

Spike - a tool for reproducible simulation experiments

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

Jacek Chodak

geboren am 1977-02-21 in Barlinek, Polen

Vorsitzender: Prof. Dr. rer. nat. habil. Klaus Meer
Gutachterin: Prof. Dr.-Ing. Monika Heiner
Gutachter: Prof. PhD David Gilbert

Tag der mündlichen Prüfung: 2021-12-15

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 work was conducted from 2017 to 2021 under the supervision of Prof. Dr.-Ing Monika Heiner at Brandenburg University of Technology Cottbus-Senftenberg.

Jacek Chodak

Thanks to all who made it possible.

Abstract

Reproducibility of simulation experiments is still a significant challenge and has attracted considerable attention in recent years. One cause of this situation is bad habits of the scientific community. Many results are published without data or source code, and only a textual description of the simulation set-up is provided. Other causes are: no complete simulation set-up, no proper output data analysis and inconsistency of published data, which makes it impossible to compare results.

The progress of computational modelling, amount of data and complexity of models requires designing experiments in such a way that ensures reproducibility. A textual description does not provide all the needed details. A computer code is more reliable than a textual description. It is the precise specification that describes a simulation configuration, model, etc. When computer code, data, models and all parameters are provided, the simulation results become reproducible.

The main goal of this thesis is to develop a tool that ensures reproducibility and efficient execution of simulation experiments, often involving many individual simulation runs. The tool should support a wide range of application scenarios, where the typical scenario is simulation of biochemical reaction networks, which are represented as (coloured) Petri nets interpreted in the stochastic, continuous or hybrid paradigm. The model to be simulated can be given in various formats, including SBML.

The result is a command line tool called Spike, which can be used for various scenarios, including benchmarking, simulation of adaptive models and parameter optimization. It builds on a human-readable configuration script *SPC*, supporting the efficient specification of multiple model configurations as well as multiple simulator configurations in a single configuration file.

Keywords: continuous, stochastic, hybrid, coloured (hierarchical) Petri nets; parallel simulation; configuration; reproducibility; parameter scanning; parameter optimization; simulation of adaptive models

Abstrakt

Die Reproduzierbarkeit von Simulationsexperimenten ist nach wie vor eine große Herausforderung und hat in den letzten Jahren viel Aufmerksamkeit bekommen. Eine Ursache für diese Situation sind schlechte Gewohnheiten der wissenschaftlichen Gemeinschaft. Aus irgendwelchen Gründen werden viele Ergebnisse ohne Daten und Quellcode veröffentlicht und es wird nur eine textliche Beschreibung des Simulationsaufbaus gegeben. Andere Ursachen sind: kein ordnungsgemäßer Simulationsaufbau, keine ordnungsgemäße Analyse der Ausgabedaten und Inkonsistenz der veröffentlichten Daten (was einen Vergleich der Ergebnisse unmöglich macht).

Der Fortschritt der computergestützten Modellierung, die Datenmenge und die Komplexität der Modelle erfordern es, Experimente so zu gestalten, dass die Reproduzierbarkeit gewährleistet ist. Eine textliche Beschreibung liefert nicht alle Details. Ein Computercode ist zuverlässiger als eine Textbeschreibung. Es bedarf einer genauen Spezifikation einer Simulationskonfiguration, eines Modells usw. Durch die Bereitstellung von Computercode, Daten, Modellen und allen Parametern werden die Ergebnisse einer Simulation reproduzierbar.

Das Hauptziel dieser Arbeit ist es, ein Werkzeug zu entwickeln, das die Reproduzierbarkeit und effiziente Durchführung von Simulationsexperimenten, die oft viele einzelne Simulationsläufe umfassen, gewährleistet. Das Werkzeug soll vielfältige Anwendungsszenarien unterstützen, wobei das typische Szenario die Simulation biochemischer Reaktionsnetzwerke ist, die als (farbige) Petrinetze dargestellt werden, die im stochastischen, kontinuierlichen oder hybriden Paradigma interpretiert werden. Das zu simulierende Modell kann in verschiedenen Formaten, einschließlich SBML, vorliegen.

Das Ergebnis ist ein Kommandozeilenwerkzeug namens Spike, das in verschiedenen Anwendungsfällen eingesetzt werden kann, darunter Benchmarking, Simulation adaptiver Modelle und Parameteroptimierung. Es basiert auf einem für Menschen lesbaren Konfigurationsskript *SPC* und unterstützt die effiziente Spezifikation mehrerer Modellkonfigurationen sowie mehrerer Simulatorkonfigurationen in einer einzigen Konfigurationsdatei.

Schlagwörter:: kontinuierlich, stochastisch, hybrid, farbig (hierarchisch) Petri Netze; Parallelsimulation; Konfiguration; Reproduzierbarkeit; Parameterscannen; Parameteroptimierung; Simulation von adaptiven Modellen

Contents

List of Figures	v
List of Tables	ix
Glossary	xi
1 Introduction	1
1.1 Motivation	1
1.2 Objectives and Contributions	2
1.3 Organization of the Thesis	4
2 Background and Related Work	7
2.1 Petri Nets	7
2.2 Extended Petri Nets	9
2.3 Quantitative Petri Nets	11
2.4 Coloured Petri Nets	14
2.5 Coloured Quantitative Petri Nets	17
2.6 Unfolding	18
2.6.1 Equivalent Standard Petri Nets	18
2.6.2 Unfolding Algorithm	20
2.7 Simulation	22
2.7.1 Mass-Action Kinetics	22
2.7.2 Stochastic Simulation	25
2.7.3 Deterministic Simulation	29
2.7.4 Hybrid Simulation	32
2.8 Reduction	34
2.8.1 Pruning Clean Siphons	34
2.8.2 Pruning Constant Places	36
2.9 Reproducible Simulation	38
2.9.1 Rules to Drive Reproducible Experiments	38
2.9.2 Encoding of Simulation Experiments	40
2.9.3 Adaptive Model Simulation	43
2.10 Closing Remarks	45

CONTENTS

3	Configuration Language	47
3.1	<i>SPC</i> Format	48
3.2	Experiment Definition	49
3.3	Main <i>SPC</i> Objects	52
3.3.1	Import	52
3.3.2	Configuration	54
3.3.3	Log	57
3.4	Basic definitions	58
3.4.1	Value	58
3.4.2	Literal	58
3.4.3	Variable	60
3.4.4	Object	61
3.4.5	Array	62
3.4.6	Range	63
3.5	Expressions	63
3.5.1	Arithmetic Expression	64
3.5.2	Boolean Expression	65
3.5.3	Comparison Expression	66
3.5.4	Concatenation	66
3.5.5	Precedence	67
3.6	Conditional Block	69
3.7	Stepwise Simulation	70
3.8	Configuration Branching	73
3.9	Closing Remarks	74
4	Spike Architecture	77
4.1	Spike Functionality	79
4.2	Simulation	81
4.3	Parallel Simulation	82
4.4	Inter-Process Communication	83
4.5	Stepwise Simulation	86
4.6	Reproducible Stochastic Simulation	88
4.7	Conversion	88
4.8	IDD-based unfolding	89
4.8.1	IDD Reduction	90
4.8.2	Unfolding	90
4.8.3	Algorithms	92
4.8.4	The <i>elemOf</i> Operator and Boolean Colour Set	97
4.9	Closing Remarks	105

5	Use Cases	107
5.1	Benchmarking	107
5.2	Simulation of Adaptive Models	120
5.3	Spike as a Backend Simulator for Parameter Optimization	133
5.4	Closing Remarks	142
6	Conclusions and Outlook	143
6.1	Conclusions	143
6.2	Outlook	144
6.3	Availability	145
6.4	Acknowledgement	145
	References	147
	Appendices	151
A	Grammar of Configuration Script	153
A.1	Graphical notations	153
A.2	Main <i>SPC</i> Objects	154
A.3	Basic definitions	155
A.4	Expressions	156
A.5	Conditional Block	158
A.6	onStep	158
B	Source Code: Heuristic Method of Parameter Optimization	159
B.1	SIR Model In ANDL Format: SIR-SPN.andl	159
B.2	SPC Configuration Template: SIR-CPN-spc.tmp	160
B.3	Experiment Set-up in Python: optimization.py	161
B.4	Genetic Algorithm Library for Python: geneticalgorithmjch.py	165
C	A Quick Guide to SPC	175

List of Figures

2.1	SIR model, as \mathcal{PN}	9
2.2	SIR model, as \mathcal{XPN}	11
2.3	SIR model as \mathcal{SPN}	13
2.4	Variation of the SIR model as \mathcal{XSPN}	14
2.5	SIR model, as coloured \mathcal{PN}	17
2.6	SIR model, as coloured \mathcal{SPN}	18
2.7	SIR model, as coloured \mathcal{CPN}	18
2.8	SIR model, as coloured \mathcal{HPN}	18
2.9	A simple reaction as \mathcal{SPN}	22
2.10	Influence of the number of simulation runs N on recorded and approximate stochastic simulation traces of \mathcal{SPN} model	27
2.11	Simulation traces of the stochastic \mathcal{XPN} model	27
2.12	Comparison of the simulation traces between \mathcal{SPN}^c and \mathcal{CPN}^c models	28
2.13	Decay events of a disease	28
2.14	SIR model, as \mathcal{CPN}	29
2.15	Simulation traces of \mathcal{HPN}^c model	34
2.16	Removing insufficiently marked siphons	36
2.17	Replacing a constant place	37
3.1	High level overview of the relations between main components of the experiment	50
3.2	Graphical representation of relations between the experiment and main \mathcal{SPC} objects	51
3.3	The three main objects of \mathcal{SPC}	52
3.4	Import object	53
3.5	SBML object	53
3.6	Configuration object	54
3.7	Model object	55
3.8	Simulation object	55
3.9	\mathcal{SPC} value	58
3.10	\mathcal{SPC} number	58
3.11	\mathcal{SPC} string	59

LIST OF FIGURES

3.12	<i>SPC</i> logical value	59
3.13	<i>SPC</i> declaration	60
3.14	<i>SPC</i> identifier	60
3.15	<i>SPC</i> assign	60
3.16	<i>SPC</i> object	61
3.17	<i>SPC</i> access	62
3.18	<i>SPC</i> array	62
3.19	<i>SPC</i> range	63
3.20	<i>SPC</i> expression	64
3.21	<i>SPC</i> arithmetic	64
3.22	<i>SPC</i> boolean	66
3.23	<i>SPC</i> comparison	67
3.24	<i>SPC</i> string concatenation	67
3.25	<i>SPC</i> conditional block	69
3.26	onStep object	71
3.27	do object	71
3.28	Graphical representation of branching	73
4.1	Graphical representation of commands dispatching	78
4.2	Graphical representation of commands flow	78
4.3	<i>PetriNuts</i> framework	79
4.4	Overview of Spike functionality	80
4.5	High level overview of performing parallel simulations	83
4.6	Life cycle of a broker and a worker process	84
4.7	Inter-process communication	86
4.8	Graphical representation of instantiating simulation threads	88
4.9	Data format conversions supported by Spike	89
4.10	Data format conversions supported by Spike	90
4.11	Coloured <i>SPN</i> SIR model with a more flexible solution to specify colour-dependent rate functions	99
4.12	SIR model, as coloured <i>SPN</i> driven by <i>elemOf</i> expressions	100
4.13	SIR model, as coloured <i>SPN</i> with mutual exclusion	102
4.14	Stepwise <i>IDD</i> computation	104
5.1	Comparison of simulation traces between deterministic and stochastic SIR model used in Example 5.1.	111
5.2	SIR and SEIR models used in Example 5.2.	114
5.3	Comparison of the simulation traces between SIR and SEIR models used in Example 5.2.	115
5.4	SIR models used in Example 5.1.	119
5.5	Control feedback loop	120
5.6	Results of the stepwise simulation	126
5.7	SIR model used in Example 5.5.	127

LIST OF FIGURES

5.8	Results of the simulation - <i>bigbang</i> vs <i>smooth</i> relaxation rules	132
5.9	SIR model as CPN.	135
5.10	Reference data traces	135
5.11	Optimization through simulation	137
5.12	Progress of the optimization	140
5.13	Reference data traces and optimization	141

List of Tables

3.1	Arithmetic operators.	65
3.2	Boolean operators.	65
3.3	The truth table of boolean operations.	65
3.4	Test operators.	66
3.5	Precedence of operators.	67
4.1	List of Spike modules with their commands.	79
4.2	List of messages.	85
4.3	The truth table of <i>IDD</i> Boolean expression.	101
5.1	The performance of hybrid simulation algorithms supported by Spike. . .	119

Glossary

Name	Description	Page List
BDD	Binary decision diagrams	88
CPN	Continuous Petri net	v, 4, 12, 13, 18, 26, 30, 32, 43, 86
CPN^c	Coloured Continuous Petri net	v, 4, 17, 43
$CTMC$	Continuous Time Markov Chain	12, 13, 25
DAG	Directed acyclic graphs	88
HPN	Hybrid Petri net	v, 4, 12, 13, 18, 26, 33, 43
HPN^c	Hybrid Continuous Petri net	v, 4, 17, 34, 43
IDD	Interval Decision Diagrams	vi, 20, 88--91, 93, 95, 96, 99
\mathbb{N}	Set of positive integer numbers	8
\mathbb{N}_0	Set of non-negative integer numbers	8
ODE	Ordinary Differential Equation	13, 28--31, 34
PN	Petri net	v, 4, 7--9, 14, 17, 18, 22, 34--38, 41--43, 52, 76, 102, 135, 136
PN^c	Coloured Petri net	7, 14, 16--20, 34, 81, 88, 102, 136
\mathbb{Q}^+	Set of positive rational numbers	8
\mathbb{R}_0^+	Set of non-negative real numbers	8
$ROIDD$	Reduced ordered interval decision diagrams	88--90
SPC	Spike's configuration	ii, iii, v--vii, ix, 3, 4, 40, 41, 44, 46--51, 53, 55, 57--63, 65--68, 73, 102--104, 111, 112, 118, 133, 135, 136, 145, 146
SPN	Stochastic Petri net	v, vi, 4, 12, 13, 18, 22, 25--27, 37, 43, 86, 98--100
SPN^c	Coloured Stochastic Petri net	v, 4, 17, 43
τ	Current time	29
$\mathcal{X}PN$	Extended Petri net	v, 4, 9--12, 14, 27, 35, 43
$\mathcal{X}SPN$	Extended Stochastic Petri net	v, 12, 14
$X(\tau)$	System state at time	29

1

Introduction

1.1 Motivation

Petri nets [Mur89] have been proven to be useful for modelling a wide range of applications, including, among others, biochemical networks [GHL07]. They provide an intuitive graphical representation and a well-developed mathematical theory for system analysis. In addition, Petri nets might bridge the gap between computational theoretician and experimentalist. This thesis focuses on quantitative Petri nets [GHL07] (stochastic, continuous and hybrid Petri nets) and their high level representations, coloured Petri nets [Liu12], which are used as modelling paradigms.

Simulation of biochemical models can be time and memory consuming. Thus, simulations should be delegated for performance reasons to be executed on a server. Additionally, when experiments require running multiple simulations, the time spent can be particularly long, when the individual simulations are merely executed one after another. Frequently, it is required to prepare a set of simulation experiments in order to find appropriate model parameters (e.g., initial conditions, kinetic parameters) or simulator options (e.g., simulator type, length of simulation traces, resolution of the traces recorded). Manual preparation of a new simulation run for each new model and/or simulator configuration is time consuming and potentially error-prone. The reproducibility of the entire experiment suffers if one of the runs is not well documented.

There are a couple of tools allowing the simulation of Petri net models, however most of them have a graphical user interface (GUI) which usually involves additional dependencies. Application tools with a GUI are not well suitable as a simulation process to be executed on a server. Running simulation on a server helps to save user resources and speed up simulations. On a server, a user can schedule multiple simulation experiments which can be executed simultaneously or sequentially. Often, a user wants to check how a model behaves for different sets of parameters. In this case, a user is forced to make changes in the model using an appropriate tool. Each time a model is changed, the simulation needs to be repeated. To compare how a model behaves under different types of simulation algorithms (stochastic, continuous, hybrid), it is necessary

1. INTRODUCTION

to configure, each time separately, the simulation and the model. This scenario can require to use separate tools for different types of simulations. To ensure reproducible simulations, all parameters of the model and simulation configurations have to be saved. To simplify the workflow, the configuration of the model and the configuration of the simulation should be supported by a script language, which allows for easy modification of any model and simulation parameters.

Simulation of dynamically changing processes (which change their dynamic behaviour in response to the occurrence of external events) require an ongoing adaptation in terms of time, quality, and flexibility. Therefore, to simulate such processes, it is necessary to adjust the model according to its current simulation state during the simulation run time. This allows to improve the quality of the simulation results. Such functionality requires the implementation of a stepwise simulation that will allow the simulation of adaptive models.

So far there is no tool that would allow easy configuration of simulation experiments with support of a wide range of Petri nets classes and simulation types. However, there are tools that partially cover some of these issues. For example, the tool COPASI [HSG+06] supports stochastic, deterministic and hybrid simulation of biochemical networks. It allows the definition of the export of multiple results. There is no direct support for Petri nets. Configuration files follow a markup language format, which hinders their readability by a user. In turn the tool Renew (The Reference Net Workshop) [KWD+04] supports modelling and simulation of models designed with the help of the reference nets formalism which is an extension of Petri nets, where tokens can be references to arbitrary objects, especially other nets, thus allowing nested net models. Renew allows running simulation on a server [PJC14], however its core does not support quantitative net classes (stochastic, continuous and hybrid).

1.2 Objectives and Contributions

The developed tool named Spike is part of the *PetriNuts* family of tools for dealing with a variety of related models, for which Petri nets are used as an umbrella modelling paradigm. The *PetriNuts* framework consists of tools for modelling (Snoopy [HHL+12]), analysing (Marcie [HRS13], Charlie [HSW15]), simulating (Snoopy, Marcie, Spike [CH19]) and animating (Snoopy, Patty [Sch08]).

Spike has been designed to address the following objectives:

- Reproducible simulation experiments - the amount of data produced by simulation experiments and the complexity of models requires to design an experiment in such way which ensures reproducibility. By providing computer code, data, models and any parameters to configure model and/or the simulators involved, it is possible to reproduce results of a simulation. Encoding of a model, which is de facto a structure description that can comprise initial conditions and kinetic values, does not sufficiently describe experiments. An experiment should be fully encoded if it

is meant to be reproducible. The encoding should be human-readable and allow the creation of easily modifiable configuration scripts without any special tools.

- Efficient simulation - a simulation experiment can consist of a set of separate simulations. To perform simulation efficiently, the set of simulations should be set up in an automatized way and be executed sequentially or in parallel, depending on available resources.
- Simulation of stochastic, continuous and hybrid Petri nets, coloured and uncoloured ones, as supported by the *PetriNuts* framework.
- Support of a variety of use cases, including: benchmarking, simulation of adaptive models, scanning of model parameters and simulation options, model parameter optimization.

To achieve this, the main contributions of this thesis are:

- Design of a new language *SPC* to specify reproducible simulation experiments.
- Development of a new tool, named Spike, to process *SPC* files.

Spike builds on a human-readable configuration script, supporting the efficient specification of multiple model configurations as well as multiple simulator configurations in a single file. Reproducibility is ensured by the requirement to provide unambiguous parameter values of a model and the simulation engine.

- Support of a variety of use cases, including:
 - Scanning of model parameters and simulation options - when evaluating a configuration, it can be split into separate branches. Branching processes are triggered by defining a set of configuration parameters to scan. A set of values is assigned to each parameter. For each value in the set, a new configuration branch is created. Such a feature allows a configuration script to be split into separate branches, what results in multiple simulation configurations. Each configuration branch is treated as a separate process and can be executed in parallel or sequentially.
 - Benchmarking - Spike supports three types of simulations: stochastic, deterministic and hybrid . Depending on the configuration, a given model is simulated according to the specified simulation type, regardless of the model type. Such functionality allows designing benchmarking experiments, the main goal of which is to compare the performance of the model and simulation algorithms.

1. INTRODUCTION

- Simulation of adaptive models - Through stepwise simulation, Spike allows for the dynamic adaptation of the model during the simulation runtime. The stepwise simulation advances in a given time interval and the parameters (any constants to specify initial markings, arc weights, kinetic parameters) of the model and its state can be adjusted/adapted after each simulation step. The adaptation is based on an evaluation of boolean conditions, that may involve the current state of the model and the simulation.
- Spike as a backend simulator for parameter optimization - Spike features such as parameter scanning and parallel execution of configuration branches make it suitable for performing simulation tasks, while an optimization strategy must be implemented separately.

Coloured models can be unfolded using *IDD*-based unfolding, which is integrated in the internally developed `dssd_util` library. The `dssd_util` library allows Spike to import and export \mathcal{PN} models in various formats (e.g. ANDL, CANDL, SBML, PNML). The `dssd_util` library comes with the stand-alone tool `ANDLconverter`, which allows unfolding coloured Petri nets and prune constant places and clean siphons.

To perform a simulation, Spike uses an internally developed simulation library; it is capable to run three basic types of simulations: stochastic, deterministic and hybrid, where each comes with several algorithms.

Both libraries are integral parts of the *PetriNuts* framework and are used by Snoopy, Marcie and Spike.

