonising of coloured Petri nets in Snoopy with the CANDL format and uncoloured Petri nets as they are supported by Snoopy.

Firstly, grouping of the constants is now supported, which allows us to assign a set of constants to a specific group, for which many value sets can be defined. This has a big advantage for developing scalable models. Furthermore, Snoopy was previously using parameter nodes (elapses on canvas) to define kinetic parameters, this would make the model untidy and may have an influence on the size of the $\mathcal{PN}$ model file, as the graphics information has to be kept as well. Now, we can define constants in the same way as in uncoloured Petri nets. Secondly, due to some inconsistencies in Snoopy's implementation, some colour expressions were not working, especially for initialising the marking for coloured places, and even checking the syntax of colour expressions was done in a manual way (meaning no parsing technique was utilised). However, we now use the *dssd_util* to initialise the places' marking and to check the syntax of colour expressions. Thirdly, we extended Snoopy's parser (which is responsible for interpreting colour expressions and related colour operations) with new operations, i.e. *elemOf* and set difference operation; here we presented two (teaching) case studies for illustrating their usage. Fourthly, we harmonised some operators for colour expressions, e.g. = and ==.

As uncoloured Petri nets, coloured Petri nets now support observer definitions over coloured places/transitions, place/transition instances or even over a combination of all kinds. Finally, we introduced the concept of dependency graph and we showed how to make use of declaration dependencies for cleaning unused declarations and for importing declarations in a selective way.

Last but not least, we presented Snoopy's command-line feature together with the currently supported commands, and showed how to use this feature by means of some examples.

5 Conclusions and Outlook

In this chapter, we briefly recall our contributions, and then for each contribution, we sketch some possible extensions.

5.1 Conclusions

During this research work, we presented two different contributions to the interdisciplinary field of Petri nets and their applications on the field of systems biology. The two contributions are: the definition together with the implementation of new Petri net classes, fuzzy Petri nets, and the harmonisation of coloured Petri nets in Snoopy as they are supported by the CANDL format as well as uncoloured Petri nets. Of course, our approaches are not only useful for modelling and simulating biological systems, but also for other different systems. In the following sections, we summarise our contributions and some possible suggestions for future work.

5.1.1 Fuzzy Petri Nets

The modelling of biological systems is often hampered by parametric uncertainty, which usually comes from unavailable or imprecise parameters due to some environmental factors or lack of exact knowledge. When stochastic methods are not able to deal with such models, analysing them by giving uncertain band of all outputs of interest might be an alternative. Furthermore, more precise information about associated uncertainties can be obtained by producing timed membership functions for each output variable. We combine fuzzy logic with (coloured) quantitative Petri nets to address this issue yielding a new family of Petri nets called fuzzy Petri nets.

Fuzzy Petri nets capture the parametric uncertainties by representing each uncertain kinetic parameter as triangular fuzzy number. Uncertainty can be turned into fuzzy numbers in the following way: we first obtain a rough estimate for the interval of parameter values, and then extract the pessimistic value (a), the most possible value (b) and the optimistic value (c); all of these three values (a, b , c) define a triangular fuzzy number. For this purpose, a fuzzy simulation algorithm needs to discretise each fuzzy number into its crisp values, and then to perform the simulation on each sample. Thus, sampling strategies have to be utilised. The more efficient the sampling strategy is, the less redundant samples.

We presented the following sampling strategies: Basic sampling, Reduced sampling and *LHS* sampling. The basic sampling strategy discretises each level (of the fuzzy number) with the same number of samples, thus redundant samples may occur, which yields unnecessary simulation traces. The reduced sampling strategy tries to address the redundant samples issue by reusing those samples from the first level of all levels. For the first and second strategies, the number of samples depends on the number of the involved fuzzy kinetic parameters. The *LHS* sampling strategy is the most efficient strategy, as it always reduces the number of required samples, no matter how many fuzzy kinetic parameters are involved. The *LHS* sampling strategy produces a sampling matrix with N x K, where N is the number of desired samples and K is the number of fuzzy kinetic parameters. Here there is an overhead induced by the required computation to obtain the matrix which are performed by the *LHS* library.

In order to reduce the memory load, we produce for each fuzzy band the minimum and maximum traces over time. Finally, we presented some performance measurements for the fuzzy simulation algorithms.

5.1.2 Harmonising Coloured Petri Nets

Coloured Petri nets as they are supported in Snoopy suffered from some inconsistencies related to the usage of constants, colour expressions and observers, due to some former implementation issues. To use our tools. This means to use coloured Petri nets in a uniform way, for example, usage of colour expressions should be the same for, e.g. Snoopy and Spike. CANDL is a coloured Petri net format shared among all our tools, so we harmonised the usage of the definitions of coloured Petri nets as follows:

Snoopy now supports grouping of constants in the same way as they are supported in uncoloured Petri nets, which allows to assign many value sets for each group, and then users can run their experiments (for animation and simulation) on different constant values, without having to re-define the constants in use; what they need to do indeed is to change the value sets of the constant groups. The old way to define constants (to be used as kinetic parameters) required adding new graphic nodes, which makes the final model untidy and increases the size of the Petri net document. Now, constants can be defined by using the constant definitions window which is a more efficient and easy-to-use way.