1.3 Organization of the Thesis

Chapter 2: Background and Related Work - provides the necessary definitions of the used Petri net classes and briefly introduces the general paradigm of modelling with Petri nets, starting with the definition of basic \mathcal{PN} through extended and quantitative Petri nets, i.e. \mathcal{XPN} , \mathcal{SPN} , \mathcal{CPN} , \mathcal{HPN} , and ending with their coloured counterparts \mathcal{SPN}^c , \mathcal{CPN}^c , \mathcal{HPN}^c . By the use of examples it shows how the developed models behave under three types, i.e. deterministic, stochastic and hybrid simulation.

Chapter 3: Configuration Language - describes the structure and grammar of the Spike configuration script language (*SPC*), the main goal of which is to efficiently support reproducible simulation experiments by setting up model parameters and simulation options.

Chapter 4: Spike Architecture - provides general information about architecture and functionality of Spike. Additionally, it describes implementation aspects of the unfolding of coloured Petri nets by the use of an unfolding engine based on Interval Decision Diagrams.

Chapter 5: Use Cases - illustrates the functionality of Spike based on three use cases:

- benchmarking - to compare the computational complexity of models and simulation algorithms,
- simulation of adaptive models - stepwise simulation as a discrete-time adaptive modelling system,
- Spike as a backend simulator for simulative parameter optimization.

Chapter 6: Conclusions and Outlook - summarizes the achieved results, encountered issues and provides some ideas for future research.

2

Background and Related Work

Petri nets (\mathcal{PN}), originating from the dissertation of Carl Adam Petri [Pet62], provide a modelling paradigm, which fits well for parallel, concurrent, asynchronous and non-deterministic systems. This makes \mathcal{PN} a proper tool for modelling biological systems. However, standard \mathcal{PN} do not easily scale. To overcome this issue, Coloured Petri nets (\mathcal{PN}^c) were introduced as a suitable tool for modelling and analysing biological systems. \mathcal{PN}^c s easily scale and allow for modelling huge systems without loss of the analysis capabilities of standard \mathcal{PN} . This is possible through automatic unfolding of a \mathcal{PN}^c to its corresponding uncoloured \mathcal{PN} . This is a necessary step to apply analysis and simulation techniques as most of them require standard Petri nets. Applying simulation techniques allows for analysis of dynamic behaviours of a modelled system. It is an essential tool for studying biochemical systems. Simulation types are divided into three main classes: deterministic, stochastic and hybrid. Which one will be applied depends on the model as well as the properties of interest.

After unfolding, the number of nodes can be much larger than in its coloured counterpart. Reduction of a model may yield a more optimized (in terms of size) model, provide insights into structural properties and reduces a simulation overhead. The main challenge of a reduction is to preserve the main three properties of a \mathcal{PN} model: liveness, reversibility and boundedness. The two simplest techniques that preserve the main three properties are pruning of clean siphons and constant places.

All of this should result in reproducible experiments. An experimenter should design an experiment in a way that ensures reproducibility by obtaining consistent results when using the same input data.

2.1 Petri Nets

A Petri net is a directed bipartite graph. Nodes in a \mathcal{PN} are represented by transitions (i.e. events that may occur) and places (i.e. local states). They are connected with weighted arcs. The places represent pre- and/or post-conditions (described by the arcs) for the transitions. They can contain a discrete number of marks called tokens. The

2. BACKGROUND AND RELATED WORK

initial distribution of tokens (the initial marking) of all places represents the initial state of a system (network, model). In systems biology, places and transitions often represent species and biochemical reactions (or transport steps), respectively. The number (concentrations) of species is represented by tokens, while stoichiometries are represented with the help of weighted arcs. The following formal notations are used throughout this thesis.

Notation 1. *Formal notations:*

$m(p)$ - the current marking of a place p ;

$\bullet t$ - set of pre-places of a transition t ;

t^\bullet - set of post-places of a transition t ;

$\bullet p$ - set of pre-transitions of a place p ;

p^\bullet - set of post-transitions of a place p ;

Definition 1 (Petri net). *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.
- $f : A \rightarrow \mathbb{N}$ is a function that assigns a positive integer number (weight) to each arc $a \in A$.
- $m_0 : P \rightarrow \mathbb{N}_0$, is a function that assigns a non-negative integer number to each place as the initial marking.

The dynamic behaviour of a \mathcal{PN} is characterized by the enabling and firing of transitions. A transition t is enabled if the marking of each of its pre-places $\bullet t$ is at least equal to the weight of the corresponding arc (the arc connecting the place with the transition). If a transition is enabled, it can fire and change the markings on its pre-places $\bullet t$ and post-places t^\bullet by subtracting and adding tokens, respectively. The amount of removed or added tokens depends on the arcs weights connecting the transition t with its pre- and post-places. By firing of a transition a new state of the system is set (achieved). The enabling and firing of a transition is determined by the following two definitions.

Definition 2 (Transition enabling). *A transition $t \in T$ is enabled in a marking m , denoted by $m[t]$, if and only if the following condition is satisfied:*

- $m(p) \geq f(p, t), \forall p \in \bullet t.$

Definition 3 (Transition firing). *A transition $t \in T$ enabled in a marking m , denoted by $m[t]$, can fire and reach a new marking m' denoted by $m[t]m'$, with:*

- $m'(p) = m(p) - f(p, t), \forall p \in \bullet t.$
- $m'(p) = m(p) + f(t, p), \forall p \in t^\bullet.$

Example 2.1. Figure 2.1 builds on the SIR model [HLM14], a simple compartmental model used to model an epidemic process. The model consists of three compartments (susceptible population, infected population and recovered population) and two events (Infect and Recover), which are represented by places and transitions, respectively. In the example as considered here, an epidemic spreads separately in two populations, *A* and *B*. An individual from a given population may belong to one of three sub-populations (states): *Susceptible*, *Infected* and *Recovered*. Due to the occurrence of the infection events (represented by the transition *Infect*) an individual becomes infected and is shifted from the *SusceptiblePopulation* to the *Infected* sub-population. The infection events may occur as long as there are individuals in the *Susceptible* and *Infected* sub-populations. Similarly, due to the occurrence of the second event (represented by the transitions *Recover*) an individual recovers and is shifted from the *Infected* to the *Recovered* sub-population. The recovery events occur as long as there is an individual in the *Infected* sub-population.

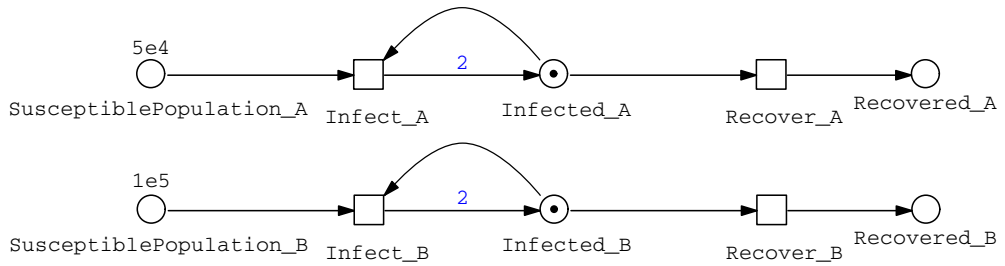


Figure 2.1: SIR model, as \mathcal{PN} ; where \square - transition, \circ - place, \longrightarrow - directed arc.

2.2 Extended Petri Nets

Extended Petri net \mathcal{XPN} is an extension of \mathcal{PN} , where the set of arcs A is extended by four special arcs:

1. $\text{---}\bullet$ *read* arc - does not change the marking after firing of a transition and enables it, if the amount of tokens is greater or equal to the arc weight;

2. BACKGROUND AND RELATED WORK

2. $\text{---}\circ$ *inhibitor arc* - does not change the marking after firing of a transition and enables it, if the amount of tokens is less than the arc weight;
3. $\text{---}\bullet\bullet$ *equal arc* - does not change the marking after firing of a transition and enables it, if the amount of tokens is equal to the arc weight (the equal arc can be replaced by a combination of the two arcs, read and inhibitor);
4. $\text{---}\blacktriangleright$ *reset arc* - changes the marking on the tested place by removing all tokens upon transition firing and does not add any additional restrictions to enable the transition.

All four arcs connect a place with a transition.

Definition 4 (Extended Petri net). *An extended 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.
- $P \cap T = \emptyset$.
- $A = A_d \cup A_r \cup A_i \cup A_e \cup A_z$ is a finite set of arcs defined as the union of:
 1. $A_d \subseteq (P \times T) \cup (T \times P)$ is a finite set of directed arcs,
 2. $A_r \subseteq (P \times T)$ is a finite set of read arcs,
 3. $A_i \subseteq (P \times T)$ is a finite set of inhibitor arcs,
 4. $A_e \subseteq (P \times T)$ is a finite set of equal arcs,
 5. $A_z \subseteq (P \times T)$ is a finite set of reset arcs.
- $f : A \rightarrow \mathbb{N}$ is a function that assigns a positive integer number to each arc $a \in A$ depending on the arc type. If an arc is not explicitly weighted, then the weight of one is assigned:
$$f : \begin{cases} A_d \rightarrow \mathbb{N}, \\ A_r \rightarrow \mathbb{N}, \\ A_i \rightarrow \mathbb{N}, \\ A_e \rightarrow \mathbb{N}, \\ A_z \rightarrow \{1\}. \end{cases}$$
- $m_0 : P \rightarrow \mathbb{N}_0$ is a function that assigns a non-negative integer number to each place as the initial marking.

An arc of \mathcal{XPN} can be self-modifying, if its weight depends on the marking of a place [Val78]. This can be restricted to a transition's pre-place to make the dependencies clearly visible in the net. In the end, it is no restriction at all, because one can overcome it by adding a read arc from the needed place with an arc weight set to this place [Roh17].

Definition 5 (Self-modifying arcs). f is a function that assigns a marking-dependent arc weight to each arc $a \in A$ depending on the arc type:

$$f : \begin{cases} A_d \rightarrow D, \\ A_r \rightarrow D, \\ A_i \rightarrow D, \\ A_e \rightarrow D, \\ A_z \rightarrow \{1\}. \end{cases}$$

where D is defined as follows: $D = \{d_t | d_t : \mathbb{N}_0^{|\bullet t|} \rightarrow \mathbb{N}_0, t \in T\}$ is the set of all marking-dependent arc weight functions, and $f(t) = d_t, \forall t \in T$.

Example 2.2. Figure 2.2 presents a part of the model in Figure 2.1 as \mathcal{XPN} . Its purpose is to demonstrate the use of special arcs. In this model an epidemic may start if the state of a susceptible population (place *SusceptiblePopulation*) is at least one and there also exists at least one infected specimen (the read arc connecting the transition *Infect* with the place *Infected*). The process of population recovery (*Recover*) can continue only up to the limited number *LIMIT*. After exhausting the limit, the recovery process is blocked by the inhibitor arc connecting the transition *Recover* with the place *Recovered*. If the limit *LIMIT_2* is reached (the equal arc connecting the transition *NewEpidemic* with the place *Recovered*), a new epidemic may start by resetting the model to its initial state (with the help of reset arcs).

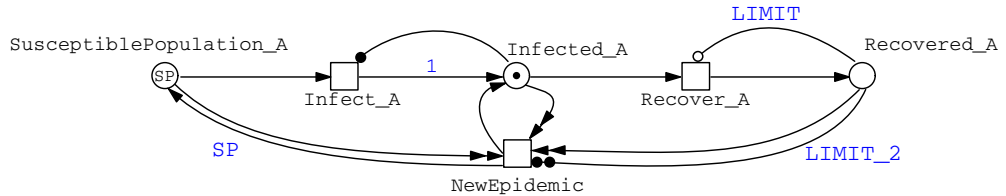


Figure 2.2: SIR model, as \mathcal{XPN} ; where SP - constant, which represents the initial state of a susceptible population, LIMIT - weight of the inhibitor arc, LIMIT_2 - weight of the equal arc, \square - transition, \circ - place, \rightarrow - directed arc, $\rightarrow\rightarrow$ - reset arc, $\text{---}\bullet$ - read arc, $\text{---}\bullet\bullet$ - equal arc, $\text{---}\circ$ - inhibitor arc.

2.3 Quantitative Petri Nets

Quantitative Petri net is an extension of Petri net, where each transition is connected with an arbitrary mathematical function (rate function) that defines the firing rate of a transition. The rate usually depends on the system state, may involve kinetic parameters (constants) and often follows specific kinetic patterns (e.g. mass/action kinetics). To prevent that net structure and rate functions diverge, a constraint rule has been adopted. The rule states that only pre-conditions (pre-places) of a given transition can be used as variables.

2. BACKGROUND AND RELATED WORK

Definition 6 (Quantitative Petri net). *A 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 Petri net (Definition 1).
- $v : T \rightarrow H$ is a function which assigns a firing rate function h_t to each transition $t \in T$, whereby $H = \{h_t | h_t : \mathbb{R}_0^{+|\bullet t|} \rightarrow \mathbb{R}_0^+, t \in T\}$ is the set of all marking dependent firing rate functions, and $v(t) = h_t, \forall t \in T$.

The interpretation of the firing rate determines three types of quantitative Petri nets [Her13]: stochastic, continuous and hybrid.

Stochastic Petri net - \mathcal{SPN} . In a \mathcal{SPN} model, a place contains a discrete number of tokens (markings) (discrete place). Markings represent a system state whereas transitions are associated with events of a Markov process. The probability of an event occurrence is equal to a transition firing rate (stochastic transition). This allows representing \mathcal{SPN} behaviour as a continuous time Markov chains (\mathcal{CTMC}) and apply stochastic simulation algorithms. Furthermore, a time delay can be assigned to transition firing and, depending on the interpretation, it can be a deterministic delay or stochastic delay where the delay is randomly exponentially distributed. This leads to extended stochastic Petri net \mathcal{XSPN} as defined in [MRH12].

Definition 7 (Extended stochastic Petri net). *A Stochastic Petri net is a 6-tuple $N = \langle P, T, A, f, v, m_0 \rangle$, where:*

- $\langle P, T, A, f, v, m_0 \rangle$ is a \mathcal{XPN} (Definition 4).
- $T = T_{stoch} \cup T_{immediate} \cup T_{timed} \cup T_{scheduled}$ is a finite set of transitions defined as the union of:
 1. T_{stoch} is a finite set of stochastic transitions, that fire stochastically after an exponentially distributed waiting time,
 2. $T_{immediate}$ is a finite set of immediate transitions, that all fire with waiting time zero,
 3. T_{timed} is a finite set of deterministically delayed transitions, that fire after a deterministic time delay
 4. $T_{scheduled}$ is a finite set of scheduled transitions, that fire at predefined time points.
- v is a set of functions $v = \{g, w, d, c\}$ where :
 1. $g : T_{stoch} \rightarrow H_s$ is a function that assigns a stochastic hazard function h_{st} to each transition $t \in T_{stoch}$, whereby $H_s = \{h_{st} | h_{st} : \mathbb{R}_0^{|\bullet t|} \rightarrow \mathbb{R}_0^+, t \in T_{stoch}\}$ is the set of all stochastic hazard functions, and $g(t) = h_{st}, \forall t \in T_{stoch}$,

2. $w : T_{immediate} \rightarrow H_w$ is a function that assigns a weight function h_w to each immediate transition $t \in T_{immediate}$, such that $H_w = \{h_{w_t} | h_{w_t} : \mathbb{R}_0^{|\bullet t|} \rightarrow \mathbb{R}_0^+, t \in T_{im}\}$ is the set of all weight functions, and $w(t) = h_{w_t}, \forall t \in T_{immediate}$,
3. $d : T_{timed} \cup T_{scheduled} \rightarrow \mathbb{R}_0^+$, is a function that assigns a constant waiting time to each deterministically delayed transition.
4. $c : T_{scheduled} \rightarrow \mathbb{R}_0^+$, is a function that 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.

Continuous Petri net - CPN. In a CPN model, a place contains a continuous number of markings (continuous place) whereas a transition rate is associated with the continuous change of markings of the pre- and/or post-conditions (places). This allows CPN to be represented as ordinary differential equations (ODEs) and carrying out a numerical integration (finding numerical approximations to the solutions of ODEs).

Hybrid Petri net - HPN. A HPN model is a fusion of the SPN and CPN modelling approach. Within the same model, different types (discrete and continuous) of places and transitions can exist side by side with some restrictions on the connection between them [HH12]. A simulation has to apply a stochastic or continuous algorithm depending on the type of subnet. The continuous subnet usually represents events that occur frequently and the stochastic subnet represents less frequent events. In relation to this, the continuous simulation will be applied most of the time and occasionally, depending on the schedule of stochastic event occurrence, the stochastic one [HH18].

Example 2.3. Figure 2.3 presents the quantitative extension of the model in Figure 2.1 with additional rate functions that define transition firing rates. The rate functions control how quickly a disease spreads. A rate is expressed by mass-action kinetics pattern: $MassAction(k)$; where k is a kinetic parameter. For more information on the mass-action kinetics pattern, see Subsection 2.7.1 of this thesis.

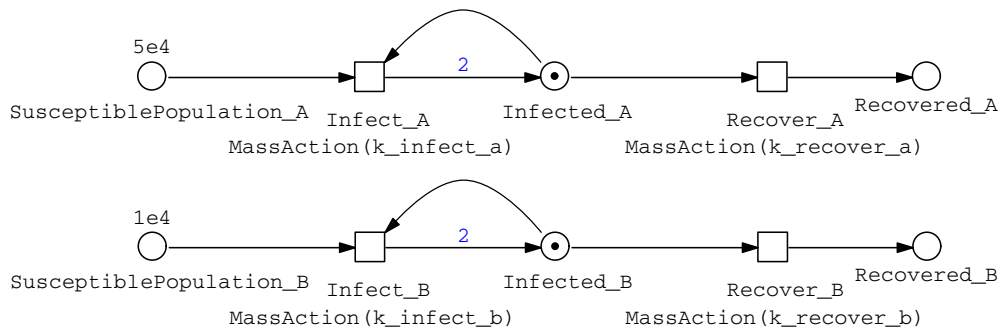


Figure 2.3: SIR model as SPN.

2. BACKGROUND AND RELATED WORK

Example 2.4. Figure 2.4 presents the extended quantitative model as stochastic \mathcal{XPN} . The model is a combined variation of the SIR models in Figures 2.2 and 2.3, where rate functions control how quickly a disease spreads.

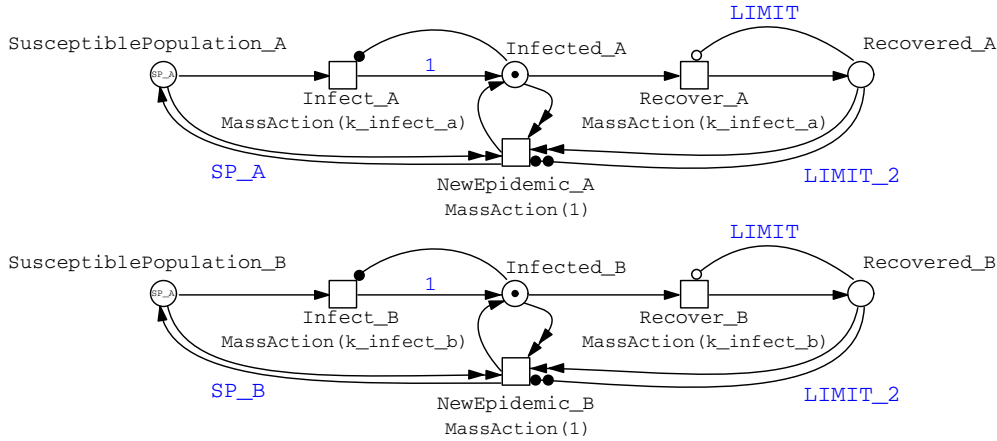


Figure 2.4: Variation of the SIR model as \mathcal{XSPN} ; where SP - constant which represents an initial state of a susceptible population, LIMIT - weight of the inhibitor arc, LIMIT_2 - weight of the equal arc, \square - transition, \circ - place, \rightarrow - directed arc, \Rightarrow - reset arc, \longrightarrow - read arc, \longrightarrow - equal arc, \longrightarrow - inhibitor arc.

2.4 Coloured Petri Nets

Coloured Petri net \mathcal{PN}^c [GL79, Kur81] is an extension that preserves properties of standard \mathcal{PN} and combines the power of graphical modelling with the expressiveness of a programming language. This combination is the main advantage of \mathcal{PN}^c and permits the modelling of complex systems in a compact and structured way. Like \mathcal{PN} , it consists of places and transactions connected by arcs. The main addition of \mathcal{PN}^c is enhanced by discrete data types. In a programming language, a data type is a set of values that obey some attributes [CW85] from which an expression (e.g. user-defined functions) may take its values. In \mathcal{PN}^c , discrete data types are represented by colour sets. The main basic data types are: *integer*, *Boolean*, *string* and *enumeration*. They can be used to define expressions (colour expressions) that are used to define multisets, initial markings, arcs inscriptions, and guards. A colour set is assigned to each place, which may contain distinguishable tokens. As a place can contain multiple numbers of tokens of the same colour, the best way to describe them is a multiset. The multiset is a colour expression over the colour set assigned to the place. A guard is associated with each transition. The guard is a *Boolean* expression over constants, variables or functions. It enables an associated transition only if the expression evaluates to true. Additionally, an expression is assigned to each arc which defines a multiset over the colour set of the connected place [Liu12, LHG19]. In addition to improved readability, \mathcal{PN}^c allows to easily scale a model by use of coloured expressions and adding or removing colours from colour sets.

Definition 8 (Multiset). A multiset S_{MS} over S is a function $m : S \rightarrow \mathbb{N}_0$ that maps each element $s \in S$ onto a non-negative integer $m(s) \in \mathbb{N}_0$. It is denoted by a formal sum $m = \sum_{s \in S} m(s)'s$, where:

- S is a finite, non-empty set.

Definition 9 (Multiset operations). Let S be a finite, non-empty set, and $\forall m_1, m_2, m \in S_{MS}$. Addition (+), scalar multiplication (*), comparison (\leq), subtraction (-) and size $|m|$ are defined as follows:

1. $(m_1 + m_2)(s) = m_1(s) + m_2(s), \forall s \in S$.
2. $(n * m)(s) = n * m(s) \forall n \in \mathbb{N}_0, \forall s \in S$.
3. $m_1 \leq m_2 \Leftrightarrow m_1(s) \leq m_2(s), \forall s \in S$.
4. $(m_2 - m_1)(s) = m_2(s) - m_1(s), \forall s \in S$ and $m_1 \leq m_2$.
5. $|m| = \sum_{s \in S} m(s)$.

Example 2.5. Let a color set be $S = \{a, b, c\}$, then $m = 1'a+2'b+4'c$ is a multiset over S , which contains 1 occurrence of element a , 2 occurrences of element b and 4 occurrences of element c , i.e. $m(a) = 1, m(b) = 2$ and $m(c) = 4$.

Definition 10 (Coloured Petri net). A coloured Petri net is an 8-tuple $N = \langle P, T, A, \sum, C, g, f, m_0 \rangle$, 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.
- \sum is a finite, non-empty set of colour sets.
- $C : P \rightarrow \sum$ is a colour function that assigns to each place $p \in P$ a colour set $C(p) \in \sum$.
- $g : T \rightarrow EXP$ is a guard function that assigns to each transition $t \in T$ a guard expression of the Boolean type.
- $f : A \rightarrow EXP$ is an arc function that assigns to each arc $a \in A$ an arc expression of a multiset type $C(p)_{MS}$, where p is the place 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 $C(p)_{MS}$.

2. BACKGROUND AND RELATED WORK

After unfolding a \mathcal{PN}^C $N = \langle P, T, A, \sum, C, g, f, m_0 \rangle$, an uncoloured place instance $p(c)$ represents one colour $c \in C(p)$ from colour set $C(p)$ associated with a coloured place $p \in P$. The set of all instances of a place $p(c)$ of p is defined as $I_P(p)$. The joined set of all $I_P(p)$ of all places $p \in P$ is defined as I_P .

Definition 11 (Place instance). *A place instance $p(c)$ is a pair (p, c) with $p \in P$ and $c \in C(p)$.*

Each expression (a guard of a transition and expressions on its adjacent arcs) associated with a transition needs to be evaluated. If the expressions involve a set of variables, then for each variable $Var(t)$ associated with transition $t \in T$ a binding [JKW07] must be applied. Through the binding b to each variable $v \in Var(t)$ a value of a suitable data type is assigned. After unfolding, each uncoloured transition instance $t(b)$ represents one binding $b \in B(t)$ from the transition binding set $B(t)$ (which represents all bindings of given transition). The set of all transition instances $t(b)$ of t is defined as $I_T(t)$. The joined set of all $I_T(t)$ of all transitions $t \in T$ is defined as I_T .

Definition 12 (Transition instance). *A transition instance $t(b)$ is a pair (t, b) with $t \in T$ and $b \in B(t)$.*

A transition instance $t(b)$ is enabled if the guard $g(t)$ and the adjacent arc $f(p, t)$ expressions evaluates to true. The enabled transition instance $t(b)$ can fire only if pre-places have enough tokens of given colours that are denoted by arc expressions after evaluation for a given binding.

Definition 13 (Transition instance enabling). *A transition instance $t(b) \in I_T$ is enabled in a marking m , denoted by $m[t(b)]$, if and only if the following conditions are satisfied:*

1. $g(t) \langle b \rangle = true$,
2. $m(p) \geq f(p, t) \langle b \rangle, \forall p \in \bullet t$.

Definition 14 (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 $m[t(b)]m'$, with*

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

Upon firing, tokens are removed from all pre-place instances $p(c)$ and added to all post-place instances $p(c)$ denoted by an arc expression .

Example 2.6. Figure 2.5 presents the coloured version of the SIR model in Figure 2.1. The colour set *Population* is of the enumeration type, with two defined colours *A* and *B*. It represents two populations and is assigned to each place. As a place may have several tokens of different colours, the place *SusceptiblePopulation* is initialized with the multiset expression $5e4'A + 1e5'B$ over the colour set *Population*. In this case, the place contains $5e4$ tokens of colour *A* and $1e5$ tokens of colour *B* making in total $1.5e5$ tokens. Similarly, the place *Infected* is initialized with one token of each colour from *Population*, which describes the expression $1'all$.

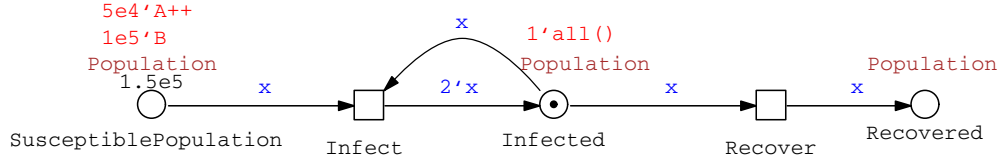


Figure 2.5: SIR model, as coloured \mathcal{PN} . The colour definitions are as follows: colour set: $enum\ Population = \{A, B\}$; variable: x of type $Population$.

2.5 Coloured Quantitative Petri Nets

An extension of \mathcal{PN}^C is the coloured quantitative Petri net. Similar like for the quantitative \mathcal{PN} , a rate function is assigned to each transition - it sets a delay (temporize transition) or a probability (immediate transition) of a transition firing. Depending on the interpretation, this allows to define coloured stochastic Petri nets (\mathcal{SPN}^C) and coloured continuous Petri nets (\mathcal{CPN}^C) [Liu12]. The fusion of \mathcal{SPN}^C and \mathcal{CPN}^C yields to coloured hybrid Petri nets (\mathcal{HPN}^C) [HLR+18].

Definition 15 (Coloured quantitative Petri net). *A coloured quantitative Petri net is a 9-tuple $N = \langle P, T, A, \sum, C, g, f, v, m_0 \rangle$, where:*

- $\langle P, T, A, \sum, C, g, f, m_0 \rangle$ 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 $v(t(b)) = h_{t(b)}$, $\forall t(b) \forall t \in T$.

The rate function can be colour-dependent and therefore rate functions can vary between transition instances.

Example 2.7. Figure 2.6 presents the quantitative extension of the model in Figure 2.5 with additional colour-dependent rate functions e.g. $[x=A] : MassAction(k)$; where the type of variable x is the colour set $Population$, k is the kinetic parameter and the equation $x = A$ determines the dependence on the colour. Rate functions of both transitions are colour-dependent and follow specific kinetic patterns, mass-action. The parameters of the mass-action rate reactions are parametrized by kinetic parameters (crisp values) which depend on the colour value, e.g. $[x=A] MassAction(k_{infect_a}) ++ [x=B] MassAction(k_{infect_b})$. If the colour value is A , then $MassAction$ is applied with the kinetic parameter k_{infect_a} ; similarly for the value B . In the model the standard arcs connect transitions with pre- and post-places.

Example 2.8. The model in Figure 2.7 consists of continuous nodes, and it is derived by a straightforward conversion of the model in Figure 2.6.

Example 2.9. Figure 2.8 presents a hybrid version of the model in Figure 2.6. For the purpose of this example, the model is statically partitioned and consists of two clusters

2. BACKGROUND AND RELATED WORK

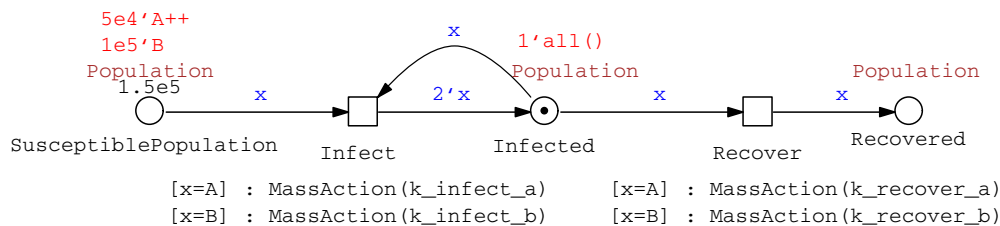


Figure 2.6: SIR model, as coloured \mathcal{SPN} ; where \square - stochastic transition, \circ - discrete place, \rightarrow - directed arc. A more flexible solution to specify colour-dependent rate functions can be found in Section 4.8.

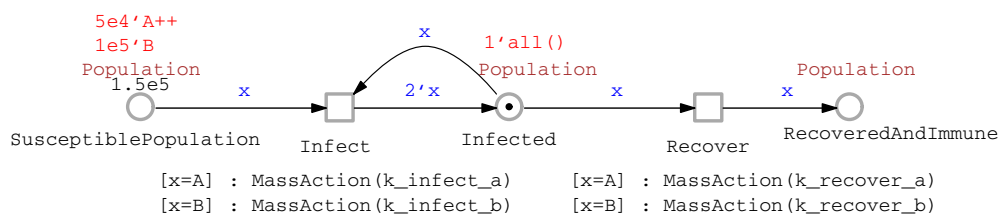


Figure 2.7: SIR model, as coloured \mathcal{CPN} ; where \square - continuous transition, \circ - continuous place, \rightarrow - directed arc.

(subnets), continuous and stochastic. These two parts model the infection and recovery process, respectively.

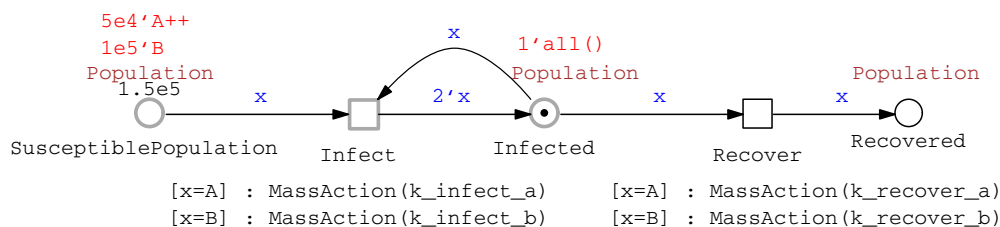


Figure 2.8: SIR model, as coloured \mathcal{HPN} ; where \square - stochastic transition, \square - continuous transition, \circ - discrete place, \circ - continuous place, \rightarrow - directed arc.

2.6 Unfolding

Currently, unfolding a \mathcal{PN}^c (with finite discrete colour sets) to its corresponding uncoloured \mathcal{PN} is a necessary step to apply analysis and simulation techniques as most of them require standard Petri nets. At this step each colour of a place and each binding of a transition is unfolded to a place and transition instance, respectively.

2.6.1 Equivalent Standard Petri Nets

The equivalent standard \mathcal{PN} of a \mathcal{PN}^c with finite colour sets is defined by the Definition 16 [Jen92].

Definition 16 (Unfolded Petri net). *Let $N = \langle P, T, A, \sum, C, g, f, m_0 \rangle$ be a coloured Petri net, its unfolded Petri net $N^* = \langle P^*, T^*, A^*, f^*, m_0^* \rangle$ is defined by:*

1. $P^* = I_P$.
2. $T^* = I_T$.
3. $A^* = \{(p(c), t(b)) \in P^* \times T^* \mid (f(p, t)\langle b \rangle)\langle c \rangle > 0\} \cup \{(t(b), p(c)) \in T^* \times P^* \mid (f(t, p)\langle b \rangle)\langle c \rangle > 0\}$.
4. $\forall (p(c), t(b)) \in A^* : f^*(p(c), t(b)) = (f(p, t)\langle b \rangle)\langle c \rangle,$
 $\forall (t(b), p(c)) \in A^* : f^*(t(b), p(c)) = (f(t, p)\langle b \rangle)\langle c \rangle.$
5. $\forall p(c) \in P^* : m_0^*(p(c)) = m_0(p)\langle c \rangle.$

1. The places of the Petri net N^* correspond to the place instance $p(c) \in I_P$ in the coloured Petri N . This means splitting of each coloured place $p \in P$ into as many uncoloured places $p^* \in P^*$ as there are colours $c \in C(p)$. This allows for distinguishing coloured tokens (the colours are lost after translation) as they are assigned to different places in the Petri net N^* .
2. The transitions of the Petri net N^* correspond to each binding (transition instance) $t(b) \in I_T$ in the coloured Petri N . This means splitting of each coloured transition $t \in T$ into as many uncoloured transitions $t^* \in T^*$ as there are bindings $b \in B(t)$.
3. An arc connecting $p(c)$ with $t(b)$ exists iff an occurrence of t with the binding b removes at least one coloured token c from p , i.e. $(f(p, t)\langle b \rangle)\langle c \rangle > 0$. Analogously, an arc connecting $t(b)$ with $p(c)$ exists iff an occurrence of t with the binding b adds at least one coloured token c to p , i.e. $(f(t, p)\langle b \rangle)\langle c \rangle > 0$.
4. The weight of the arc connecting $p(c)$ with $t(b)$ is the number of the c tokens $(f(p, t)\langle b \rangle)\langle c \rangle$ which an occurrence of t with the binding b removes from p . Analogously, the weight of the arc connecting $t(b)$ with $p(c)$ is the number of the c tokens $(f(t, p)\langle b \rangle)\langle c \rangle$, which an occurrence of t with the binding b adds to p .
5. The number of initial tokens of place $p(c) \in P^*$ is equal to the number of the c tokens $m_0(p)\langle c \rangle$ of place $p \in P$.

The above definition does not include \mathcal{PN}^C with special arcs and any time information. If \mathcal{PN}^C contains special arcs, then the unfolded counterparts need to be of the same types. Likewise, if \mathcal{PN}^C contains time information, then it must be added to the unfolded transitions.

2. BACKGROUND AND RELATED WORK

2.6.2 Unfolding Algorithm

In [Liu12] an unfolding algorithm for \mathcal{PN}^c is proposed. To be efficient, the Algorithm 1 may adopt the following optimization techniques:

- optimization techniques for transition instance computation: constraint satisfaction approach, partial binding - partial test principle, merging identical patterns and the fewer-colours-first policy,
- removal of false guarded transitions; during binding process (not at the end of the unfolding), transition instances are removed if guards are evaluated to be false,
- removal of isolated places or transitions; during the binding process, isolated places and transitions are removed as they do not contribute to the behaviour of the net.

Valid bindings are computed for each transition t in the net N (Line 2). If the variable set $V(t)$ of the transition t is not empty and simultaneously the binding set B is empty, then t is isolated and can be immediately excluded from the unfolded net N^* (Lines 3-5). Then for each binding $b \in B$:

- a transition $t^*(b) \in T^*$ is instantiated with the assigned value $t(b) \in T$ (Line 7);
- for each pre-arc $A(p, t)$ of t , its expression is evaluated with regard to binding and then for each colour c in the evaluated expression $(f(p, t)\langle b \rangle)$:
 - a place $p^*(c) \in P^*$ is instantiated with the assigned value $p(c) \in P$ (Line 10);
 - the number of c tokens on place p is assigned to the place $p^*(c)$ (Line 11);
 - an arc expression $(f^*(p^*(c), t^*(b)))$ is instantiated with the number of c tokens $(f(p, t)\langle b \rangle)\langle c \rangle$ in $(f(p, t)\langle b \rangle)$ (Line 12);
 - an arc $(p(c), t(b))$ is added to the net N^* (Line 13);
- the post-arcs are created in the same way as the pre-arcs (Lines 16-23);

This algorithm does not consider \mathcal{PN}^c with special arcs and time information, but the principles stay the same - only special arcs and time information need to be added to the algorithm. It is a basic algorithm that says nothing about the implementation. The description of an efficient implementation that builds on Interval Decision Diagrams (*IDD*) can be found in Section 4.8.

Algorithm 1: Unfolding a coloured Petri net [Liu12].

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

- 1 **for each** *transition* $t \in T$ **do**
- 2 $B = \text{ComputeBindings}(t)$;
- 3 **if** $V(t)$ *is not empty* and B *is empty* **then**
- 4 t *is isolated*;
- 5 **end**
- 6 **for each** *binding* $b \in B$ **do**
- 7 $t^*(b) \leftarrow t(b)$;
- 8 **for each** *pre-arc* (p, t) *of* t **do**
- 9 **for each** *colour* c *in* $(f(p, t)\langle b \rangle)$ **do**
- 10 $p^*(c) \leftarrow p(c)$;
- 11 $m_0^*(p^*(c)) \leftarrow m_0(p)\langle c \rangle$;
- 12 $f^*(p^*(c), t^*(b)) \leftarrow (f(p, t)\langle b \rangle)\langle c \rangle$;
- 13 $(p^*(c), t^*(b)) \leftarrow (p(c), t(b))$;
- 14 **end**
- 15 **end**
- 16 **for each** *post-arc* (t, p) *of* t **do**
- 17 **for each** *colour* c *in* $(f(t, p)\langle b \rangle)$ **do**
- 18 $p^*(c) \leftarrow p(c)$;
- 19 $m_0^*(p^*(c)) \leftarrow m_0(p)\langle c \rangle$;
- 20 $f^*(t^*(b), p^*(c)) \leftarrow (f(t, p)\langle b \rangle)\langle c \rangle$;
- 21 $(t^*(b), p^*(c)) \leftarrow (t(b), p(c))$;
- 22 **end**
- 23 **end**
- 24 **end**
- 25 **end**

2. BACKGROUND AND RELATED WORK

2.7 Simulation

Modelling a system is the first step to understand it. The following step is often a simulation, which allows for the analysis of dynamic behaviours of a modelled system. It is an essential tool for studying biochemical systems. As presented in [Her13, Roh17], simulation types are divided into three main classes: deterministic, stochastic and hybrid. What kind of simulation class will be applied, depends on the model and the properties of interest.

2.7.1 Mass-Action Kinetics

In biochemical reaction networks, the transition firing rates usually follow specific kinetic patterns, e.g. mass-action kinetics. The mass-action law explains the relation between the rates of reactions and the concentrations of reactants in the reaction network.

Reaction Networks. A chemical reaction network comprises a set of *species* (which consists of subsets of *reactants* and *products*) and *reactions*. In \mathcal{PN} , *species* are represented by places and *reactions* by transitions. This can be clarified by considering as example a simple reaction (2.1) and the corresponding \mathcal{SPN} in Figure 2.9, where the species $S_1, \dots, S_{|S|}$ appear in the reaction with at least one non-zero coefficient α_x or β_x .

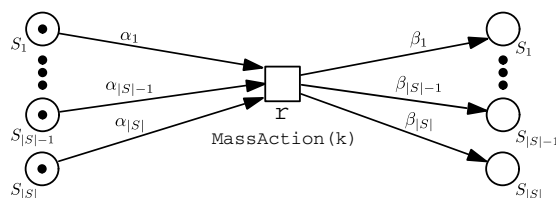
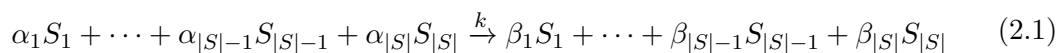


Figure 2.9: A simple reaction as \mathcal{SPN} .

The species of the reaction (2.1) are *reactants* (left-hand side of the *reaction* (denoted by directed arrows)) and *products* (right side). The *rate constant* of the reaction is denoted by k . The reaction can concisely be represented by the equivalent equation (2.2)



where α_x and β_x are the stoichiometric coefficients of the species S_x (that describe how many molecules of the species S_x react in each occurrence of the reaction), and $|S|$ is

the number of species which is equivalent to the length of the species vector in the matrix-vector notation. Using the matrix-vector notation, the equation (2.2) can take the form

$$\alpha S \xrightarrow{k} \beta S \quad (2.3)$$

where $S = [S_1, S_2, \dots, S_{|S|}]^T$, $\alpha = [\alpha_1, \alpha_2, \dots, \alpha_{|S|}]$ and $\beta = [\beta_1, \beta_2, \dots, \beta_{|S|}]$.