Colour expressions are now used in the same way as in the CANDL format, which will be used to initialise the markings of places, arc expressions, guards or colour dependent-rates. This includes harmonising the logical (boolean) operations, e.g. & and &&.

We harmonised the usage of colour functions, so that one colour function can use nested function calls. Moreover, colour functions are now allowed to be used as elements inside tuple expressions. Moreover, we support new operations for colour expressions, these operations are: *elemOf* and --. All these new features/operations increase the

modelling power of coloured Petri nets. Please check Chapter 4 for some case studies.

Observers are mathematical functions over places and transitions. Now, we support the definition of observers to observe either coloured places, coloured transitions, place instances, transition instances or a combination of all these possibilities.

We then investigated the dependencies among user-defined declarations, and introduced a way depending on the dependency graphs to make use of such existing dependencies to detect and clean unused declarations, and to import ANDL/CANDL files (into Snoopy) in a selective way.

5.2 Outlook

In the following, we discuss some possible extensions for the contributions which have been presented in this thesis. Some of these extensions aim to get a better performance, while others aim to extend the current work with new features.

5.2.1 Extending Fuzzy Petri Nets

Extending fuzzy Petri nets presented in Chapter 3 includes both modelling and simulation aspects. In the following, we present some possible extensions:

Supporting other types of fuzzy numbers As we have seen, fuzzy kinetic parameters may be represented as triangular fuzzy numbers, where uncertainties are turned into triangular fuzzy numbers by means of three points. But it is also interesting to investigate the other types, as more accurate information about uncertainties could be obtained. We suggest to consider the following types:

- Trapezoidal fuzzy numbers: an uncertain kinetic parameter is represented using four points, which allows to give each point a different interpretation.
- Bell-shaped fuzzy numbers: like triangular fuzzy numbers, bell-shaped fuzzy numbers are represented by means of three points.
- Gaussian fuzzy numbers: each gaussian fuzzy number is determined by using two parameters, e.g. c and σ, which control the centre and the width of the shape of the gaussian fuzzy number, respectively.

Parallelising the fuzzy simulation algorithm For large-scale coloured fuzzy Petri net models, we noticed annoying simulation time, particularly when we deal with $\mathcal{FSPN}^{\mathcal{C}}$ and $\mathcal{FHPN}^{\mathcal{C}}$, whereby the number of stochastic runs (for each simulation) can be specified. This can be done more efficiently by, e.g. assigning the samples of each α-level to a separate core, which will then perform the simulation for that level.

Parallelising the computation of fuzzy bands and timed-membership functions After the fuzzy simulator finished, the computation of fuzzy bands and the

timed-membership functions is triggered, which causes annoying waiting time, especially when we have a large interval of the simulation time, as the membership functions of each output variable have to be computed. As each computation is independent, these computations can be parallelised by distributing them on several cores.

Extending ANDL/CANDL format by fuzzy Petri nets As we have seen in Chapter 3, exporting an $\mathcal{FPN}/\mathcal{FPN}^{\mathcal{C}}$ model to the ANDL/CANDL format will approximate the model on hand to its non-fuzzy counterpart, as ANDL and CANDL do not support fuzzy Petri nets so far, this extension includes two steps: extending the ANDL/CANDL grammar to accept the corresponding types of fuzzy Petri nets and extending the constant types by permitting constants to be defined as, e.g., triangular fuzzy numbers.

Investigating other kinds of uncertainties uncertainty of the model structure could be an example, as we have seen in the literature. Dealing with such kind of uncertainty requires implementing a new class of fuzzy Petri nets, called basic fuzzy Petri nets. Two perspectives have to be considered here: the ability to define a set of fuzzy rules and then applying a fuzzy reasoning approach using a fuzzy inference engine. For more details, see the literature, e.g. [V.R06, MNP11].

5.2.2 Extending Coloured Petri Nets

In the following, we propose some useful extensions for coloured Petri nets:

Harmonising The specification style for arc expressions and rate functions should be harmonized with that of marking expression. Marking expression widget (marking grid window for specifying the marking of places) basically consists of one column for specifying the colour and another one for specifying multiplicity. Each row of this window represents one part of the multi-set expression, where each part is separated by ++. Specifying both arc expression and colour-dependent rate have to be in the same style. Moreover, adding/removing/editing new value sets has to be allowed as well. The the appropriate value set can be chosen, when simulating/animating the given model.

Colour sets It should be possible to define new colour sets based on mathematical set operations on pre-defined colour sets (for integer colour sets), e.g. CS3 = CS1 - CS2; where each colour of the colour set CS3 is obtained by performing the subtract operation (on sets). Another extension is to support dynamic colour sets which are crucial for modelling and simulating dynamic membrane systems which may create or remove membranes while the system evolves over time [AHF22].

IDD-based Unfolding We still have some inconsistencies for the IDD unfolding algorithm, these consistencies are listed in our CANDL report, for more detailed information, please check [ACR+21]. For instance, unfolding models which make use of the operator -- is not working using the IDD unfolding engine.

Command-line feature The list of Snoopy's commands can be extended, so that

we can make use of the command-line feature to perform the same functionalities which already exist in Snoopy, such as model unfolding and simulation. Then, Snoopy will be able to perform, e.g. simulation for a set of $\mathcal{PN}$models by configuring an external script to call Snoopy in the same way as the automatic layout feature, see Section 4.4.