Similarly, for a reversible reaction (2.4)

$$\alpha_1 S_1 + \dots + \alpha_{|S|-1} S_{|S|-1} + \alpha_{|S|} S_{|S|} \xrightleftharpoons[k_2]{k_1} \beta_1 S_1 + \dots + \beta_{|S|-1} S_{|S|-1} + \beta_{|S|} S_{|S|} \quad (2.4)$$

which represents forward and backward reactions that can be represented in an irreversible form (2.5)

$$\begin{aligned} \alpha_1 S_1 + \dots + \alpha_{|S|-1} S_{|S|-1} + \alpha_{|S|} S_{|S|} &\xrightarrow{k_1} \beta_1 S_1 + \dots + \beta_{|S|-1} S_{|S|-1} + \beta_{|S|} S_{|S|} \\ \alpha_1 S_1 + \dots + \alpha_{|S|-1} S_{|S|-1} + \alpha_{|S|} S_{|S|} &\xrightarrow{k_2} \beta_1 S_1 + \dots + \beta_{|S|-1} S_{|S|-1} + \beta_{|S|} S_{|S|} \end{aligned} \quad (2.5)$$

where reaction rate constants are denoted by k_1 and k_2 . This allows to derive the reaction network (equation (2.6)) where every reaction in the reaction network is represented as irreversible.

$$\sum_{x=1}^{|S|} \alpha_x S_x \xrightarrow{k_j} \sum_{x=1}^{|S|} \beta_x S_x, \quad j = 1, \dots, M; \quad (2.6)$$

where M is the number of reactions

Mass-Action Law. The dynamics of this reaction network can be derived by mass-action law, which states that for an elementary reaction, that is, a reaction in which all the stoichiometric coefficients of the reactants are one, the rate of reaction is proportional to the product of the concentrations of the reactants [SFH99].

For the i -th species in a single irreversible reaction (a special case of a reaction network where number of reactions is equal one), the rate of a reaction is given by the equation

$$\frac{d[S_i]}{d\tau} = (\beta_i - \alpha_i) k \prod_{x=1}^{|S|} \prod_1^{\alpha_x} [S_x] = (\beta_i - \alpha_i) k \prod_{x=1}^{|S|} [S_x]^{\alpha_x} \quad (2.7)$$

Consequently, for the i -th species in a reaction network the equation has the form

2. BACKGROUND AND RELATED WORK

$$\frac{d[S_i]}{d\tau} = \sum_{j=1}^M (\beta_{ji} - \alpha_{ji}) k_j \prod_{x=1}^{|S|} [S_x]^{\alpha_{jx}} \quad (2.8)$$

where $(\beta_{ij} - \alpha_{ij})$ is a concentration of the species S_i that changes during the occurrence of the j -th reaction. The equation (2.2) can be expressed in the matrix-vector notation

$$\frac{d[S_i]}{d\tau} = (\beta - \alpha)^T k [S]^\alpha \quad (2.9)$$

where $\alpha = [\alpha_{M|S|}]$ and $\beta = [\beta_{M|S|}]$ are matrices of coefficients,

$$\alpha = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,|S|} \\ \alpha_{2,1} & \alpha_{2,2} & \cdots & \alpha_{2,|S|} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{M,1} & \alpha_{M,2} & \cdots & \alpha_{M,|S|} \end{pmatrix}; \quad \beta = \begin{pmatrix} \beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,|S|} \\ \beta_{2,1} & \beta_{2,2} & \cdots & \beta_{2,|S|} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{M,1} & \beta_{M,2} & \cdots & \beta_{M,|S|} \end{pmatrix}; \quad (2.10)$$

$k = \text{diag}(k_1, k_2, \dots, k_M)$ is the diagonal matrix of kinetic parameters,

$$k = \text{diag}(k_1, k_2, \dots, k_M) = \begin{pmatrix} k_1 & 0 & \cdots & 0 \\ 0 & k_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & k_M \end{pmatrix}; \quad (2.11)$$

$S = [S_1, S_2, \dots, S_{|S|}]^T$ the vector of species and S^α is the vector-matrix exponentiation [Mei19]. It is an operation that maps S and α to its vector-matrix power S^α defined as

$$S^\alpha = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_{|S|} \end{pmatrix}^\alpha = \begin{pmatrix} S_1^{\alpha_{1,1}} S_2^{\alpha_{1,2}} \cdots S_{|S|}^{\alpha_{1,|S|}} \\ S_1^{\alpha_{2,1}} S_2^{\alpha_{2,2}} \cdots S_{|S|}^{\alpha_{2,|S|}} \\ \vdots \\ S_1^{\alpha_{M,1}} S_2^{\alpha_{M,2}} \cdots S_{|S|}^{\alpha_{M,|S|}} \end{pmatrix}; \quad (2.12)$$

and takes the form of a column vector with $|S|$ entries.

The expression

$$k \prod_{x=1}^{|S|} [S_x]^{\alpha_x} \quad (2.13)$$

also refers to the pattern $MassAction(k)$ which is frequently used in the models of this thesis.

2.7.2 Stochastic Simulation

The stochastic simulation [Gil76, Gil77] fits well for biochemical systems as they are discrete and stochastic in nature. The stochasticity results from the fact that it is impossible to predict when a next reaction will occur. It can be described by a chemical master equation as presented in [Gil76]. It is valid in all cases as long as it performs a single run. For multiple runs, simulation traces are averaged and, as a result, some events can be hidden (lost by averaging). In this case, it is comparable to a deterministic simulation, and it is valid in the same situations [OSW69, Kur72]. Moreover, it deals well with low concentration of species [MA99, Pah09] where the deterministic simulation fails.

The semantics behind \mathcal{SPN} is described by a \mathcal{CTMC} which represents a state space that can be infinite. Instead of computing \mathcal{CTMC} directly, a simulation approximates it by generating different paths. A path is generated by repeatedly firing transitions starting from an initial marking m_0 [Roh17]. The main issue of a stochastic simulation is to identify which reaction would occur next and when, which is described by the reaction probability density function 2.14 [Gil76]. During simulation this issue leads to a race condition as the next system state is determined by the fastest reaction.

$$\begin{aligned}
 P(\tau, t_j | m) d\tau &\equiv \text{probability at given state } X(\tau) = m \\
 &\quad \text{that reaction } t_j \text{ will occur in} \\
 &\quad \text{the next time interval } [\tau, \tau + \delta\tau)
 \end{aligned}
 \tag{2.14}$$

Proposed by Gillespie [Gil76, Gil77], a method to construct the numerical realizations of species concentration is a Monte Carlo procedure for numerically generating paths through the \mathcal{CTMC} . Gillespie's Stochastic Simulation Algorithm (called SSA or Gillespie's algorithm) generates random walks through a \mathcal{CTMC} . It has many variants of implementations and optimizations, but basically each of them follows Algorithm 2. The description of some of them can be found in [Roh17, Her13].

Algorithm 2 returns a trace of stored system states (one possible path through a \mathcal{CTMC}) for a given time interval. Executing the loop of the algorithm (lines: 4-9), the current time τ is increased starting from the initial time τ_0 until it reaches the end time τ_{end} . A reliable insight into a system behaviour is only possible, if system states of several runs are examined. Each run starts from the same initial state, in which a random generator is initialized with a different/random seed (line: 1). This allows for various runs of a stochastic process. A system state at time point τ of each simulation run is recorded and the mean state at this point is given by Equation 2.15.

$$\bar{X}(\tau) = (1/N) \sum_{n=1}^N X(n, \tau)
 \tag{2.15}$$

2. BACKGROUND AND RELATED WORK

Algorithm 2: Basic stochastic simulation algorithm [Roh17].

Data: \mathcal{SPN} with initial state $X(\tau_0)$;
time interval $[\tau_0, \tau_{end}]$;
Result: trace of stored system states;

```

1  initRand(seed);
2  time  $\tau = \tau_0$ ;
3  state  $X(\tau) = X(\tau_0)$ ;           /* make initial state to current state */
4   $X(\tau) \rightarrow store$ ;         /* add  $X(\tau_0)$  to the trace */
5  while  $\tau < \tau_{end}$  do
6     $\delta\tau =$  determine duration until next firing by computing
7      rate function  $h$  of transitions depending on the current state  $X(\tau)$ ;
8     $\tau = \tau + \delta\tau$ ;           /* determine the next time point */
9     $t =$  select the next transition to fire depending on current state  $X(\tau)$ ;
10    $X(\tau) = t \rightarrow fire$ ;    /* compute new state  $X(\tau)$  by firing of  $t$  */
11    $X(\tau) \rightarrow store$ ;       /* add  $X(\tau)$  to trace */
12 end

```

As shown in Figure 2.10 (page 27), the number of runs N has an influence on the accuracy of a simulation as its recorded traces (results) are approximated by the mean state at point τ .

Example 2.10. Figure 2.11 (page 27) presents the results of the stochastic simulation of the model in Figure 2.4. The simulation traces 2.11.(a, c) and 2.11.(b, d) represent the states of the populations A and B over time, where the traces 2.11.(a, b) represent 1 simulation run, while 2.11.(c, d) are averaged over 100 simulation runs. The comparison of the simulation traces 2.11.(a) and 2.11.(c) gives an insight into how a detailed view of the system state can be lost due to averaging. A detailed view on a system state can be provided by single runs of a stochastic simulation. In Figure 2.11.(a) the reset event (*TransientImmunity*) is marked by values of *Susceptible* and *Recovered* population that are set to the initial state, while it is flattened in Figure 2.11.(c).

Example 2.11. A stochastic or deterministic simulation can be applied to any model type (\mathcal{SPN} , \mathcal{CPN} , \mathcal{HPN}) after a straightforward conversion of its nodes to the appropriate types determinates by the simulation. The model in Figure 2.6 (page 18) consists of stochastic nodes and their continuous counterpart (see Figure 2.7, page 18) is derived by a straightforward conversion. Figure 2.12 (page 28) presents the traces of the stochastic (2.12.(a, b)) and the deterministic (2.12.(c, d)) simulation. Notably, the resulting traces differ between Figure 2.12.(b) and 2.12.(d). This can be explained by decay of a disease when all specimen in an infected population recovered without spreading a disease (isolation) onto a susceptible population. This event cannot be clearly seen in the traces of the stochastic simulation, as they show the averaged results of 10 runs. It is similar in the case of the deterministic simulation. This event is superseded,

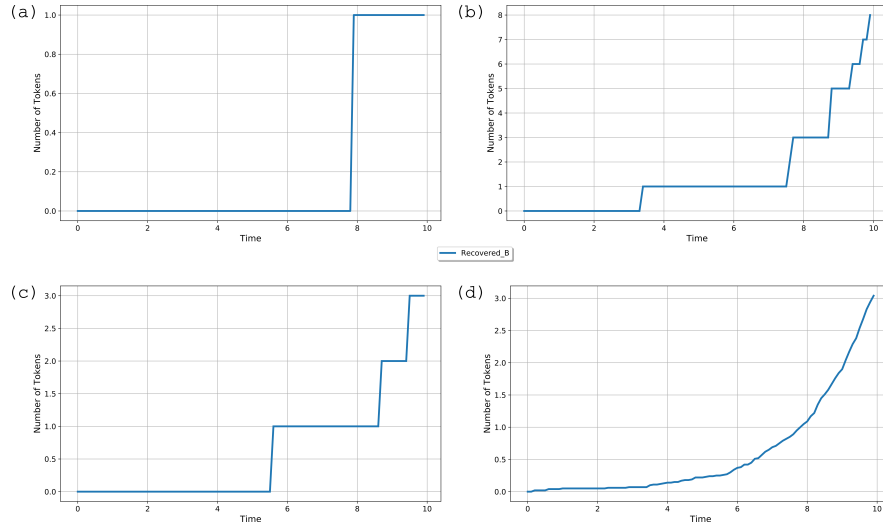


Figure 2.10: Influence of the number of simulation runs N on recorded and approximated stochastic simulation traces of the SPN model in Figure 2.3; number of tokens on place *Recovered_B* for single run (a - c) and an average of 100 runs (d).

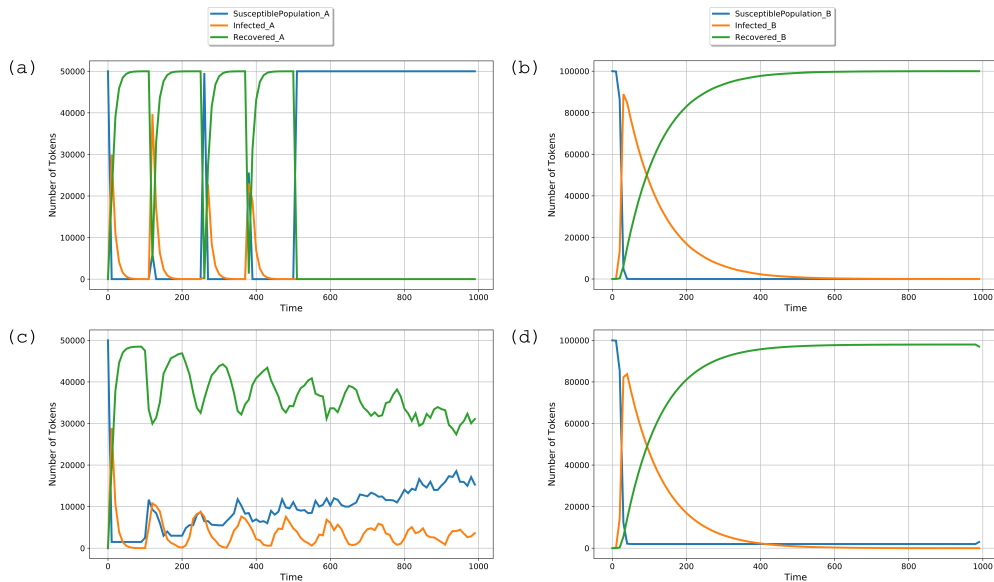


Figure 2.11: Simulation traces of the stochastic XPN model in Figure 2.4 represents results of the stochastic simulation; (a, b) - 1 run, (c, d) - an average of 100 runs; for the model configured as follows: $SP_A = 5e4$; $SP_B = 1e5$; $LIMIT_A = SP_A + 1$; $LIMIT_B = SP_B + 1$; $k_{infect.a} = 5.0e-5$; $k_{infect.b} = 5.0e-6$; $k_{recover.a} = 1.0e-1$; $k_{recover.b} = 1.0e-2$.

2. BACKGROUND AND RELATED WORK

since the traces are results of approximating the solutions of $ODEs$. To spot this event, it is necessary to apply a single run of a stochastic simulation. Figure 2.13 (page 28) presents the resulting traces of such simulations. It is easy to spot the decay of a disease (when all specimen in an infected population recovered) in the populations.

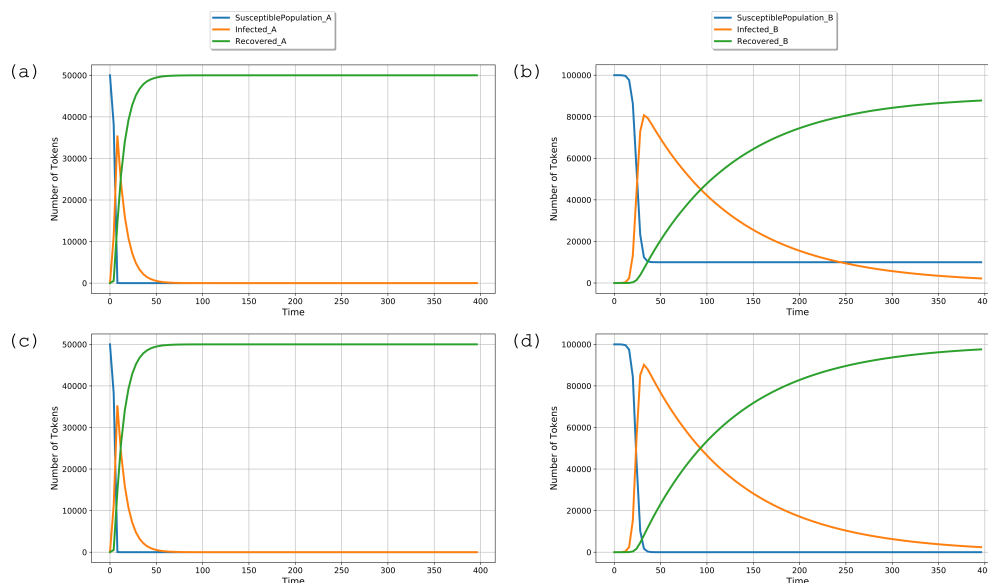


Figure 2.12: Comparison of the simulation traces of the models in Figure 2.6 and 2.7; average of 10 stochastic simulation runs (a, b) versus deterministic (continuous) simulation (c, d); where $SusceptiblePopulation$ is set to $5e4'A + 1e5'B$; $k_{infect_a} = 5.0e - 5$; $k_{infect_b} = 5.0e - 6$; $k_{recover_a} = 1.0e - 1$; $k_{recover_b} = 1.0e - 2$.

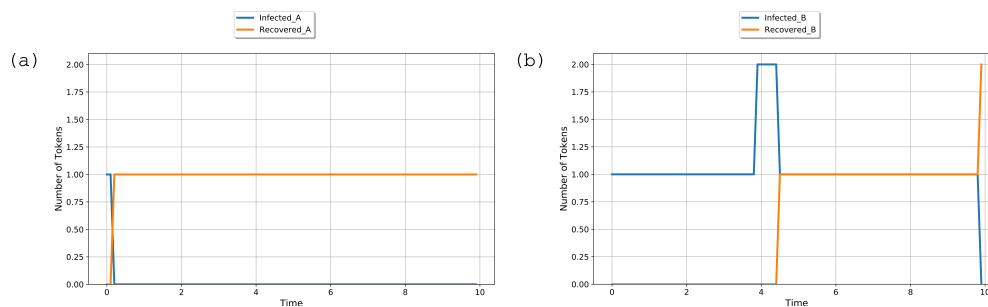


Figure 2.13: Decay events of a disease (when all specimen in an infected population have recovered) in the populations. This event cannot be clearly seen in the traces of the stochastic simulation if it is an averaged result of several runs (e.g. Figure 2.12(a,b)). Similarly, for the deterministic simulation, this event is superseded in resulting traces, since it is the result of approximating the solutions of $ODEs$ (e.g. Figure 2.12(c,d)). The traces (a, b) are results of stochastic simulations (set to single run) of the model in Figure 2.6; where $SusceptiblePopulation$ is set to $5e4'A + 1e5'B$; $k_{infect_a} = 5.0e - 5$; $k_{infect_b} = 5.0e - 6$; $k_{recover_a} = 1.0e - 1$; $k_{recover_b} = 1.0e - 2$.

The stochastic simulation is computationally expensive, especially if it deals with large biological models that involve large numbers of species [ACT+05, LCP+08, Pah09].

2.7.3 Deterministic Simulation

The deterministic simulation is widely used as it is a traditional way to simulate biochemical systems [WUK+04, Gil07, Pah09]. It is well documented with established mathematical basis. It is an accurate approach for a system with sufficient concentration of species. With the assumption that a concentration and a volume of a system is infinite (follow to infinity), an evaluation of a system (reaction influence on species concentration) can be represented as a set of ordinary differential equations (\mathcal{ODE} s) [HR02, WUK+04, Gil07]. The equations have the form of (2.16):

$$\frac{d[S_i]}{d\tau} = f_i([S_1], \dots, [S_N]), \quad (2.16)$$

where $[S_i]$ is a concentration of the species S_i at the current time τ and $f_i([S_1], \dots, [S_N])$ is a function of the species' concentrations. By solving \mathcal{ODE} s, a concentration can be approximated by a continuous variable [GB00]. Through a simulation, the system state (marking) $X(\tau)$ at the current time τ is described as a continuous deterministic process [Gil01]. During this process, a concentration evolves deterministically with time, what means, when a process (simulation) is repeated, starting from the same initial system state, the same state will be reached in any future time point.

Through an elementary kinetic rate laws (e.g. **mass-action**), it is possible to derived/obtain a system of \mathcal{ODE} s.

Example 2.12. The model in Figure 2.14 presents a part of the model in Figure 2.3 (page 13) as \mathcal{CPN} . The model consists of continuous nodes and is derived by a straightforward conversion.

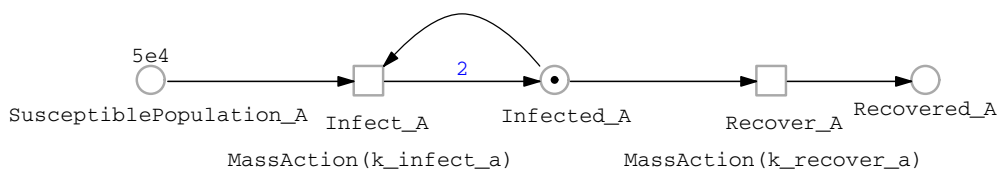
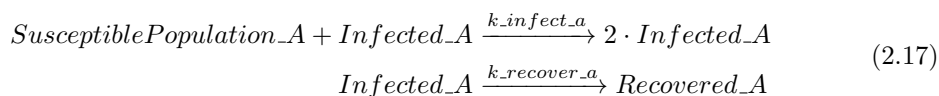


Figure 2.14: SIR model, as \mathcal{CPN} ; where \square - continuous transition, \bigcirc - continuous place, \longrightarrow - directed arc.

The model can be expressed by two elementary reactions (2.17),



2. BACKGROUND AND RELATED WORK

and the system of corresponding $ODEs$ (2.22) can be obtained by applying either (2.8) or (2.9). The following derivation relates to (2.9), as shown in the sequence of steps:

1. The difference of coefficients between *products* and *reactants* species;

$$(\beta - \alpha) = \begin{vmatrix} 0 & 2 & 0 \\ 0 & 0 & 1 \end{vmatrix} - \begin{vmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \end{vmatrix} = \begin{vmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{vmatrix} \quad (2.18)$$

2. The diagonal matrix of kinetic parameters;

$$k = \begin{vmatrix} k_{infect_a} & 0 \\ 0 & k_{recover_a} \end{vmatrix} \quad (2.19)$$

3. The vector-matrix exponentiation;

$$\begin{aligned} S^\alpha &= \begin{vmatrix} SusceptiblePopulation_A \\ Infected_A \\ Recovered_A \end{vmatrix}^\alpha = \\ &= \begin{vmatrix} SusceptiblePopulation_A^1 \cdot Infected_A^1 \cdot Recovered_A^0 \\ SusceptiblePopulation_A^0 \cdot Infected_A^1 \cdot Recovered_A^0 \end{vmatrix} = \\ &= \begin{vmatrix} SusceptiblePopulation_A \cdot Infected_A \\ Infected_A \end{vmatrix} \end{aligned} \quad (2.20)$$

4. The i th species rate;

$$\begin{aligned} \frac{d[S_i]}{d\tau} &= \begin{vmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{vmatrix} \begin{vmatrix} k_{infect_a} & 0 \\ 0 & k_{recover_a} \end{vmatrix} \begin{vmatrix} SusceptiblePopulation_A \cdot Infected_A \\ Infected_A \end{vmatrix} = \\ &= \begin{vmatrix} -k_{infect_a} & 0 \\ k_{infect_a} & -k_{recover_a} \\ 0 & k_{recover_a} \end{vmatrix} \begin{vmatrix} SusceptiblePopulation_A \cdot Infected_A \\ Infected_A \end{vmatrix} = \\ &= \begin{vmatrix} -k_{infect_a} \cdot SusceptiblePopulation_A \cdot Infected_A \\ k_{infect_a} \cdot SusceptiblePopulation_A \cdot Infected_A - k_{recover_a} \cdot Infected_A \\ k_{recover_a} \cdot Infected_A \end{vmatrix} \end{aligned} \quad (2.21)$$

4. The system of corresponding \mathcal{ODE} s;

$$\begin{aligned}
 \frac{d[\text{SusceptiblePopulation}_A]}{d\tau} &= -k_{\text{infect}_a} \cdot [\text{SusceptiblePopulation}_A] \cdot [\text{Infected}_A] \\
 \frac{d[\text{Infected}_A]}{d\tau} &= k_{\text{infect}_a} \cdot [\text{SusceptiblePopulation}_A] \cdot [\text{Infected}_A] \\
 &\quad - k_{\text{recover}_a} \cdot [\text{Infected}_A] \\
 \frac{d[\text{Recovered}_A]}{d\tau} &= k_{\text{recover}_a} \cdot [\text{Infected}_A]
 \end{aligned}
 \tag{2.22}$$

Following **mass-action** law and equation (2.8), the resulting system contains three \mathcal{ODE} s (the number of \mathcal{ODE} s corresponds to the number of unique species in the elementary reactions). Each of them describes the change in a species concentration over time due the occurrence of one of the elementary reactions. The change is proportional to the sum of the product of a transition rate constant and to the reactants concentration, over all elementary reactions, where the stoichiometric coefficients for the reactants are negative, and for the products - positive.

The underlying \mathcal{ODE} s are generated automatically by the simulation library which is an integral part of the *PetriNuts* framework and is used by Spike, Snoopy and Marcie. A numerical solution of the obtained \mathcal{ODE} s can be found by applying different solvers; a classification can be found in [HNW93, HW96]. The basics steps of deterministic simulation are presented by Algorithm 3.

Algorithm 3: Basic deterministic simulation algorithm [HH17].

```

Data:  $\mathcal{CPN}$  with initial state  $X(\tau_0)$ ;
         time interval  $[\tau_0, \tau_{end}]$ ;
         step size  $\delta\tau$  where  $\delta\tau < (\tau_{end} - \tau_0)$ ;
Result: trace of stored system states;
1 define function  $f$  by constructing the ODEs induced by CPN;
2 time  $\tau = \tau_0$ ;
3 state  $X(\tau) = X(\tau_0)$ ;                                /* make ODE solver initial state to  $X(\tau_0)$  */
4  $X(\tau) \rightarrow store$ ;                                   /* add  $X(\tau_0)$  to trace */
5 while  $\tau < \tau_{end}$  do
6      $\tau = \tau + \delta\tau$ ;                                  /* determine next time point */
7      $X(\tau) = X(\tau) + \delta\tau \cdot f(s)$ ;              /* compute a new state */
8      $X(\tau) \rightarrow store$ ;                          /* add  $X(\tau)$  to trace */
9 end

```

The deterministic approach is not the best choice for non-linear systems, due to discreteness and random fluctuations in species concentrations, in particular, when the concentrations are small [MA99, Pah09]. In the deterministic simulation the results are

2. BACKGROUND AND RELATED WORK

approximations of the solutions of ODE s. This may lead to a loss of details in simulation traces. For instance in the averaged traces of the SIR model simulation, it is easy to overlook the case when one becomes infected and then recovers without affecting the rest of a population. In contrast, an exact trace of a single run of a stochastic simulation contains all details as the result is not averaged.

2.7.4 Hybrid Simulation

The hybrid simulation should be applied if a model contains a mix of: slow and fast reactions, or/and small and high concentrations of species. Simulation of hybrid models has been previously investigated in e.g. [HR02, KMS04, SK05, ACT+05, GCP+06]. Applying only deterministic or stochastic approaches may lead to inaccurate or inefficient simulation, respectively. A hybrid simulation overcomes these problems by clustering a model. A cluster contains fast or slow reactions [Pah09], which are simulated in a deterministic or stochastic way, respectively. Fast reactions occur frequently and can be handled/treated as continuous processes. For simulation efficiency, it is better to simulate them using the deterministic approach. In turn, slow reactions are of low frequency and can be a source of various fluctuations/stochastic noise (e.g. volume variation, molecule fluctuations), which may affect the behaviour of a model. This make them more suitable for the stochastic approach. The hybrid approach comes with two main issues, clustering (partitioning) and synchronization of the stochastic and continuous regimes (systems), which need to be solved to achieve efficient and accurate simulation.

A proper clustering is a source of efficient hybrid simulation. When reactions are clustered inefficiently, it can be slower than a stochastic simulation. To cluster fast reactions effectively, it must be taken into account that their reactants have to satisfy thermodynamic conditions i.e. species concentration and system volume should be large enough or close to infinity [Gil07]. A clustering process can be static (off-line, e.g. done by a modeller before start of a simulation) or dynamic (online, during executing a simulation).

A synchronization between both regimes (stochastic and continuous) is essential for obtaining accurate simulation results. As a clustering does not split a model, these two types of simulations have an effect on each other. Fast reactions may depend on the state of the stochastic simulator and the propensities of slow reactions may change with time when the continuous simulator advances [HR02].

Example 2.13. Figure 2.15 (page 34) presents resulting traces of the hybrid simulation of the model in Figure 2.8 (page 18). It is worth noting that the *SusceptiblePopulation* reached negative values (Figure 2.15.(a)). For the efficiency of the continuous simulation, a transition firing is only guarded by a rate function (in the case when the bio-semantic is applied to simulate biochemical reactions). To avoid negative values, a rate function should depend on (be proportional to) the concentrations of reactant species (pre-places).

Algorithm 4: Basic hybrid simulation algorithm - an extended version of this algorithm is introduced in [Her13].

Data: \mathcal{HPN} with initial state $X(\tau_0)$;
time interval $[\tau_0, \tau_{end}]$;
step size $\delta\tau_d$ where $\delta\tau_d < (\tau_{end} - \tau_0)$;

Result: trace of stored system states;

- 1 define function f by constructing the ODEs induced by continuous part of HPN;
- 2 time $\tau = \tau_0$;
- 3 state $X(\tau) = X(\tau_0)$; /* make initial state to current state */
- 4 $X(\tau) \rightarrow store$; /* add $X(\tau_0)$ to trace */
- 5 **while** $\tau < \tau_{end}$ **do**
- 6 **ensure** ODE solver is initialized with $X(\tau)$;
- 7 $\delta\tau =$ determine duration until next firing by computing
- 8 rate function h of transitions depending on current state $X(\tau)$;
- 9 $\tau = \tau + \delta\tau$; /* determine next time point */
- 10 $\tau_d = \tau$;
- 11 **while** $\tau_d < \tau$ **do**
- 12 $\tau_d = \tau_d + \delta\tau_d$;
- 13 $X(\tau_d) = X(\tau_d) + \delta\tau_d \cdot f(s)$; /* compute new state */
- 14 $X(\tau_d) \rightarrow store$; /* add $X(\tau_d)$ to trace */
- 15 **end**
- 16 $t =$ select transition to fire depending on current state $X(\tau)$;
- 17 $X(\tau) = t \rightarrow fire$; /* compute a new state $X(\tau)$ by firing of t */
- 18 $X(\tau) \rightarrow store$; /* add $X(\tau)$ to trace */
- 19 **end**

2. BACKGROUND AND RELATED WORK

If concentrations fall to zero, a firing rate function should prevent a transition from firing (it fires with zero rate) [HH17].

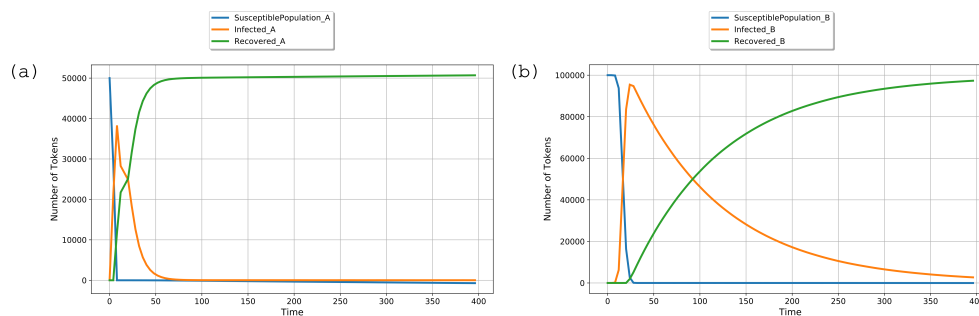


Figure 2.15: Simulation traces of the \mathcal{HPN}^C model in Figure 2.8.

More details about the hybrid approach can be found in [Her13].

2.8 Reduction

A growing amount of experimental data is leading to the development of complex models that may contain numerous nodes. In the case of a \mathcal{PN}^C model, it needs to be unfolded to a \mathcal{PN} before a simulation. After unfolding, the number of nodes can be much larger than in its coloured counterpart. Reduction of a model yields a more optimized (in terms of size) model that provides insight knowledge about structural properties of the model and reduces simulation overhead.

The reduction of \mathcal{PN} models is continuously studied in computer science. The main challenge of reduction is to preserve the main three properties of \mathcal{PN} model: liveness, reversibility and boundedness.

Spike is able to structurally reduce a model by pruning clean siphons (a set of empty places; a marking of those will never be changed because any reaction that would cause a change depends on this set) and constant places (places, occurring only as side conditions).

If a model is to be read as ordinary differential equations (\mathcal{ODEs}), it can be reduced by finding equivalence relations over variables. In [CTT+16] two equivalence relations are presented. FDE (Forward differential equivalence) and BDE (backward differential equivalence) which are implemented in ERODE [CTT+17]. Spike allows to export a model to the ERODE format, however this functionality has **experimental** status.

2.8.1 Pruning Clean Siphons

A Siphon S is a non-empty subset of places $S \subseteq P$ if every transition having an output place in S has also an input place in S , i.e. $\bullet S \subseteq S^\bullet$ [Mur89]. This means that if a siphon once loses all its tokens in its places, there will never be any token in those places. By definition, a clean siphon is a set of empty places, the marking of those will

never be changed because any reaction which would cause a change depends on this set. Therefore, if a clean siphon is empty in the initial state $X(\tau_0)$ or insufficiently marked, it can be safely pruned from a model as a simulation will not change the state of the siphon.

Based on the algorithm proposed in [KLP06], Christian Rohr developed in 2016 the Algorithm 5 that finds a maximal insufficiently marked siphon and a corresponding set of dead transitions, which can be pruned from a \mathcal{PN} model.

Algorithm 5: Maximal insufficiently marked siphon [HRS13].

Data: \mathcal{XPN} with initial state $X(\tau_0)$;
Result: maximal insufficiently marked siphon S and dead transitions D ;

```

1  $S = P$ ;
2  $D = T$ ;
3 for all  $t \in D$  do
4   if  $X(\tau_0) \geq t^- \wedge X(\tau_0) \geq t_r^- \wedge X(\tau_0) < t_i^-$  then
5      $S = S \setminus \bullet t$ 
6   end
7 end
8 do
9    $s = |S|$ ;                                /* store the current size of unmarked siphon set */
10   $d = |D|$ ;                                /* store the current size of dead transitions set */
11  for all  $t \in D$  do
12    if  $\bullet t \cap S = \emptyset$  then
13       $S = S \setminus t^\bullet$ ;
14       $D = D \setminus t$ ;
15    end
16  end
17 while  $s \neq |S| \vee d \neq |D|$ ;
```

Example 2.14. 5 To find a maximal insufficiently marked siphon in Figure 2.16 (a) (page 36), initially all places are considered to belong to the maximal insufficiently marked siphon set S and all transitions are considered to be dead (lines 1-2 of the algorithm). If a transition is live in the initial state $X(\tau_0)$, then all its pre-conditions are removed from the siphon set S (lines 3-7). For all transitions in the set D , if all its pre-conditions are outside the siphon (the transition can structurally be fired), remove all its post-conditions from the set S and remove the transition from the set of dead transitions D (lines 11-16). Repeat this step until no place or transition is removed from the siphon S or dead transitions set D , respectively (lines 8-17). Finally, all places in the siphon and all dead transitions can be pruned from a model. The final result is presented in Figure 2.16 (c).

2. BACKGROUND AND RELATED WORK

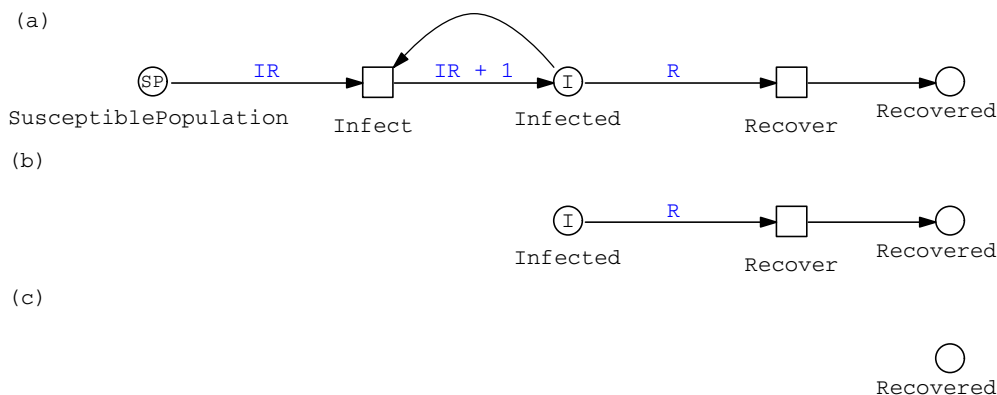


Figure 2.16: (a) Example \mathcal{PN} representing the SIR model where SP and I are initial markings, IR and R are arcs weights. The arc weight IR represents the infection (transition) rate. Assuming that the transition *Infect* is dead in the initial state (e.g. $SP > 0$ and $I = 0$), then the marked place *SusceptiblePopulation* ($SP > 0$) belongs to the insufficiently marked siphon. The place can be pruned with the associated dead post-transition *Infect* which results in the reduced model (b); The model has an additional insufficiently marked siphon if the place *Infected* is initially unmarked ($I = 0$) or the arc weight R is greater than the initial marking of this place (the associated dead post-transition *Recover* will never fire, which allows treating this place as unmarked). This results in the subsequent model reduction (c). Finally, the place *Recovered* can be also pruned as it is an unmarked siphon as well.

2.8.2 Pruning Constant Places

Before a simulation, constant places (places, occurring only as side conditions) can be pruned from rate functions and arc expressions. If an expression depends on a constant place, then the dependency can be removed by replacing a place by its marking value. If an expression involves the *MassAction*(k) pattern then one additional pre-step is necessary. The pattern must be replaced by the expression (2.13) before substituting a place. This allows to speed up an expression evaluation during simulation. To find a constant place, its total weight of pre- and post-arcs can be used. If a total weight is equal zero, then the adjacent arcs do not change the number of place tokens after firing of the pre- and post-transitions. The Algorithm 6 presents this idea in more details. The result of the algorithm execution is a set of constant places which can be pruned by applying Algorithm 7.

Example 2.15. In the model in Figure 2.17 (page 37), the place C is the constant place (its marking is constant upon firing the transition r) which occurs only as a side condition. After replacing the rate function pattern *MassAction*(k) by the expression $(k \cdot A \cdot C)$, Figure 2.17.(b), it can be seen that the reaction rate function depends on the place C . To speed up a simulation, the place can be replaced with its marking value, Figure 2.17.(c).

Algorithm 6: Finding constant places [HRS13].

Data: \mathcal{PN} with initial state $X(\tau_0)$;
Result: constant places C ;

```

1 totalWeight = 0;                               /* total sum of arcs weight adjacent to p */
2  $C = \emptyset$ ;                               /* initialisation of constant places set */
3  $NC = \emptyset$ ;                             /* initialisation of non constant places set */
4 for all  $t \in T$  do
  /* for all places that are simultaneous pre- and post-condition of t */
5   for all  $p \in (\bullet t \cup t \bullet)$  do
6     if  $p \in NC$  then
7       continue;
8     end
9      $A(p) = (p \times t) \cup (t \times p)$ ;        /* finite set of arcs adjacent to p */
10    for all  $a \in A(p)$  do
11      if  $f(p, a)$  increase marking on  $p$  then
12        /* increase total arcs weight */
13         $totalWeight(p) = totalWeight(p) + f(a)$ ;
14      else
15        /* decrease total arcs weight */
16         $totalWeight(p) = totalWeight(p) - f(a)$ ;
17      end
18      if  $totalWeight(p) \neq 0$  then
19         $NC = NC \cup p$ ;
20      end
21    end
22  $C = P \setminus NP$ ;

```

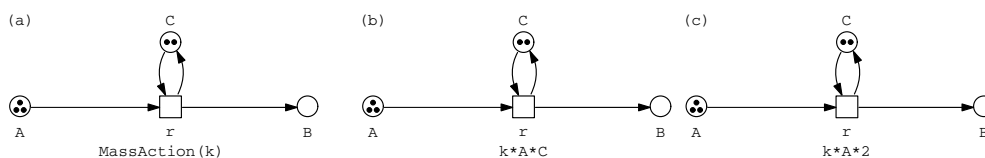


Figure 2.17: (a) Example \mathcal{SPN} with the constant place C , where the reaction rate function depends on this place; (b) the expression $(k \cdot A \cdot C)$ replaced the pattern $MassAction(k)$; (c) the \mathcal{SPN} with the reaction rate function where the place C has been replaced by its marking.

2. BACKGROUND AND RELATED WORK

Algorithm 7: Pruning constant places.

Data: \mathcal{PN} with initial state $X(\tau_0)$;
set of constant places C ;
Result: pruned \mathcal{PN} ;

```
1 for all  $t \in T$  do
2   for all  $p \in \bullet t$  do
3     if  $p \in C$  then
4       if  $v(t)$  depends on  $p$  then
5         /* make it depended on the place marking value */
6          $v(t) = f(m(p))$ ;
7       end
8     end
9 end
```

2.9 Reproducible Simulation

The amount of data and complexity of models force to design an experiment in a way that reproducibility is ensured. By providing computer code, data, models and parameters, one can reproduce results of a simulation.

The term *reproducibility* coexists with the term *replicability*. For these two terms contradicted interpretations exist that vary across a variety of scientific disciplines. More about this issue can be found in [Bar18], where these terms are categorized according to their use in scientific disciplines.

In this thesis the definitions of *reproducibility* and *replicability* are adopted as it stands in [NAP19]:

reproducibility - obtaining consistent results using the same input data; computational steps, methods, and code, and conditions of analysis,

replicability - obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data.

2.9.1 Rules to Drive Reproducible Experiments

The problem with reproducibility of published results is reported in [Hil17] and [KCC05]. One cause of this situation is bad habits of the scientific community. Many results are published without data and source code. Other causes are: no proper simulation set-up, no proper output data analysis, inconsistency of published data (which makes it impossible to compare results). To deal with all these issues, it is necessary **to define** and **adopt** a workflow (set of rules) that will allow reproducing an experiment. Guidelines that can help to establishing such a workflow are proposed in [SNT+13, WAB+11a].