Bibliography

[ACR+21] Assaf, G; Chodack, J; Rohr, C; Schwarick, M ; Heiner, M: CANDL Report / Brandenburg University of Technology Cottbus, Department of Computer Science. 2021 (01-52). – Technical Report. not published yet

[AD98] Alla, H.; David, R.: Continuous and Hybrid Petri Nets. In: *Journal of Circuits, Systems and Computers* 08 (1998), Nr. 01, pp. 159–188

[AHF22] Assaf, G; Heiner, M ; FLiu: Coloured fuzzy Petri nets for modelling and analysing membrane systems. In: *Biosystems* (2022), pp. 104592. – ISSN 0303–2647

[AHL19] Assaf, G; Heiner, M ; Liu, F: Biochemical reaction networks with fuzzy kinetic parameters in Snoopy. In: Bortolussi, L (Eds.); Sanguinetti, G (Eds.): *Proc. CMSB 2019* Volume 11773, Springer, LNCS/LNBI, September 2019, pp. 302–307

[AHL21a] Assaf, G; Heiner, M ; Liu, F: Colouring fuzziness for systems biology. In: *Theoretical Computer Science* (2021). – ISSN 0304–3975

[AHL21b] Assaf, G; Heiner, M ; Liu, F: Fuzzy Petri Nets in Snoopy - User Manual / Brandenburg University of Technology Cottbus, Department of Computer Science. 2021 (01-34). – Technical Report

[BCP08] Blossey, R.; Cardelli, L. ; Phillips, A.: Compositionality, Stochasticity and Cooperativity in Dynamic Models of Gene Regulation. In: *HFSP Journal* 2 (2008), Nr. 1, pp. 17–28

[BHM15] Chapter 7 In: Blätke, MA; Heiner, M ; Marwan, W: *BioModel Engineering with Petri Nets*. Elsevier Inc., March 2015, pp. 141–193

[BMZM18] Bordon, J.; Mokon, Miha; Zimic, Nikolaj ; Mraz, Miha: Semi-quantitative Modelling of Gene Regulatory Processes with Unknown Parameter Values Using Fuzzy Logic and Petri Nets. In: *Fundamenta Informaticae* 160 (2018), 04, pp. 81–100

[BPS99] Brailsford, S. C.; Potts, C. N. ; Smith, B. M.: Constraint Satisfaction Problems: Algorithms and Applications. In: *European Journal of Operational Research* 119 (1999), Nr. 3, pp. 557–581

[BR12] B, Beachkofski; R, Grandhi: *Improved Distributed Hypercube Sampling.* 2012

[CBL17] Chiou, Jian-Geng; Balasubramanian, Mohan K. ; Lew, Daniel J.: Cell Polarity in Yeast. In: *Annual review of cell and developmental biology* 33 (2017), pp. 77–101. – ISSN 1530–8995

[CGH21] Connolly, S; Gilbert, D ; Heiner, M: From Epidemic to Pandemic Modelling / Brunel University London and Brandenburg University of Technology Cottbus. 2021 (arXiv preprint). – Technical Report

[CH18] Chodak, J; Heiner, M: Spike - a command line tool for continuous, stochastic & hybrid simulation of (coloured) Petri nets . In: *Proc. 21th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2018).* University of Augsburg, October 2018, pp. 1–6

[CH19] Chodak, J; Heiner, M: Spike – reproducible simulation experiments with configuration file branching. In: Bortolussi, L (Eds.); Sanguinetti, G (Eds.): *Proc. CMSB 2019* Volume 11773, Springer, September 2019, pp. 315–321

[CK04] Christensen, S.; Kristensen, L.: Implementing Coloured Petri Nets Using a Functional Programming Language. In: *Higher-Order and Symbolic Computation* 17 (2004), pp. 207–243. – ISSN 0965–9978

[DLPK10] Drawert, B; Lawson, M; Petzold, L ; Khammash, M: The diffusive finite state projection algorithm for efficient simulation of the stochastic reaction-diffusion master equation. In: *The Journal of Chemical Physics* 132 (2010), Nr. 7, pp. 074101

[DRGP11a] Daigle, Bernie J.; Roh, Min K.; Gillespie, Dan T. ; Petzold, Linda R.: Automated estimation of rare event probabilities in biochemical systems. In: *The Journal of Chemical Physics* 134 (2011), Nr. 4, pp. 044110

[DRGP11b] Daigle, Bernie J.; Roh, Min K.; Gillespie, Dan T. ; Petzold, Linda R.: Automated estimation of rare event probabilities in biochemical systems. In: *The Journal of Chemical Physics* 134 (2011), Nr. 4, pp. 044110

[DRQ+20] Duan, Yingying; Rong, Haina; Qi, Dunwu; Valencia-Cabrera, Luis; Zhang, Gexiang ; Pérez-Jiménez, Mario J.: A Review of Membrane Computing Models for Complex Ecosystems and a Case Study on a Complex Giant Panda System. In: *Complexity* 2020 (2020), Sep, pp. 1312824. – ISSN 1076–2787