One of the main principles of Spike is to support reproducible simulation experiments. Where it is appropriate, Spike supports ten simple rules proposed in [SNT+13]:

1. **For every result, keep track of how it was produced** - e.g. log all intermediate steps (call of commands, scripts, programs) in a logbook; **with the help of Spike all intermediate steps related to simulation are stored in a single configuration file.**
2. **Avoid manual data manipulation steps** - e.g. all data manipulation should be done by a script or program. If it is not possible, all manual data manipulation should be described and stored in the logbook.
3. **Archive the exact versions of all external programs used** - e.g. an external program may not be any more available or some features have been removed.
4. **Version control all custom scripts** - e.g. even smallest changes to a script can affect the end results.
5. **Record all intermediate results, when possible in standardized formats** - e.g. intermediate results can be used as breakpoints during debugging, provide insight into how experiments were performed, or help find issues if an experiment fails. **Spike allows to log the simulation progress and resulting data.**
6. **For analyses that include randomness, note the underlying random seeds** - e.g. if an experiment involves stochastic simulation, then a seed should be stored what ensures identical final results by initializing a random number generator with the same seed. **Spike allows to log a configuration set-up, and when it is appropriate, includes a seed. For this purpose the logging of a seed is introduced by Spike to achieve a reproducible stochastic simulation.**
7. **Always store raw data behind plots** - e.g. raw data allows the use of various data visualization or analysis techniques. **Spike allows storing raw data in the CSV (comma-separated values) format.**
8. **Generate hierarchical analysis output, allowing layers of increasing detail to be inspected** - e.g. the results of each stochastic simulation run should be stored/logged to inspect the detailed values underlying the final results of a stochastic simulation which represents average of all runs. **Spike allows to set-up multiple simulation in such way that the resulting raw data can be stored in separated files. In addition, Spike allows results of a single run of stochastic simulation to be saved along with the averaged one in single resulting file.**
9. **Connect textual statements to underlying results** - e.g. textual interpretations should directly point to the underlying results, making them easy to trace.

2. BACKGROUND AND RELATED WORK

10. **Provide public access to scripts, runs, and results** - e.g. as supplementary online material to an article.

2.9.2 Encoding of Simulation Experiments

Encoding of a biochemical reaction model is supported by many formats (e.g. ANDL [SRH16], CANDL [LHR12], SBML [Huc15]). This allows a model to be imported / exported and reused in various experiments. Encoding of a model, which is de facto a structure description, does not describe experiments. An experiment should be encoded if it is meant to be reproducible. This issue is addressed in [WAB+11b] where SED-ML (Simulation Experiment Description Markup Language) is introduced. SED-ML is a markup language based on XML. It is built from five main components:

1. **Model component** - defines an identity and location of a model to be simulated (SED-ML supports only models which are encoded in XML-based languages); it also allows for altering attribute values or changing a model structure.
2. **Simulation component** - defines a simulation algorithm and configuration details, e.g. a range of simulation (start and end time) and a resolution (number of points to output).
3. **Task component** - assigns a defined algorithm to a model, the model and simulation are defined separately and can be combined in various ways, e.g. comparing a model behaviour under different simulation algorithms.
4. **Data generator component** - describes transformations of raw simulation data by applying numerical equations, which allow, e.g. for normalization or scaling resulting data.
5. **Output component** - describes grouping of output data from generators, which allows for generating 2D and 3D plots, or output data streams as a set of unrelated arrays.

SED-ML components can occur zero or more times, allowing multiple experiments to be defined in a single SED-ML document.

Rigid standards such as SED-ML cannot cover off-standard use cases, therefore it is necessary to develop domain specific languages that allow for cutting-edge experiments. A good example of such a language is SESSL that bases on the Scala language. SESSL it is a domain-specific language for simulation experiments [EU14]. It acts as a separate software layer on top of external simulation systems. SESSL uses bindings to support a variety of modelling languages, e.g. ML-Rules [WHU17], which is a rule-based modelling language for dynamically nested biochemical reaction networks [MRU11]. SESSL's design focuses on simplicity, which allows users to design an experiment without any deep programming knowledge. However, users with the programming knowledge can

extend SESSL as its components are loosely coupled. Interdependencies between different SESSL components and the dependencies to third-party software are managed by Apache Maven (<https://maven.apache.org/>). This allows for portability and reproducibility as software artefacts used in an SESSL experiment are stored in a Maven repository. Such a solution enables an identical software to repeat simulation experiments on different machines and operating systems. SESSL supports parameter scanning and with the use of bindings integrates various software systems, what facilitates reuse of experiments across simulation systems and analysis of the performance of simulation algorithms at runtime, such as simulation-based optimization.

Spike is built on the configuration script language (SPC - Spike's configuration), which is a domain specific language and has a human-readable format. While the configuration functionality of *SPC* is very similar to SED-ML and SESSL, it is specially tailored to support reproducible simulation experiments of models designed using the *PN* modelling language. *SPC* comprises a set of features that so far are not present in SED-ML, e.g. parameter scanning and stepwise simulation. A stepwise simulation is also not supported by SESSL. *SPC* allows to describe an experiment and its configuration in one file. More details about the implementation of *SPC* can be found in Section 3.1.

Spike unlike to SESSL, does not base on any dependency repositories as all dependencies are included. Both solutions have pros and cons. An external repository requires a careful maintenance as the reproducibility can suffer from changing a software artefact in the repository. From the other hand, dependencies of Spike makes its executable grows with each new dependency added.

Reproducibility suffers when a software has to run on different systems. Some programming languages are more prone to reproducibility and portability issues than others. Inconsistent results may appear in compiled languages e.g. C or C++ due to:

- a floating point numbers precision,
- an undefined behaviour of signed integer overflows,
- how the lengths of certain data types, e.g. *int* and *long*, are defined differently across compilers.

Especially in the case of a floating point, differences on different systems are amplified over many iterations and can be responsible for reproducibility failures although the code is correct. It is worth to consider use of software libraries that provide a consistent floating point precision to obtain reproducible results across different systems as suggested in [MSD+06, BBD+16]

An increase in the degree of reproducibility and portability can be achieved with the help of a virtualization, As it is presented in [Boe15], a virtualization can be achieved with help of e.g. Docker (<https://www.docker.com>). Docker is an open-source project that provides operating system level virtualization. It allows for deployment of applications as portable packages called containers. Containers are isolated from one another and

2. BACKGROUND AND RELATED WORK

resolve the issue of *Dependency Hell* by bundling their own software, libraries and configuration files. However, containers are not a perfect solution, as the virtualization is on an operating system level, a container must match the host architecture. Hardware and related libraries (drivers) require pulling in features from the host itself. This method can threaten long-term reproducibility due to upgrade of hardware and related libraries on the host system. This is notably the case in the High Performance Computing [CY19]. For an application to achieve a top performance, it often needs to be optimized for the architecture and capable of taking advantage of advanced hardware such as accelerators. To achieve a greater degree of reproducibility, a full virtualization should be adopted with a help of a Virtual Machine (VM). Unlike containers, a VM provides a complete system encapsulation, including system-level drivers, a full operating system kernel, and emulation of hardware. A container image could be invoked atop the VM. The dual-virtualization method allows avoiding incompatibilities of hardware and related libraries as the hardware is fully virtualized. Such solution has a drawback, it introduces a measurable source of overhead that can negatively affect the performance.

A different approach towards achieving reproducibility is taken by a workflow software, which are usually built on a well-established collaborative framework between domain and computer scientists. A workflow software usually integrates a set of tools installed on a server that communicates via various communication channels. One of the integral components of the workflow software is a notebook [DTT+16, MCK+18], which helps to achieve a reproducibility and usability. A notebook allows describing an experiment configuration and store it on the server with an associated model. The drawback of the workflow software is the lack of flexibility and the maintenance on the server. A change in one software component can have a significant impact on reproducibility.

In [KR12] is presented a workflow management system called Snakemake. It is a tool to create reproducible and scalable data analyses. Snakemake uses a domain specific language to describe workflows. The language relies on Python language, which allows access to the full power of the underlying programming language e.g. for implementing conditional execution and handling configuration. A workflow is described through a set of rules. Each rule describes a step in an analysis, defining how to obtain input files from output files of previous step. The set of rules creates a directed acyclic graph that represents an execution plan of rules. Each node in the graph represent a job i.e. the execution of a rule. A directed edge joining two jobs A and B defines that the rule underlying job B needs the output of job A as an input file. A path in the graph represents a sequence of jobs that have to be executed sequentially. Two disjoint paths can be executed in parallel. Such approach allows for large-scale data analyses that involve the chained execution of many command line applications. workflow engines like Snakemake help to automate these pipelines and ensure reproducibility.

2.9.3 Adaptive Model Simulation

A simulation experiment typically consists of repeated series of simulation runs in which, after each run, the simulation results are evaluated and the simulation and / or model parameters are modified. Problems, which simulation means have to solve, generally fall into two categories [KW85]:

- System identification or behaviour analysis - when the behaviour and characteristic parameters of a system under various conditions are investigated.
- Reconstruction - when the structure and parameters of a model corresponding to given specifications are determined.

Simulation of dynamically changing processes, which change their dynamic behaviour following the occurrence of external events, requires an ongoing adaptation in terms of time, quality, and flexibility. Therefore, if a model represents such a process, it is necessary to adjust the model according to its state and the current simulation state at the simulation run time. This can improve the quality of the simulation results due to [Jav92]:

- (a) The evaluated input information is obtained from the dynamic trajectory of the model in the time-state space; as opposed to the usual case when static patterns are dealt with.
- (b) The action based on inferencing influences the source of information (i.e. model and experimental conditions) itself. Thereby the procedure forms an inherent part of a closed loop feedback system [BSG+09].

The dynamic adaptation of the model during the simulation runtime falls into the category of self-adaptive systems, an overview of which can be found in [KRV15]. In the case of \mathcal{PN} models, the modelling process can be traded as programming and the simulation of the model as execution of the program. In the context of this informal definition the adaptive model simulation can be treated as adaptive programming [MH91]. An adaptive program changes its behaviour according to its environment. A discrete-time adaptive modelling system (stepwise simulation) scans the set of states S (including historical ones) of the model/system, which provides feedback. Based on the feedback the internal parameters of the model are automatically adjusted by means of predefined conditions. The conditions can have the form of rules e.g. [Jav92]:

$$if \langle condition \rangle \text{ then } \langle action \rangle \tag{2.23}$$

where *condition* can be a boolean expression consisting of variable values at a given time t , i.e.,

2. BACKGROUND AND RELATED WORK

$$\bigcap_{i=1}^{|S_j|} S_i(t), \quad (2.24)$$

where $S_i(t) \in S_j \subseteq S$, j is a variable index and $t \in [0, now]$, where *now* is the current value of simulation time.

As presented in [KCR+09, Kan12], the feedback provided from a model state can be used to dynamical change a simulation algorithm, which allows performing hybrid simulation. Based on the analysis of a system state, transitions can be clustered and assigned to different simulation algorithm. This approach reflects the ability of the system to change its behaviour with respect to the simulation type. [KKV04] presents a different approach, which reflects the ability to change the structure and the behaviour of the simulated model. The structure is changed by replacing components of the model. Depending on the predefined conditions, a new component is selected from the database to replace the old one which is no longer suitable.

[KH96] describes a fast simulation approach for rare events. The technique is based on the RESTART (REpetitive Simulation Trials After Reaching Threshold [VV+91]) method. To find rare events, the state of the model is monitored after each simulation step, which provides the feedback. The set of predefined conditions defines thresholds. When a value of the simulation traces reaches a threshold at time point A , the state of the model is saved. If the threshold is reached from the opposite direction at time point B , the model state is restored and the interval $[A, B)$ is simulated again.

Based on control theory [BSG+09], the model adjustment can be controlled by a feedback loop, which provides the generic mechanism for self-adaptation. The feedback loop comprises activities which describe performed actions. Figure 5.5 (see page 120) presents an example feedback control loop with four defined activities:

- simulation - collects data from executing a model and its current state,
- analyse - analyses the data to infer trends and identify symptoms,
- decide - decides on how to act on the execution of a model based on analyses which are defined by conditions,
- alter model - alters model parameters based on decisions.

More details and an example implementation of the control loop in the stepwise simulation supported by Spike can be found in Section 5.2.

2.10 Closing Remarks

This chapter introduced the paradigm of modelling with Petri nets. Starting with the definition of the basic \mathcal{PN} through extended and quantitative Petri nets, i.e. \mathcal{XPN} , \mathcal{SPN} , \mathcal{CPN} , \mathcal{HPN} and ending with their coloured counterparts \mathcal{SPN}^c , \mathcal{CPN}^c , \mathcal{HPN}^c . The SIR model was developed as an example for each introduced type of \mathcal{PN} . Examples were shown to illustrate how the developed models behave under three types of simulation, i.e. deterministic, stochastic and hybrid.

This chapter also describes the definition of reproducible simulation with the guidelines on how to achieve it.

Finally, a brief overview of the adaptive model simulation paradigm is provided, which opens the door for more realistic simulation experiments.

The next chapter will introduce the Spike Configuration Script (\mathcal{SPC}), which allows defining reproducible simulation experiments by setting up model parameters and simulation options.


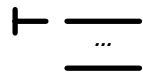




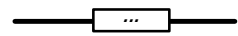

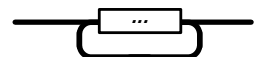
3

Configuration Language

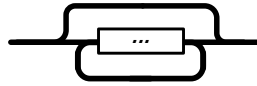
Configuration files are often used to set up initial parameters of computer programs. Similarly, the Spike Configuration File (*SPC*) allows setting up a reproducible simulation experiment. Through the embedded branching, one *SPC* script may involve a set of configurations. This feature allows scanning of model parameters and configuration options. It consists of a static and a dynamic part. The static part permits the configuration of model and simulation parameters, whereas the dynamic part allows to reconfigure a model during stepwise simulation.

The following graphical notations are used across this chapter to define relations between various entities.

Notation 2. *Graphical notations:*

	<i>definition entry point;</i>
	<i>parallel entry point - states that all entry points/paths must be chosen;</i>
	<i>definition end point;</i>
	<i>path - path to proceed;</i>
	<i>split path - path splitting states, only one direction can be chosen;</i>
	<i>join paths - combined paths become one;</i>
	<i>one occurrence of an entity;</i>
	<i>zero or one occurrence;</i>
	<i>one or many occurrences;</i>

3. CONFIGURATION LANGUAGE



zero or many occurrences;



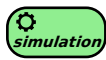
higher order abstract definitions - abstract definitions without any technical details;



SPC properties - properties used to define/configure an experiment;



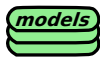
grammar definitions



abstract process - represents one occurrence of a computational activity;



abstract data - represents data in the form of a report;



abstract database - represents set of data in the form of a database;



SPC basic property - represents a property that does not group any other properties;



SPC complex property - represents a property that groups one or more basic properties;



grammar operator - represents an operator;



grammar literal - represents a constant/fixe value;



grammar rule - represents a meta variable/nonterminal symbol;

3.1 *SPC* Format

SPC is an integral part of Spike, however it is language-independent and can be adopted for use in various software. The *SPC* format is inspired by JSON (JavaScript Object Notation) [RFC7159, ECMA-404]. JSON is an open standard data interchange format. It uses human-readable text to store serialized data objects consisting of name/value pairs and array data types. It does not enforce any order of sorted data and does not allow for any condition. The base *SPC* structure is similar to the JOSN format and the conventions used by *SPC* are familiar to the programmers of the C-family languages as well as JavaScript, Python and many more. *SPC* is built on two structures:

- An unordered set of name/value pairs that defines the variables, see Figure 3.13 (page 60). In various languages this can be realized also as a record, struct,

dictionary, hash table, keyed list or associative array, see Figure 3.16 (page 61). In *SPC* it is realized by an *object*, which is a variable that groups many variables. Any variable declared inside an object becomes its member and is called a property. An object is called a **complex property**. A variable that does not group any other variables is called a **basic property**.

- An ordered set of values, is realized in *SPC* as an *array*. In various languages this can be realized also as a vector, list, sequence, see Figure 3.18 (page 62).

In contrast to JSON, *SPC* allows defining multiple variations of stored data. These variations are called *branches*. A branch represents a separated *SPC* script. Additionally, *SPC* consists of a dynamic part where the order of stored information is important. The dynamic part of *SPC* is a fully functional programming scripting language that allows for a conditional simulation execution. Spike may carry out parallel simulation of branches, what requires a lightweight formate for an interprocess communication. *SPC* meets this requirement, similar to JSON. It is extremely lightweight in comparison to formats based on a markup language, e.g. XML [ZDS14].

SPC is agnostic to data types until script evaluation. This means that casting operators are applied when working with data (during an evaluation) and variable types are enforced at runtime. *SPC* is weakly typed language. This means, it is not allowed to add a *number* to a *string* using an arithmetic operator, but there is no restriction to add a decimal to an integer *number*. This has implication on how errors are handled. *SPC* does not care for the type of a variable or literal, as long as the evaluator has a way to handle it. During the evaluation of the script, a variable can change its type several times, as the assigned value determines its type.

3.2 Experiment Definition

SPC defines a numerical experiment as a set of five descriptive components:

1. models to be used in the experiment,
2. simulation algorithms,
3. combination of models and simulation algorithms into a numerical simulation,
4. data generators,
5. output/storing of results.

An overview of the high level relations between them is presented in Figure 3.1 (page 50).

3. CONFIGURATION LANGUAGE

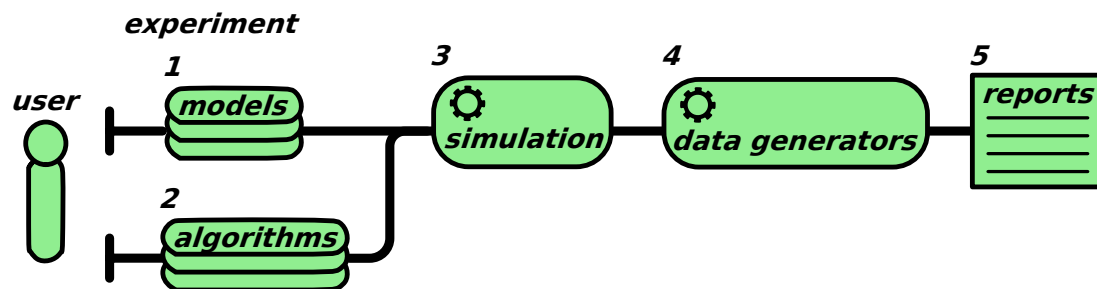


Figure 3.1: High level overview of the relations between the main components of an experiment.

These five components are reflected in *SPC*, see Figure 3.2 (page 51).

1. A model can be imported and its parameters may be modified in the model configuration object.
2. Simulation algorithms (solvers) are defined in the simulation configuration object.
3. A combination of model, simulation and optionally stepwise simulation configuration, defines a process of numerical simulation.
4. The data generation process is defined by places and transitions of a model. Additionally, observers (auxiliary variables) can be defined in a model and stepwise simulation configuration.
5. Reports are defined by export of objects.

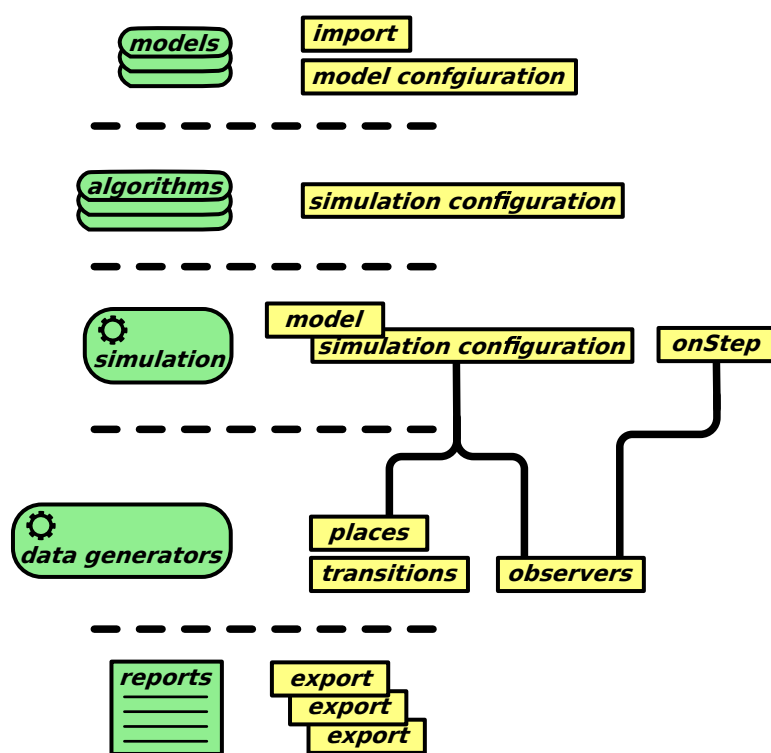


Figure 3.2: Graphical representation of relations between an experiment and main *SPC* objects.

3.3 Main *SPC* Objects

SPC consists of three main objects (complex properties), see Figure 3.3.

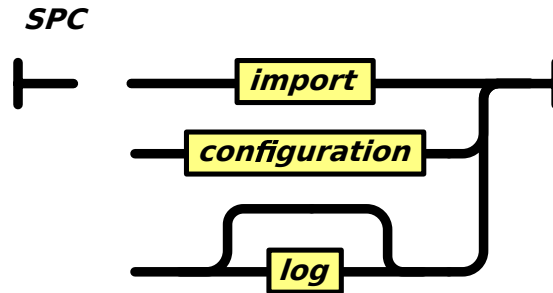


Figure 3.3: The three main objects of *SPC*.

Example 3.1. The three main objects declared in *SPC* format. NOTE: The order of components is not important unless explicitly stated.

```
1 // One line comment
2 /*
3  * Multi line comment
4  */
5
6 /*
7  * Import configuration
8  */
9 import: {...}
10
11 /*
12  * Model and simulation configuration
13  */
14 configuration: {...}
15
16 /*
17  * Logging user-defined variables configuration
18  */
19 log: {...} // [OPTIONAL]
```

3.3.1 Import

Import is a complex property that names the model and defines its location, see Figure 3.4 (page 53). It groups (consists of) three properties:

- *from* - the basic property, defines which model to import by giving a path to a model location. The path can be absolute or relative to a *SPC* file.

- *name* - the optional basic property, allows overriding the name of an imported model. If it is skipped in the configuration, then the name of an imported model is assigned as default value of this property.
- *sbml* - the optional complex property, it is required if an imported model is in SBML format.

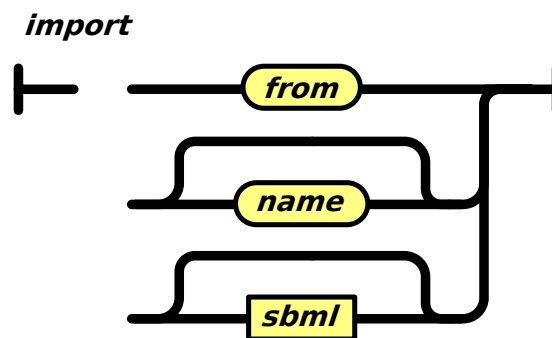


Figure 3.4: Import object.

The *sbml* is a complex property (object), see Figure 3.5, and groups three basic properties:

- *net* - allows defining whether a model should be imported as deterministic, stochastic or hybrid \mathcal{PN} ,
- *boundary* - determines if boundary reactions (in/out flow) should be added for all boundary conditions,
- *reversible* - determines if a reversible reaction should be replaced by two one-way reactions.

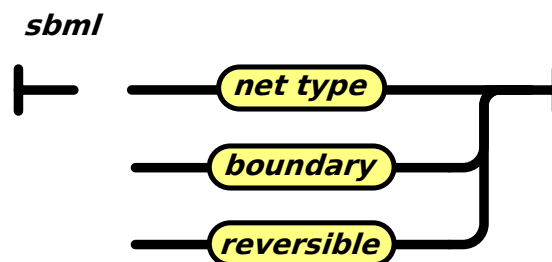


Figure 3.5: SBML object.

3. CONFIGURATION LANGUAGE

Example 3.2. Declaration of import.

```
1 /*
2  * Import configuration
3  */
4 import: {
5   from: "./path/to/model";
6   name: "example1"; // [OPTIONAL]
7   sbml: {
8     /*
9      * Import a model as CPN (continuous PN)
10     * or SPN (stochastic PN)
11     */
12    net: "CPN";
13    boundary: true;
14    reversible: false;
15  } // [OPTIONAL]
16 }
```

3.3.2 Configuration

Configuration is a complex property that allows configuring a model and a simulation (see Figure 3.6). It may consist of two complex properties: *model configuration* and *simulation configuration*.

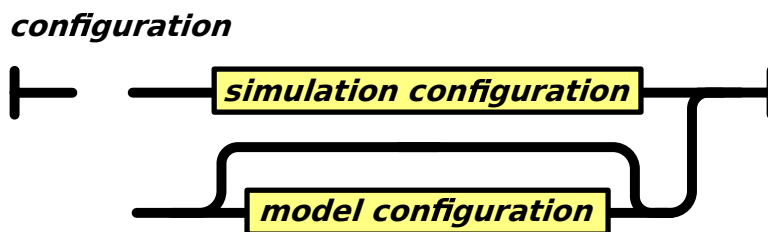


Figure 3.6: Configuration object.

The *model configuration* object, see Figure 3.7 (page 55), has three complex properties:

- *constants* - allows for altering a value of a model attributes via parameters specifying arc weights, initial marking or kinetic parameters;
- *places* - allows to directly alter the initial marking of places;
- *observers* - allows defining/overriding observers which are auxiliary variables, which allow for extra measures by defining numerical functions; depending on the type of observer, it can involve, constants (that belonged to the model), places, transitions or simultaneously places and transitions;

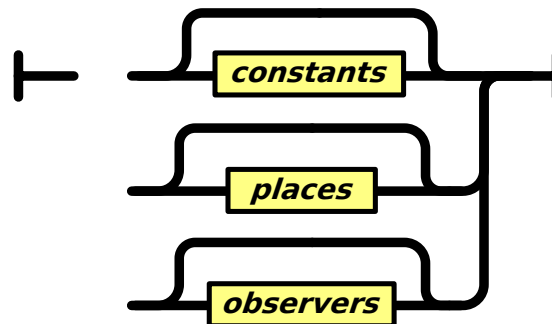
model configuration

Figure 3.7: Model configuration object.

The *simulation configuration* object (see Figure 3.8) may consist of four (or more - the property *export* can be defined multiple times) complex properties:

- *simulation options* - allows defining a simulation algorithm (solver) and its configuration details via the usual simulator-dependent options;
- *interval* - allows defining the range of a simulation (start and end time) and a resolution (number of resulting points - snapshots taken during simulation);
- *onStep* - allows defining a steering script of stepwise simulation; this object has special behaviour and properties that are described in details in Section 3.7;
- *export* - allows specifying multiple exports of simulation results by use of regular expressions over the nodes of which the simulation traces are to be recorded; it is possible to combine the results of places, transitions and observers, coloured and uncoloured, in one CSV file.

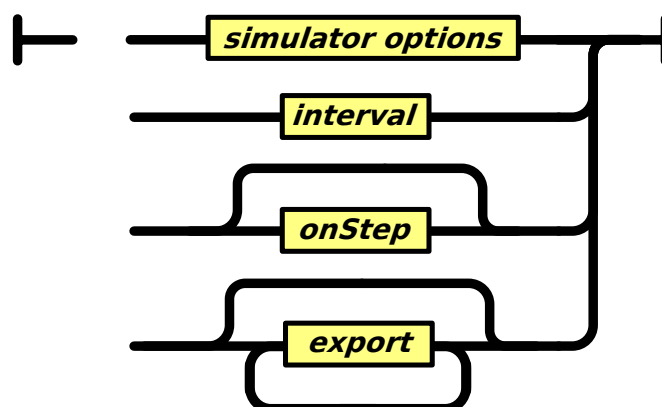
simulation configuration

Figure 3.8: Simulation configuration object.

3. CONFIGURATION LANGUAGE

Example 3.3. Declaration of a model configuration. **Note:** An observer function is given as string as it is evaluated by an external evaluator.

```
1 configuration: {
2   ...
3   model: {
4     // Overriding constants
5     constants: {
6       valueset: "Main"; // Select global values set - [OPTIONAL]
7       all: { // The group name in a ANDL/CANDL model e.g.: "all"
8         // Select values set for the group
9         valueset: "Main"; // [OPTIONAL]
10        C1: 1; // Set constant value
11        C2: 2;
12      }
13    }
14    // Overriding initial markings
15    places: {
16      P1: 1; // Not coloured model
17      P2: "90'a++80'b"; // Coloured model
18    }
19    // Overriding/deflation observers
20    observers: {
21      place: { // [OPTIONAL]
22        OP: {
23          function: "(P1 + P2) / 2";
24        }
25      }
26      transition: { // [OPTIONAL]
27        OT: {
28          function: "t1 / 4";
29        }
30      }
31      // Involve places and transitions
32      mixed: { // [OPTIONAL]
33        OM: {
34          function: "P1 + t1";
35        }
36      }
37    }
38  }
39  ...
40 }
```

Example 3.4. Declaration of a simulation configuration.

```

41 configuration: {
42   ...
43   simulation: {
44     name: "example"; // Name of a simulation
45     type:nameOfType: {
46       solver:nameOfSolver: {
47         // Solver options
48         ...
49       }
50     }
51     interval: 0:100:100; // start:splitting:end
52     // Define the stepwise simulation
53     onStep: { // [OPTIONAL]
54       ...
55       do: {
56         ...
57       }
58     }
59     // Export results to a file
60     export: { // [OPTIONAL]
61       ...
62     }
63   }
64   ...
65 }

```

3.3.3 Log

Log is a complex property and may consist of many basic properties that store additional data, that will be reported in a log.

Example 3.5. Declaration of the logging of user-defined variables.

```

1 /*
2  * Logging of user-defined variables
3  */
4 log: {
5   simulation: configuration.simulation.name;
6 }

```

3. CONFIGURATION LANGUAGE

3.4 Basic definitions

3.4.1 Value

A value can be an object, an array, a range, a number, a string, true or false (see Figure 3.9).

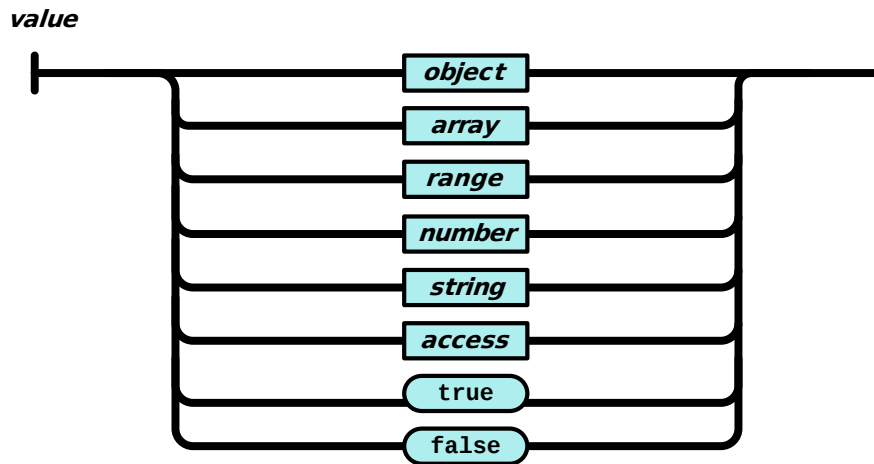


Figure 3.9: *SPC* value.

The simplest values are literals and variables.

3.4.2 Literal

Literals are explicitly written constant values. *SPC* defines three types of literals:

1. **number** - is a sequence of digital characters that may contain a decimal part separated by the *dot* `.` terminal and can also be written in scientific notation, see Figure 3.10.

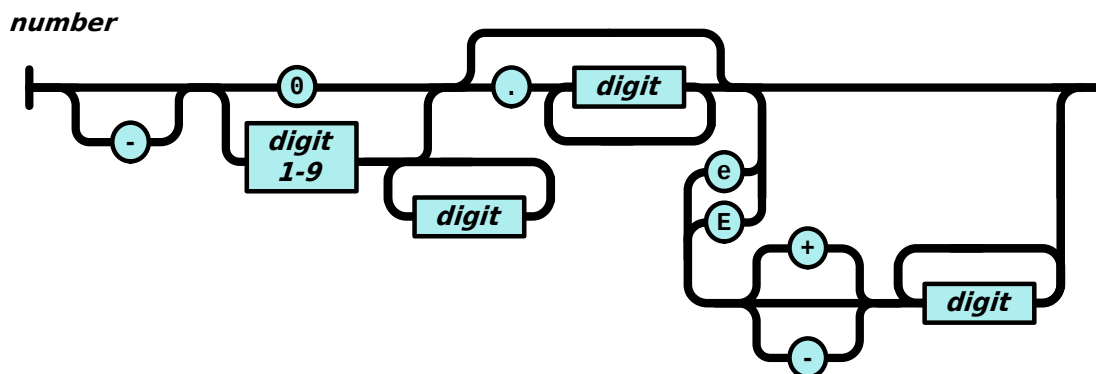


Figure 3.10: *SPC* number. A digit is an atomic character unit in the range from 0 to 9; a sequence of digits creates a number.

2. **string** - is a sequence of characters, except the quotation mark, which is used to wrap the string, see Figure 3.11.

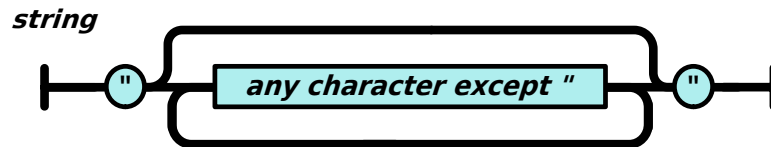


Figure 3.11: *SPC* string. A character is an atomic unit; a sequence of characters creates a text string wrapped in quotation marks.

3. **boolean** - a logical value represented by two literals, *true* and *false*, see Figure 3.12.

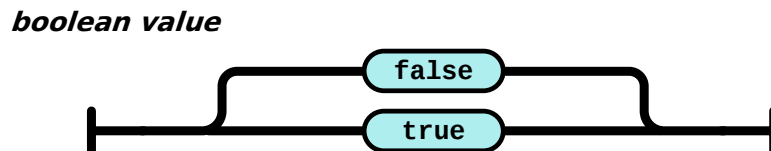


Figure 3.12: *SPC* boolean value is represented by two literals, *true* and *false*.

Example 3.6. *SPC* literals.

```

1 /*
2  * Number
3  */
4 1002    // A number without decimals
5 10.02   // A number with decimals
6 12.0e3  // 12000
7 12.0e-3 // 0.012
8
9 /*
10 * String
11 */
12 "example string"
13
14 /*
15 * Boolean
16 */
17 true
18 false

```

3. CONFIGURATION LANGUAGE

3.4.3 Variable

Variables are named values and are used to store data values. A declaration of a variable enforces its initialization with a value, Figure 3.13.

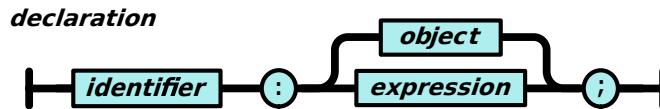


Figure 3.13: *SPC* declaration of a variable with an initial value.

A variable can have different values (which may be of different types) at different times, and it is identified by an *identifier*, see Figure 3.14.

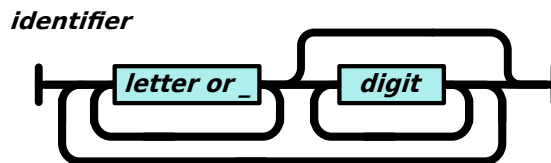


Figure 3.14: *SPC* identifier.

An *identifier* is a unique name in a scope, it is defined by a parent *object* of a variable. A new value can be assigned to a declared variable with the assignment operator **=**, see Figure 3.15.

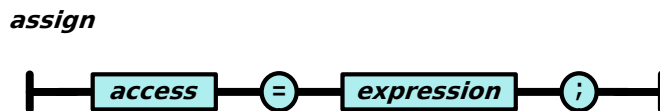


Figure 3.15: *SPC* assignments of a value to the previously declared variable.

The assignment operator is valid in the context of the stepwise simulation object. More about its usage can be found in Section 3.7.

Example 3.7. *SPC* variables.

```

1  /*
2  * Variable declaration
3  */
4  a: 1;
5  b: "text";
6  c: true;
7
8  /*
9  * Value assignments
10 */
11 a = 1;
12 b = "text";
13 c = true;

```

3.4.4 Object

An object is an unordered and unindexed data structure that groups many variables, which are surrounded by curly brackets `{...}`, see Figure 3.16. Any variable declared inside an object becomes its member and is called a property. A variable that does not group any other variables is called a **basic property**. An object is called a **complex property**, which groups statements depending on their functionality and its definition can cover several lines. Each object creates its own scope with separated set of variables. Variables can share the same identifier, iff they are not members of the same scope (object).

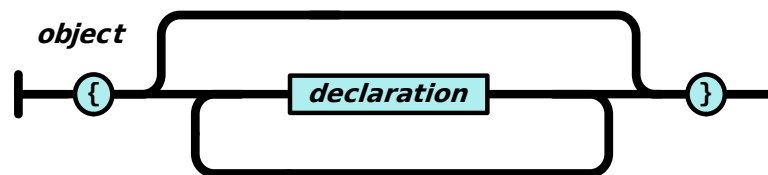


Figure 3.16: *SPC* object.

Accessing Object Members/Properties. In the scope of an object, its members can be accessed directly by their identifiers. Outside a parent scope, an object member can be addressed by using a fully-qualified or relative path. The dot `.` operator separates identifiers, see Figure 3.17.

3. CONFIGURATION LANGUAGE

access

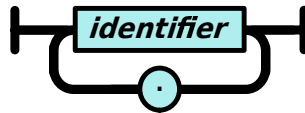


Figure 3.17: *SPC* access.

Example 3.8. *SPC* object declaration and access to properties.

```
1 /*
2  * Object declaration
3  * with properties
4  */
5 parent: {
6   a: 1;
7   child: {
8     a: 2;
9   }
10 }
11 /*
12  * Access to the object properties
13  */
14 parent.a
15 parent.child.a
```

3.4.5 Array

An array is an ordered data type, that can hold more than one value of the same type at any time. Values are comma separated and enclosed in square brackets, see Figure 3.18.

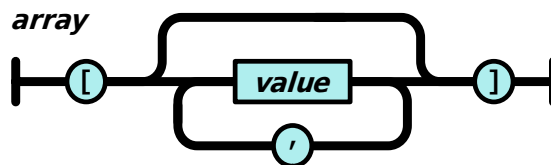


Figure 3.18: *SPC* array.

Accessing Array Elements. In the current version of *SPC*, elements of an array **CAN NOT** be accessed directly, as they are used only to set values of some configuration options.

3.4.6 Range

A range is a special type of variable that defines array values of the type *number*. It consists of three values (operands) of the type *number* separated by colon `:`, see Figure 3.19, where:

- first - defines the start of a range,
- second - step size which is used to obtain the next array element, what can be described by a simple algorithm:

```

1 value = start
2 while value <= end do
3   add_value_to_array(value)
4   value = value + step
5 end do

```

- third - the end of a range.

range



Figure 3.19: *SPC* range.

Example 3.9. *SPC* array and range declaration.

```

1 n: [1, 2, 3]; // Array of numbers
2 s: ["a", "b", "c"]; // Array of strings
3 o: [obj1:{x:1;}, obj2:{x:2;}); // Array of objects
4 /*
5  * Declaring of a range is
6  * equivalent to an array declaration
7  * r: [1.0, 1.5, 2, 2.5, 3];
8  */
9 r: [1:0.5:3];

```

3.5 Expressions

An expression is a combination of operands (variables and values) and operators. Depending on the operator used, the operands can be of different types, such as numbers, boolean values and strings, see Figure 3.20 (page 64). An expression computes a value. The expression computation is called an evaluation. An expression can be used to assign a value to a variable through the *assignment* operator or during its declaration. The evaluation result determines the type of a variable. Operations with

3. CONFIGURATION LANGUAGE

a higher precedence are evaluated first. Round brackets may be used to change the precedence and thus to control the order of evaluation.

SPC distinguishes four types of expressions: arithmetic, boolean, comparison and concatenation.

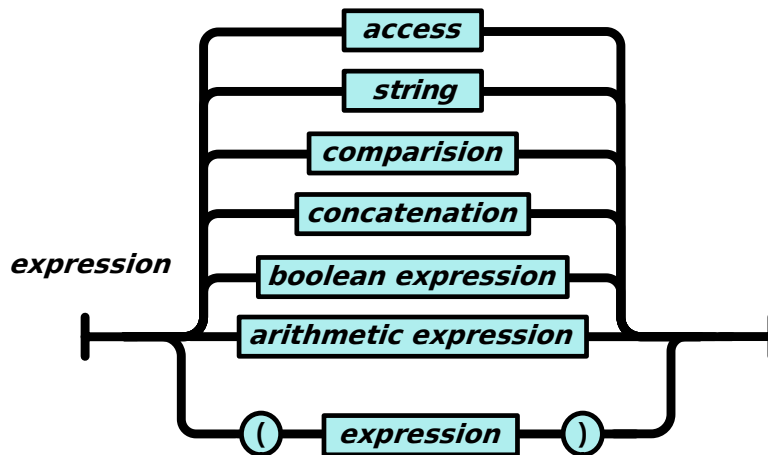


Figure 3.20: *SPC* expression.

3.5.1 Arithmetic Expression

An arithmetic expression (see Figure 3.21) is an expression, that evaluates to a number value. The simplest arithmetic expressions are numerical literals and variables.

arithmetic expression

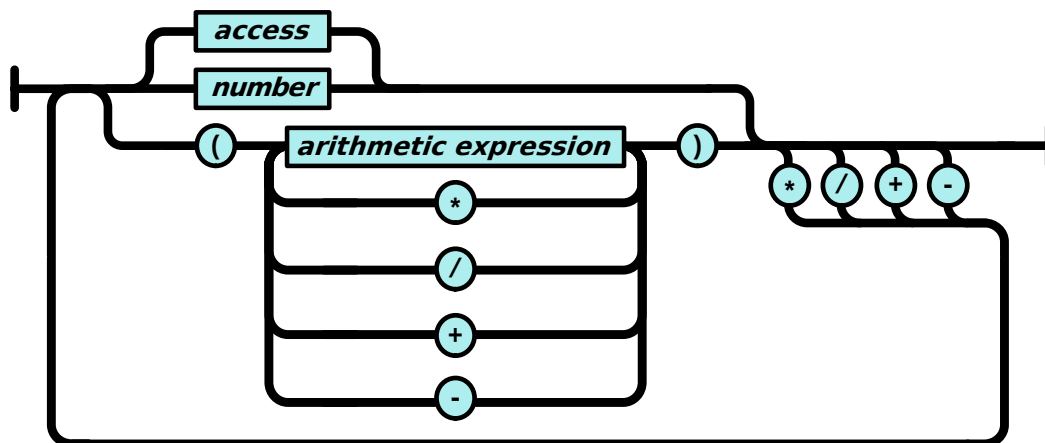


Figure 3.21: *SPC* arithmetic.