[dss] Git repositories for dssd_util, Available at `https://github.com/PetriNuts/dssd_util` [Online; accessed 25-July-2021]

[EL00] Elowitz, Michael B.; Leibler, Stanislas: A synthetic oscillatory network of transcriptional regulators. In: *Nature* 403 (2000), Jan, Nr. 6767, pp. 335–338. – ISSN 1476–4687

[ELMS21] Enevoldsen, S; Larsen, K; Mariegaard, A ; Srba, J: Dependency graphs with applications to verification. In: *Proc. PETRI NETS 2013* Volume 22, Springer, October 2021. – ISSN 1433–2787, pp. 635–654

[FRS19] Franciscus, Nigel; Ren, Xuguang ; Stantic, Bela: Dependency graph for short text extraction and summarization. In: *Journal of Information and Telecommunication* 3 (2019), 04, pp. 1–17

[Gec] Gecode: Generic Constraint Development Environment, Available at `http://www.gecode.org` [Online; accessed 15-May-2021]

[GH06] Gilbert, D.; Heiner, M.: From Petri nets to differential equations - an integrative approach for biochemical network analysis. In: Donatelli, Susanna (Eds.); Thiagarajan, P. (Eds.): *proceedings of Petri Nets and Other Models of Concurrency - ICATPN 2006, Lecture Notes in Computer Science* Volume 4024. Springer Berlin / Heidelberg, LNCS, 2006, pp. 181–200

[GHL07] Gilbert, D.; Heiner, M. ; Lehrack, S.: A unifying framework for modelling and analysing biochemical pathways using Petri nets. In: Calder, Muffy (Eds.); Gilmore, Stephen (Eds.): *Computational Methods in Systems Biology, Lecture Notes in Computer Science* Volume 4695. Springer Berlin / Heidelberg, 2007, pp. 200–216

[Gil76] Gillespie, D.: A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. In: *J. Comput. Phys.* 22 (1976), Nr. 4, pp. 403 – 434

[Gil77] Gillespie, D.: Exact stochastic simulation of coupled chemical reactions. In: *J. Phys. Chem.* 81 (1977), Nr. 25, pp. 2340–2361

[GL79] Genrich, HJ; Lautenbach, K: The analysis of distributed systems by means of predicate/transition-nets. In: Kahn, G (Eds.): *Semantics of Concurrent Computation* Volume 70, Springer, 1979, pp. 123–146

[GL81] Genrich, H. J.; Lautenbach, K.: System Modelling with High-Level Petri Nets. In: *Theoretical Computer Science* 13 (1981), Nr. 1, pp. 109–135

[GRHS00] Gouze, Jean-Luc; Rapaport, Alain ; Hadj-Sadok, M.Z.: Interval Observers for Uncertain Biological Systems. In: *Ecological Modelling* 133 (2000), 08, pp. 45–56

[HBG+05] Hindmarsh, A.; Brown, P.; Grant, K.; Lee, S.; Serban, R.; Shumaker, D. ; Woodward, C.: SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. In: *ACM Trans. Math. Softw.* 31 (2005), pp. 363–396

[Her13] Herajy, M: *Computational Steering of Multi-Scale Biochemical Networks*, BTU Cottbus, Computer Science Institute, PhD thesis, January 2013

[HGD08] Heiner, M; Gilbert, D ; Donaldson, R: *LNCS*. Volume 5016: *Petri Nets for Systems and Synthetic Biology*. Springer, 2008, pp. 215–264

[HH94] Hoffman, F. O.; Hammonds, Jana S.: Propagation of Uncertainty in Risk Assessments: The Need to Distinguish Between Uncertainty Due to Lack of Knowledge and Uncertainty Due to Variability. In: *Risk Analysis* 14 (1994), Nr. 5, pp. 707–712

[HH14] Herajy, M; Heiner, M: A Steering Server for Collaborative Simulation of Quantitative Petri Nets. In: *Proc. PETRI NETS 2014* Volume 8489, Springer, June 2014, pp. 374–384

[HH15] Herajy, M; Heiner, M: Modeling and Simulation of Multi-scale Environmental Systems with Generalized Hybrid Petri Nets. In: *Frontiers in Environmental Science* 3 (2015), Nr. 53. – ISSN 2296–665X

[HH18a] Herajy, M; Heiner, M: Adaptive and Bio-semantics of Continuous Petri Nets: Choosing the Appropriate Interpretation. In: *Fundamenta Informaticae* 160 (2018), Nr. 1-2, pp. 53–80

[HH18b] Herajy, M; Heiner, M: An Improved Simulation of Hybrid Biological Models with Many Stochastic Events and Quasi-Disjoint Subnets. In: M Rabe, N Mustafee A Skoogh S J. (Eds.); Johansson, B (Eds.): *Proceedings of the 2018 Winter Simulation Conference (WSC 2018), Gothenburg, Sweden*, IEEE, December 2018 (978-1-5386-6572-5/18). – WSC 2018, December 9-12, 2018, pp. 1346–1357

[HHL+12] Heiner, M; Herajy, M; Liu, F; Rohr, C ; Schwarick, M: Snoopy – a unifying Petri net tool. In: *Proc. PETRI NETS 2012* Volume 7347, Springer, LNCS, June 2012, pp. 398–407

[HJSS06] Helton, J.C.; Johnson, J.D.; Sallaberry, C.J. ; Storlie, C.B.: Survey of sampling-based methods for uncertainty and sensitivity analysis. In: *Reliability Engineering & System Safety* 91 (2006), Nr. 10, pp. 1175–1209. – The Fourth International Conference on Sensitivity Analysis of Model Output (SAMO 2004). – ISSN 0951–8320

[HLGM09] Heiner, M; Lehrack, S; Gilbert, D ; Marwan, W: Extended Stochastic Petri Nets for Model-based Design of Wetlab Experiments. In: *Transactions on Computational Systems Biology XI* 5750 (2009), pp. 138–163

[HRS13] Heiner, M; Rohr, C ; Schwarick, M: MARCIE - Model checking And Reachability analysis done effiCIEntly. In: Colom, JM (Eds.); Desel, J (Eds.): *Proc. PETRI NETS 2013* Volume 7927, Springer, June 2013, pp. 389–399

[HSW15] Heiner, M; Schwarick, M ; Wegener, J: Charlie an extensible Petri net analysis tool. In: Devillers, R (Eds.); Valmari, A (Eds.): *Proc. PETRI NETS 2015* Volume 9115, Springer, LNCS, June 2015, pp. 200–211

[Iba05] Ibarra, Oscar H.: On membrane hierarchy in P systems. In: *Theoretical Computer Science* 334 (2005), Nr. 1, pp. 115–129. – ISSN 0304–3975

[IHHA18] Ismail, A; Herajy, M; Heiner, M ; Atlam, E: An Efficient Approach for the Hybrid Simulation of Intracellular Calcium Dynamics. In: *13th International Conference on Computer Engineering and Systems (ICCES)* IEEE, 2018. – 18-19 December, 2018, Cairo, Egypt, pp. 665–670

[Jen81] Jensen, Kurt: Coloured Petri nets and the invariant-method. In: *Theoretical computer science* 14 (1981), Nr. 3, pp. 317–336

[Jen92] Jensen, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol 1, Basic Concepts.* Berlin Heidelberg: Springer, LNCS, 1992

[JKW07] Jensen, K.; Kristensen, L. M. ; Wells, L. M.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. In: *International Journal on Software Tools for Technology Transfer* 9 (2007), Nr. 3/4, pp. 213–254

[KDS09] Kaltenbach, Hans-Michael; Dimopoulos, Sotiris ; Stelling, Jörg: Systems analysis of cellular networks under uncertainty. In: *FEBS Letters* 583 (2009), Nr. 24, pp. 3923–3930

[KHR10] Kim, Seongdong; Hoffmann, Christoph ; Ramachandran, Varun: Analyzing the Parameters of Prey-Predator Models for Simulation Games. In: Yang, Hyun S. (Eds.); Malaka, Rainer (Eds.); Hoshino, Junichi (Eds.) ; Han, Jung H. (Eds.): *Entertainment Computing - ICEC 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. – ISBN 978–3–642–15399–0, pp. 216–223

[KTC+06] Koh, Geoffrey; Teong, Huey; Clement, Marie-Veronique; Hsu, David ; Thiagarajan, P.: A decompositional approach to parameter estimation in pathway modeling: A case study of the Akt and MAPK pathways and their crosstalk. In: *Bioinformatics (Oxford, England)* 22 (2006), 08, pp. e271–80

[LAC21] Liu, F; Assaf, G ; Chen, M: A Petri nets-based framework for whole-cell modeling. In: *Bio Systems* (2021). – under review

[LBHY14] Liu, Fei; Blätke, Mary-Ann; Heiner, Monika ; Yang, Ming: Modelling and simulating reactiondiffusion systems using coloured Petri nets. In: *Computers in Biology and Medicine* 53 (2014), pp. 297–308. – ISSN 0010–4825

[LC18] Liu, Fei; Chen, Siyuan: Colored Fuzzy Petri Nets for Dealing with Genetic Regulatory Networks. In: *Fundamenta Informaticae* 160 (2018), 04, pp. 101–118

[LCHS18] Liu, F.; Chen, S.; Heiner, M. ; Song, H.: Modeling Biological Systems with Uncertain Kinetic Data Using Fuzzy Continuous Petri Nets. In: *BMC Systems Biology* 12 (2018), pp. 64–74

[LH13] Liu, F; Heiner, M: Modeling membrane systems using colored stochastic Petri nets. In: *Nat. Computing* 12 (2013), Nr. 4, pp. 617 – 629. – ISSN 1567–7818

[LH14] Chapter 9 In: Liu, F; Heiner, M: *Petri Nets for Modeling and Analyzing Biochemical Reaction Networks*. Springer, 2014, pp. 245–272. – ISBN 978–3–642–41280–6

[LHG20] Liu, F; Heiner, M ; Gilbert, D: Fuzzy Petri nets for modelling of uncertain biological systems. In: *Briefings in Bioinformatics* 21, issue 1 (January 2020), pp. 198–210

[LHR12a] Liu, F; Heiner, M ; Rohr, C: Manual for Colored Petri Nets in Snoopy / Brandenburg University of Technology Cottbus, Department of Computer Science. 2012 (02-12). – Technical Report

[LHR12b] Liu, F; Heiner, M ; Rohr, C: Manual for Colored Petri Nets in Snoopy / BTU Cottbus, Computer Science Institute. 2012 (02–12). – Technical Report

[LHY12] Liu, F; Heiner, M ; Yang, M: An efficient method for unfolding colored Petri nets. In: *Proceedings of the 2012 Winter Simulation Conference (WSC 2012), Berlin*, IEEE, 2012 (978-1-4673-4781-5/12)

[LHY16] Liu, F; Heiner, M ; Yang, M: Fuzzy stochastic Petri nets for modeling biological systems with uncertain kinetic parameters. In: *PLoS ONE* 11 (2016), Nr. 2, pp. 1–19

[LIBa] Latin Hypercube Sample (LHS) library, Available at `https://github.com/bertcarnell/lhslib/` [Online; accessed 10-April-2021]

[LIBb] Wxwidgets library, Available at `https://www.wxwidgets.org/downloads/` [Online; accessed 25-June-2021]

[Liu12] Liu, F.: *Colored Petri Nets for Systems Biology*, BTU Cottbus, Computer Science Institute, PhD thesis, January 2012

[Lot09] Lotka, Alfred J. *Contribution to the Theory of Periodic Reactions*. Januar 1909

[LR95] Lautenbach, K.; Ridder, H.: A Completion of the S-invariance Technique by Means of Fixed Point Algorithms / Universität Koblenz-Landau. 1995 (10–95). – Technical Report

[LSHG19] Liu, F; Sun, W; Heiner, M ; Gilbert, D: Hybrid modelling of biological systems using fuzzy continuous Petri nets. In: *Briefings in Bioinformatics* (2019), 12. – ISSN 1477–4054

[LYLT17] Liu, Hu-Chen; You, Jian-Xin; Li, ZhiWu ; Tian, Guangdong: Fuzzy Petri nets for knowledge representation and reasoning: A literature review. In: *Engineering Applications of Artificial Intelligence* 60 (2017), pp. 45–56. – ISSN 0952–1976

[Mam77] Mamdani: Application of Fuzzy Logic to Approximate Reasoning Using Linguistic Synthesis. In: *IEEE Transactions on Computers* C-26 (1977), Nr. 12, pp. 1182–1191

[Mat] Matlab website, Available at `https://de.mathworks.com/products/matlab.html` [Online; accessed 25-June-2021]

[Mat89] Mattern, F: *Verteilte Basisalgorithmen. Informatik-Fachberichte*. Volume 226. Springer, LNCS, 1989. – in German

[MBC79] Mckay, M. D.; Beckman, R. J. ; Conover, W. J.: A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. In: *Technometrics* 21 (1979), pp. 239–245

[MHRK08] Marino, S; Hogue, I; Ray, C ; Kirschner, D: A Methodology For Performing Global Uncertainty And Sensitivity Analysis In Systems Biology. In: *Journal of Theoretical Biology* 254 (2008), 09, pp. 178–196

[MNP11] Mahapatra, S.S.; Nanda, Santosh K. ; Panigrahy, B.K.: A Cascaded Fuzzy Inference System for Indian river water quality prediction. In: *Advances in Engineering Software* 42 (2011), Nr. 10, pp. 787–796. – ISSN 0965–9978

[MRH12] Chapter 21 In: Marwan, W; Rohr, C ; Heiner, M: *Methods in Molecular Biology*. Volume 804: *Petri nets in Snoopy: A unifying framework for the graphical display, computational modelling, and simulation of bacterial regulatory networks*. Humana Press, 2012, pp. 409–437

[MTA+03] Matsuno, Hiroshi; Tanaka, Yukiko; Aoshima, Hitoshi; Doi, Atsushi; Matsui, Mika ; Miyano, Satoru: Biopathways representation and simulation on hybrid functional Petri Net. In: *In silico biology* 3 (2003), 02, pp. 389–404

[OGD] OGDF: Open Graph Drawing Framework, Available at `http://https://ogdf.uos.de/` [Online; accessed 10-May-2021]

[Pah09] Pahle, J.: Biochemical simulations: stochastic, approximate stochastic and hybrid approaches. In: *Brief Bioinform* 10 (2009), Nr. 1, pp. 53–64

[Pet] Git repositories for PetriNuts, Available at `https://github.com/PetriNuts` [Online; accessed 25-July-2021]

[Pet62] Petri, Carl A.: *Kommunikation mit Automaten*, Universität Hamburg, PhD thesis, 1962

[PR02] Pun, Gheorghe; Rozenberg, Grzegorz: A guide to membrane computing. In: *Theoretical Computer Science* 287 (2002), Nr. 1, pp. 73–100. – Natural Computing. – ISSN 0304–3975

[ROH17] ROHR, C.: *Simulative analysis of coloured extended stochastic Petri nets*, BTU Cottbus, Computer Science Institute, PhD thesis, January 2017

[Rüd93] Rüdiger, V: Bridging the Gap Between Place- and Floyd-Invariants with Applications to Preemptive Scheduling. In: Marsan, M., Ajmone (Eds.): *Application and Theory of Petri Nets 1993, Proceedings 14th International*

Conference, Chicago, Illinois, USA Volume 691, Springer, LNCS, 1993, pp. 433–452

[SH10] Soliman, S; Heiner, M: A Unique Transformation from Ordinary Differential Equations to Reaction Networks. In: *PLoS ONE* 5 (2010), Nr. 12, pp. e14284

[sps] spsim library, Available at `https://github.com/PetriNuts/SP_Simulator.git` [Online; accessed 25-June-2021]

[SRAJ18] Shafiekhani, S.; Rahbar, S.; Akbarian, F. ; Jafari, A. H.: Fuzzy Stochastic Petri Net with Uncertain Kinetic Parameters for Modeling Tumor-Immune System. In: *2018 25th National and 3rd International Iranian Conference on Biomedical Engineering (ICBME)*, 2018, pp. 1–5

[SRH11] Schwarick, M.; Rohr, C. ; Heiner, M.: MARCIE - Model checking And Reachability analysis done effiCIEntly. In: *Proc. QEST 2011*, 2011, pp. 91–100

[SRL+20] Schwarick, M; Rohr, C; Liu, F; Assaf, G; Chodak, J ; Heiner, M: Efficient Unfolding of Coloured Petri Nets using Interval Decision Diagrams. In: Janicki, R (Eds.); Sidorova, N (Eds.) ; Chatain, T (Eds.): *Proc. PETRI NETS 2020* Volume 12152, Springer, LNCS, June 2020, pp. 324–344

[SSW06] Shaw, Oliver; Steggles, Jason ; Wipat, Anil: Automatic Parameterisation of Stochastic Petri Net Models of Biological Networks. In: *Electronic Notes in Theoretical Computer Science* 151 (2006), Nr. 3, pp. 111–129. – Proceedings of the Second International Workshop on the Practical Application of Stochastic Modeling (PASM 2005). – ISSN 1571–0661

[Ste87] Stein, Michael: Large Sample Properties of Simulations Using Latin Hypercube Sampling. In: *Technometrics* 29 (1987), Nr. 2, pp. 143–151

[Sto05] Stocki, Rafal: A method to improve design reliability using optimal Latin hypercube sampling. In: *Computer Assisted Mechanics and Engineering Sciences* 12 (2005), 01, pp. 393–411

[TAAC15] Tuqyah Abdullah, Al Q.; Aboubekeur, Hamdi-Cherif ; Chafia, Kara-Mohamed: State of the Art of Fuzzy Methods for Gene Regulatory Networks Inference. In: *The Scientific World Journal* 2015 (2015), Nr. 24, pp. 3923–3930

[Tsa93] Tsang, Edward: Chapter 3 - Fundamental concepts in the CSP. In: Tsang, Edward (Eds.): *Foundations of Constraint Satisfaction*. Academic Press, 1993. – ISBN 978–0–12–701610–8, pp. 53–78

[VLG+18] Vernon, I; Liu, J; Goldstein, M; Rowe, J; Topping, J ; Lindsey, K: Bayesian uncertainty analysis for complex systems biology models: emulation, global parameter searches and evaluation of gene functions. In: *BMC Systems Biology* 12 (2018). – ISSN 1752–0509

[V.R06] V.R.L., Shen: Knowledge Representation Using High-Level Fuzzy Petri Nets. In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 36 (2006), Nr. 6, pp. 1220–1227

[Wik21a] Wikipedia contributors: *Latin hypercube sampling*. `https://en.wikipedia.org/wiki/Latin_hypercube_sampling`. 2021. – [Online; accessed 1-Jan-2022]

[Wik21b] Wikipedia contributors: *LotkaVolterra equations — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Lotka%E2%80%93Volterra_equations&oldid=1028368215`. 2021. – [Online; accessed 13-June-2021]

[Win13] Windhager, Lukas. *Modeling of dynamic systems with Petri nets and fuzzy logic*. April 2013

[WVR+12] Wang, G; Valerdi, R; Roedler, G; Ankrum, A ; Jr, J: Harmonising software engineering and systems engineering cost estimation. In: *International Journal of Computer Integrated Manufacturing* 25 (2012), 04, pp. 432–443

[Zad65] Zadeh, L.: Fuzzy sets. In: *Information and Control* 8 (1965), pp. 338–353

[Zim10] Zimmermann, H.-J.: Fuzzy set theory. In: *WIREs Computational Statistics* 2 (2010), Nr. 3, pp. 317–332

[ZMC14] Zhang, Xixiang; Ma, Weimin ; Chen, Liping: New Similarity of Triangular Fuzzy Number and Its Application. In: *The Scientific World Journal* 2014 (2014)

A Appendices

A.1 Fuzzy Petri nets

In this section, we give some useful material for both uncoloured and coloured fuzzy Petri nets, including:

- Fuzzy Petri nets manual, which includes numerous case studies. The manual is accessible from:
  - `https://www-dssz.informatik.tu-cottbus.de/publications/btu-reports/fpn_manual.pdf`.
- Snooy's $\mathcal{FPN}$files for all fuzzy Petri net case studies which are presented in this thesis as well as in the manual. These files are accessible from:
  - `https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Examples?dir=fpn`.
- Some video clips showing the principle of modelling and simulating $\mathcal{FPN}$ in Snoopy; please navigate the following link:
  - `https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Snoopy#manuals`.

A.2 The BNF for colour expressions in Snoopy's $\mathcal{PN}^{\mathcal{C}}$

Here we present the extended grammar of the colour expressions for coloured Petri nets in Snoopy:

ColorExpr	::=	*MultiSetExpr*
MultiSetExpr	::=	*Predicate* | *MultiSetExpr MSAddDiffOp Predicate*
MSAddDiffOp	::=	"++" | "--"
Predicate	::=	*SeparatorExpr* | "[" *OrExpr* "]" *SeparatorExpr*
SeparatorExpr	::=	*TupleExpr* | *SeparatorExpr SeparatorOp TupleExpr*
SeparatorOp	::=	"`"
TupleExpr	::=	*OrExpr*| "(" *CommaExpr* ")"
CommaExpr	::=	*TupleExpr* | *CommaExpr CommaOp TupleExpr*
CommaOp	::=	","
OrExpr	::=	*AndExpr* | *OrExpr OrOp AndExpr*
AndExpr	::=	*EqualExpr* | *AndExpr AndOp EqualExpr*
AndOp	::=	"&" | "&&"
EqualExpr	::=	*RelationExpr* | *EqualExpr EqualOp EqualExpr*
EqualOp	::=	"=" | "==" | "!=" | "<>"
RelationExpr	::=	*AddExpr* | *RelationExpr RelationOp AddExpr*
RelationOp	::=	"<" | "<=" | ">=" | ">" | "elemOf"
AddExpr	::=	*MultiplicityExpr*| *AddExpr AddOp MultiplicityExpr*
AddOp	::=	"+" | "-"
MultiplicityExpr	::=	*UnaryExpr* | *MultiplicityExpr MultiplicityOp UnaryExpr*
MultiplicityOp	::=	"*" | "/" | "%" | "^"
UnaryExpr	::=	*PostfixExpr* | *UnaryOp PostfixExpr*
UnaryOp	::=	"+" | "-" | "@" | "!"

PostfixExpr	::=	*AtomExpr* | *PostfixExpr* "[" *AtomExpr* "]" |
		PostfixExpr ":" *AtomExpr*
AtomExpr	::=	*Constant* | *Variable* | *Function* | "(" *ColorExpr* ")" |
		AllFun
AllFun	::=	"all" "(" ")"
Constant	::=	*Integer* | *String*
Variable	::=	*Identifier*
Function	::=	*Identifier* "(" *ArgumentList*")" "{" *FunctionBody* "}"
ArgumentList	::=	*OrExpr* | *ArgumentList CommaOp OrExpr*
FunctionBody	::=	*MultiSetExpr*
Integer	::=	*Digit* | *Integer Digit*
String	::=	*LetterOrDigit* | *String LetterOrDigit*
Identifier	::=	*Letter* | *Identifier LetterOrDigit*
LetterOrDigit	::=	*Letter* | *Digit*
Digit	::=	"0-9"
Letter	::=	"a-zA-Z"

A.3 Snoopy's Command-line Feature

Here we present one application scenario for Snoopy's command-line feature. In this application, we call Snoopy for performing an automatic layout for a set of Petri net files represented as *andl* files. For each file, we perform the layout by means of one of the layout algorithms (here *FMMM*). Afterwards, we export the obtained Petri net to the *eps* format. Finally, we use an external tool, e.g. *epstopdf* to convert each *eps* file to an *pdf* file.

Listing 3: Python script for performing automatic layout by calling Snoopy using the command-line.

```
# call as: python runSnoopyAutoLayout.py
#
#!/usr/bin/python

import sys
import os
import subprocess
import argparse
import math
import shutil
import re
import glob

#to be adjusted by the snoopy executable - in macosx:
#    right click snoopy (in Applications folder),
#    then choose 'show Package Contents',
#    then 'Contents' folder,
#    then MacOS, add this path here
## no '/' at the end
#Snoopy="/Applications/snoopy.app/Contents/MacOS/snoopy"

## or add snoopy path to  .bash_profile
Snoopy="snoopy"

#to be adjusted to your .andl files Folder;
#there should be '/' at the end
MODELS="Dropbox/Waddington_landscape/woodstock/test-layoutAutomised/"

def createConfig(model,start):
```

```
30    configFile = Snoopy +" "+ model +" "+ start
31    return  configFile
32
33  def createConfigPDF(model):
34    configFile =  "epstopdf "+ model +".eps"
35    return  configFile
36
37  if __name__ == "__main__":
38   #fetch all ANDL files from the folder whose path,
39   # is determined by the global variable MODELS.
40    all_files=glob.glob(MODELS+"*.andl")
41
42    #iterate all the fetched andl files and ,
43    #for each one do export run Snooy command and then convert to pdf
44    for file in all_files:
45      # adjust command line options in the second param of this function
46      config = createConfig(file,"-l1p1sc")
47      confINK = createConfigPDF(file)
48      print(config)
49      subprocess.call(config,shell=True)# call snoopy
50      subprocess.call(confINK,shell=True)# call epstopdf
51      os.remove(file+".eps")#remove the eps files,
52      #after converting them to pdf
```