Complex arithmetic expressions can be formed by connecting the simplest arithmetic expressions with one of the arithmetic operators (see Table 3.1) and can be grouped

using round brackets (...)

Table 3.1: Arithmetic operators.

Operator	Meaning
+	add - if one of the operands is a real type, the result is real
-	subtract - if one of the operands is a real type, the result is real
*	multiply - if one of the operands is a real type, the result is real
/	divide - division of two integer values will give a real result

3.5.2 Boolean Expression

A boolean expression, see Figure 3.22, is an expression that evaluates to a boolean value, i.e. *true* or *false*. The simplest boolean expressions are numerical literals and variables. Complex boolean expressions can be formed by connecting the simplest boolean expressions with one of the boolean operators, Table 3.2, and can be grouped using round brackets (...).

Table 3.2: Boolean operators.

Operator	Meaning
&&	and - denoted by $x \& \& y$
	or - denoted by $x y$
!	not - denoted by $!x$

The denoted expression values can be expressed by a truth table, see Table 3.3.

Table 3.3: The truth table of boolean operations.

x	y	$x \& \& y$	$x y$	$!x$
false	false	false	false	true
true	false	false	true	false
false	true	false	true	true
true	true	true	true	false

3. CONFIGURATION LANGUAGE

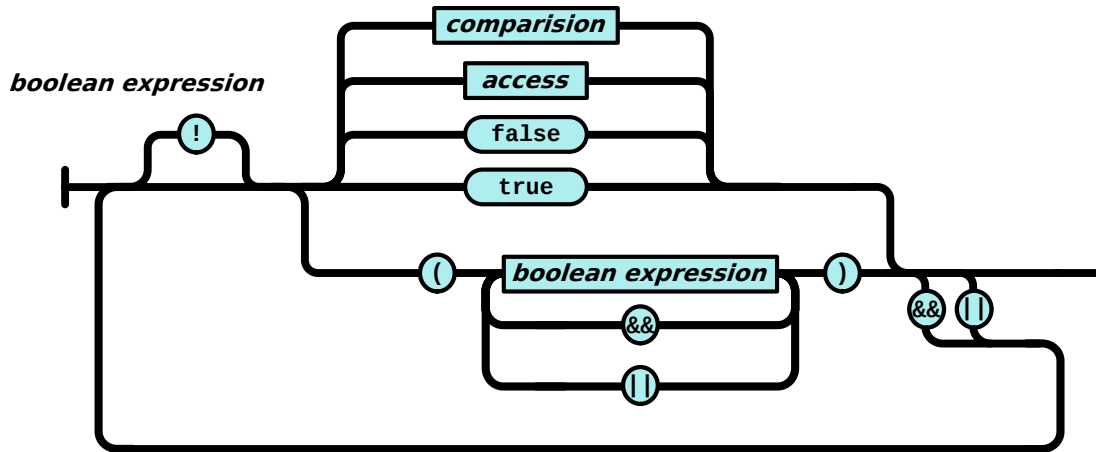


Figure 3.22: *SPC* boolean.

3.5.3 Comparison Expression

A comparison expression, see Figure 3.23, checks whether a literal value, variable value, or expression result is equal, not equal, greater than, or less than another value. The result of a comparison expression evaluation is a boolean value. Comparison expressions can be formed by connecting the arithmetic or boolean expressions with one of the test operators, see Table 3.4.

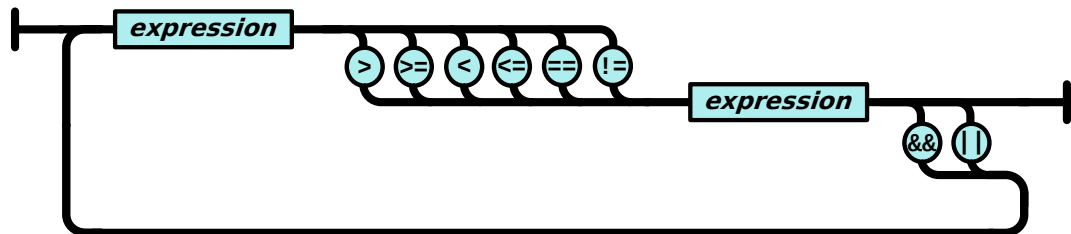
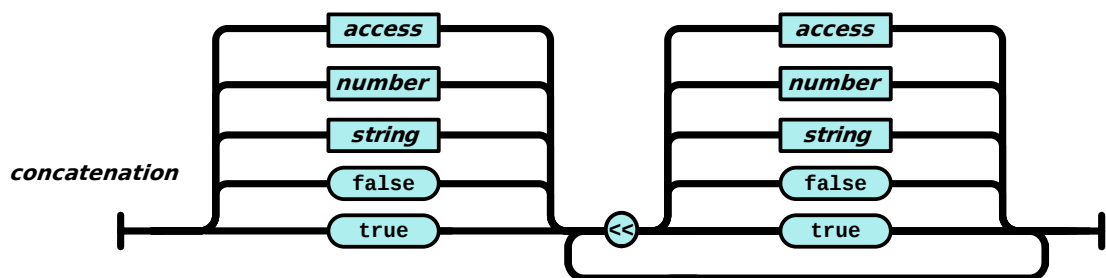
Table 3.4: Test operators.

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

The meanings of these operators are obvious, they can be combined with the boolean operators.

3.5.4 Concatenation

A concatenation expression (see Figure 3.24) concatenates its operands. Before concatenation, the value of each operand is converted to string. To avoid errors/mistakes of a false evolution, round brackets should surround expressions.

comparisonFigure 3.23: *SPC* comparison.Figure 3.24: *SPC* string concatenation.**3.5.5 Precedence**

The result of an expression evaluation depends on the precedence of operators, see Table 3.5. The order of operator evaluations in relation to each other is determined by precedence rules. Evaluation results of operators with higher precedence become the operands of operators with lower precedence.

Table 3.5: Precedence of operators.

Entries at the top of the table have the highest precedence; entries in the same table row have equal precedence.

Operators	Meaning
* /	multiply, divide
+ - <<	add, subtract, concatenation
!	logical <i>not</i>
< > <= >=	less than, greater than, less than or equal to, greater than or equal to,
== !=	test if equal, test if not equal
&&	logical <i>and</i>
	logical <i>or</i>
: =	assignment with declaration, simple assignment

Evaluation of operator precedence in an arithmetic expression:

1. Anything inside round brackets is evaluated first,

3. CONFIGURATION LANGUAGE

2. unary minus is evaluated next,
3. multiplications and divisions are evaluated before additions and subtractions,
4. operations of equal precedence are evaluated from left to right e.g. $10 - 5 - 1$ evaluates to 4 and **not** to 6,
5. assignments are evaluated last.

Evaluation of operator precedence in a boolean expression:

1. Anything inside round brackets is evaluated first,
2. arithmetic is evaluated before equality and inequality tests,
3. logical operations are evaluated after equality and inequality tests,
4. *not* (!) is evaluated before *and* (&&),
5. *and* (&&) is evaluated before *or* (||),
6. assignments are evaluated last.

Example 3.10. *SPC* expressions.

```
1 /*
2  * Expressions
3  */
4 a: 1 + 2;    // Evaluates to 3
5 b: a == 3    // Evaluates to true
6 c: (2 + a) * 5 // Evaluates to 25
7 /*
8  * String concatenation
9  */
10 s: "text_" << 1; // Evaluates to text_1
11 /*
12  * Precedence of operators
13  */
14 a + 1 > a && a > 0 // Evaluates to true
15 /*
16  * and is equivalent to
17  */
18 ((a + 1) > a) && (a > 0)
```

3.6 Conditional Block

A conditional block, see Figure 3.25, consists of conditional statements, that allow to perform different actions based on various conditions represented by boolean expressions:

- **if** - executes a block of code, if a given condition is true;
- **else if** - tests a new condition if the previous condition is false and executes an alternative block of code if the current condition is true;
- **else** - executes a block of code if all previous conditions are false.

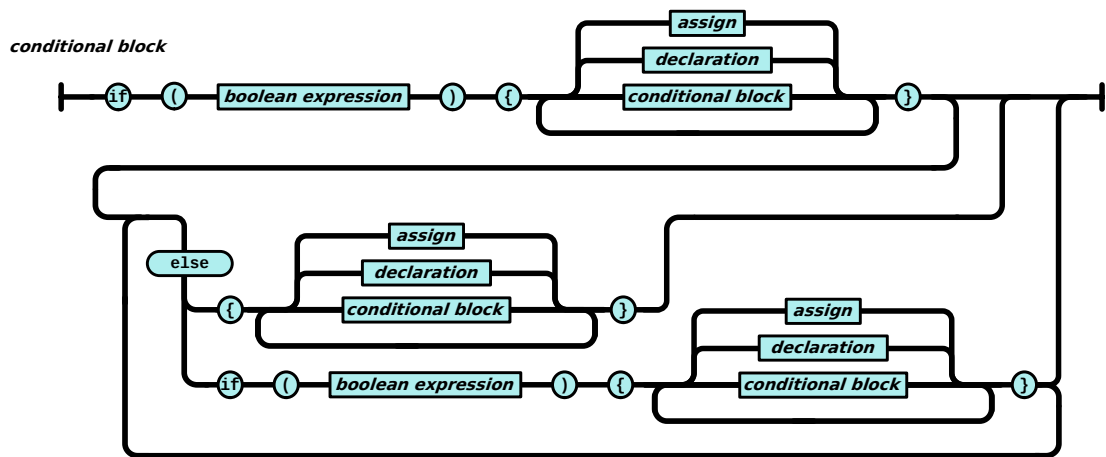


Figure 3.25: *SPC* conditional block.

3. CONFIGURATION LANGUAGE

Example 3.11. Conditional block declaration.

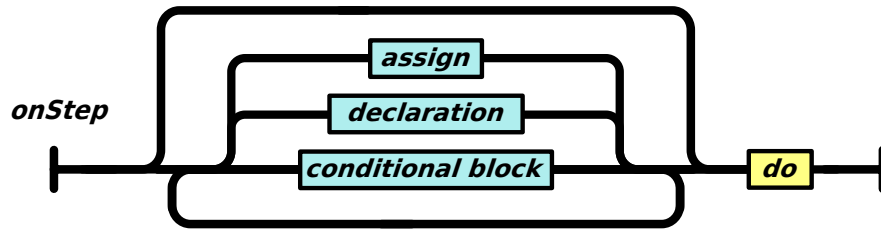
```
1 if(condition1) {
2   /*
3    * Block of code to be executed
4    * if the condition1 is true
5    */
6   ...
7 } else if(condition2) {
8   /*
9    * Block of code to be executed
10   * if the condition1 is false
11   * and condition2 is true
12   */
13   ...
14 } else {
15   /*
16   * Block of code to be executed
17   * if all previous conditions
18   * are false
19   */
20   ...
21 }
```

Example 3.12. A boolean expression can be used as a condition in a conditional statement.

```
22 x: 1;
23 y: 2;
24
25 if((y - 1) >= x && x != 0) {
26   /*
27   * Block of code to be executed
28   * if the condition is true
29   */
30   ...
31 }
```

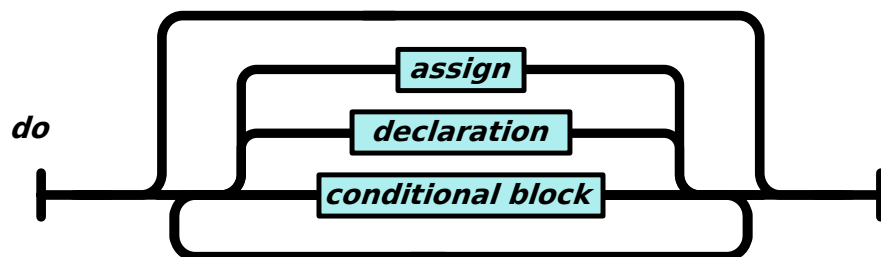
3.7 Stepwise Simulation

A stepwise simulation allows to adjust a model after each simulation step, based on the current state of a model and a running simulation. To set up a stepwise simulation, the object *onStep* needs to be declared, see Figure 3.26 (page 71). It is a member of the *simulation configuration* object.

Figure 3.26: *onStep* object.

The *onStep* object and all its properties are evaluated step by step. This means that the order of declarations of variables, occurrence of expressions and conditionals blocks is important. It can be logically divided into two parts:

- declaration - is evaluated only once, after the first simulation step (first step of simulation is performed on an initial state of a model). It allows to declare and initialize variables. A declared variable can be observed (added to a set of observers), which allows to include them in a simulation result. The value of declared variable can be changed by use of the *assignment* operator. Assigning a new value can be conditional (within the conditional block) and involves expressions consisting of previously declared variables.
- after step evaluation - the *do* object, see Figure 3.27, is evaluated after each step of a simulation. It allows for assigning new values to previously declared variables. Similarly, like in the declaration part, assigning can be conditional and involves expressions. The declaration of a new variable is allowed, however it should be avoided for better performance.

Figure 3.27: *do* object - is evaluated after each step of a simulation.

In the scope of the *onStep* object, it is allowed to read and write the values of places (markings) and constants. Read access is also possible to the current time and the current step of a simulation. Additionally, variable states (values) can be logged by assigning an expression to the predefined variable **LOG**.

3. CONFIGURATION LANGUAGE

Example 3.13. Declaration of a stepwise simulation.

```
1  /*
2  * Stepwise simulation
3  */
4  onStep: {
5      /*
6       * Declaration part - evaluates only
7       * once at the beginning of a simulation
8       */
9
10     a: 0;
11     /*
12      * A variable can be added to observers,
13      * what allows recording how variable
14      * change over a simulation time
15      */
16     b:observe: 0;
17
18     do:{
19         /*
20          * Main loop part - evaluates after
21          * each simulation step
22          */
23
24         /*
25          * Log values
26          */
27         LOG = "time: " << simulation.time;
28         LOG = "step: " << simulation.step;
29
30         if(a < 10 && place.P > 5) {
31             if(constant.C > 0) {
32                 constant.C = constant.C - 1;
33             }
34         } else {
35             constant.C = constant.C + 1;
36         }
37         a = a + 1;
38         LOG = "value_a: " << a; // Log value of the variable a
39         b = b + 1; // Increment observable variable
40     }
41 }
```

3.8 Configuration Branching

Branching is an operation on a configuration; it is triggered by defining in the configuration a set of parameters to scan. Each parameter to scan needs to be defined as an array of values. A new configuration branch is created for each value in an array (if the size of an array is > 1). To distinguish branches from a regular array, a list of values are surrounded by double square brackets - the *branching operator* `[[...]]`. Such a feature allows a configuration script to be split into separate branches, what results in multiple simulation configurations, see Figure 3.28. The set of configuration branches can be executed sequentially or in parallel.

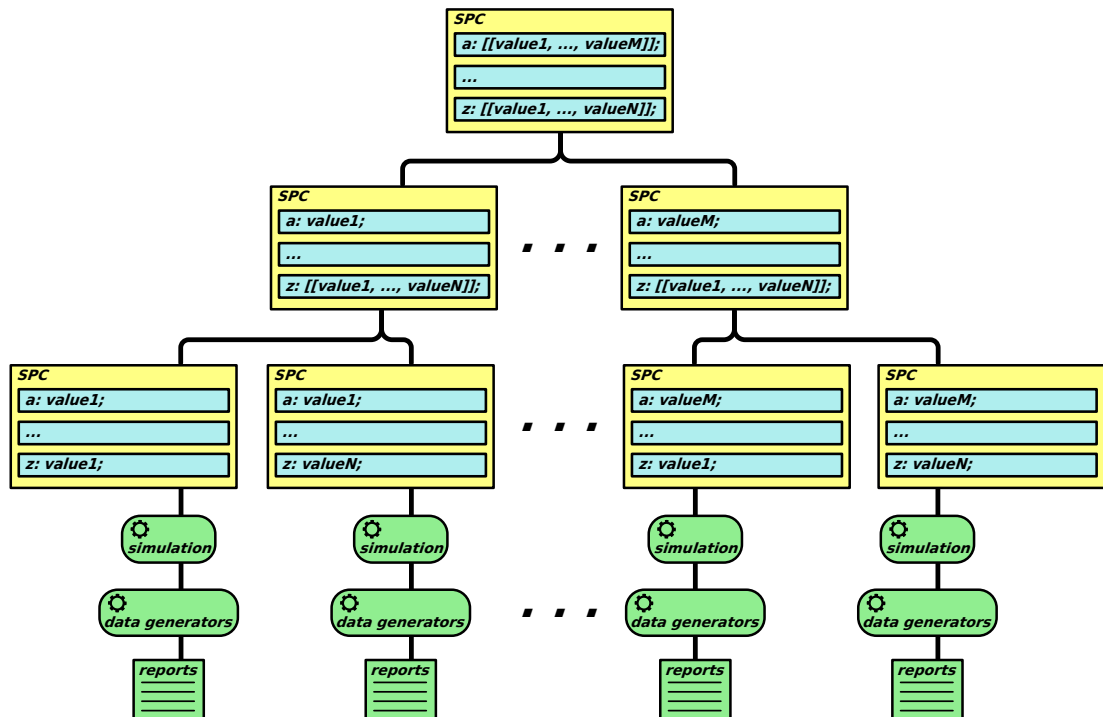


Figure 3.28: Graphical representation of branching.

3. CONFIGURATION LANGUAGE

Example 3.14. Use of the *branching operator*. After evaluation, a base configuration is split into two branches.

```
1 ...
2 x: [[1, 2]];
3 ...
1 /*
2  * Configuration branch 1:
3  */
4 ...
5 x: 1;
6 ...
1 /*
2  * Configuration branch 2:
3  */
4 ...
5 x: 2;
6 ...
```

3.9 Closing Remarks

The main goal of *SPC* is to efficiently support reproducible simulation experiments. This chapter has described the structure and grammar of *SPC*, together with examples. *SPC* has a human-readable format and allows configuring a model, a simulation and observers. Additionally, it enables to define the export of simulation results. Through the branching of configurations it is possible to set up the scanning of model parameters and simulation options. The branches of a configuration are loosely coupled (they only have common high-level/parent configuration) and can be executed in parallel. *SPC* supports adaptive stepwise simulation, which allows for reconfiguring model parameters based on the current status of a model and a simulation. All of these allow Spike to efficiently perform reproducible experiments.

Open Issues and Future Works. Even though the grammar of *SPC* is quite flexible, it lacks some features, which need to be addressed in future work:

- Full support of arrays - currently *SPC* supports only the declaration of arrays as they are used only to set values of some configuration options. Accessing of array elements will allow to reduce the number of declared variables and to collect and organize data in many useful ways.
- Conditional loop blocks - condition loops allow certain parts of a program to be run multiple times while a condition remains true. Support of a conditional loop block in connection with arrays will be very handy. This will facilitate the processing of data during stepwise simulation.

- Temporal logic - the temporal logic is focused on formulas that use temporal operators to describe how static conditions change over time. Support of the temporal logic syntax will allow to conveniently express how to alter a model after each simulation step, based on the current state of a model and a simulation.
- Parameter optimization - parameter optimization could be a complementary feature of parameter scanning. This will allow Spike to optimize a set of model parameters through an embedded optimization strategy.

The following chapter will explain some implementation aspects, and in Chapter 5 use cases will be discussed based on complete examples.

4

Spike Architecture

Spike [CH19] is a command line tool for an efficient execution of multiple simulation experiments of models, including biochemical reaction networks, represented as (coloured) \mathcal{PN} s and interpreted in the stochastic, continuous or hybrid paradigm. Simulation of biochemical models can be time and memory consuming. Thus, simulations should be delegated for performance reasons to be executed on a server. Additionally, when experiments require running multiple simulations, the time spent can be particularly long, when the individual simulations are merely executed one after another. Frequently, it is required to prepare a set of simulation experiments in order to find appropriate model parameters (e.g., initial conditions, kinetic parameters) or simulator options (e.g., simulator type, length of simulation traces, resolution of the traces recorded). Doing this manually, by preparing a new simulation run for each new model and/or simulator configuration, is time consuming and potentially error-prone. The reproducibility of the entire experiment is compromised, if one of the runs is not well documented.

Spike has been designed to address all these issues. It builds on a human-readable configuration script, supporting the efficient specification of multiple model configurations as well as multiple simulator configurations in a single file. Each specific model and simulator configuration determines a specific simulation experiment, for which Spike creates a separate branch, ready to be executed on a server, with all branches treated as parallel processes. Storing configurations in self-contained scripts allows for a simplified work-flow and reproducible simulations in a user-friendly manner.

Spike has a modular structure, where the modules are basically decoupled from each other. This allows to add new features easily. Modules communicate with each other using the command dispatcher pattern, which is globally accessible. Each module has its own list of commands with specific parameters. A command and its handler should be part of the same module. This allows adding or removing modules with minimal dependencies on each other. A command and its handler must be registered to the dispatcher during initializations of a module, see Figure 4.1. A command can be invoked outside its parent module and can be executed only by a handler associated with it. Invoked commands are processed sequentially.

4. SPIKE ARCHITECTURE

Example 4.1. Considering the following use case illustrated in Figure 4.2 -- the execution of a simple configuration script. When the command "exe" is invoked, the handler defined in the module *Configuration* will execute it. During execution, the configuration module communicates with other modules by invoking new commands.

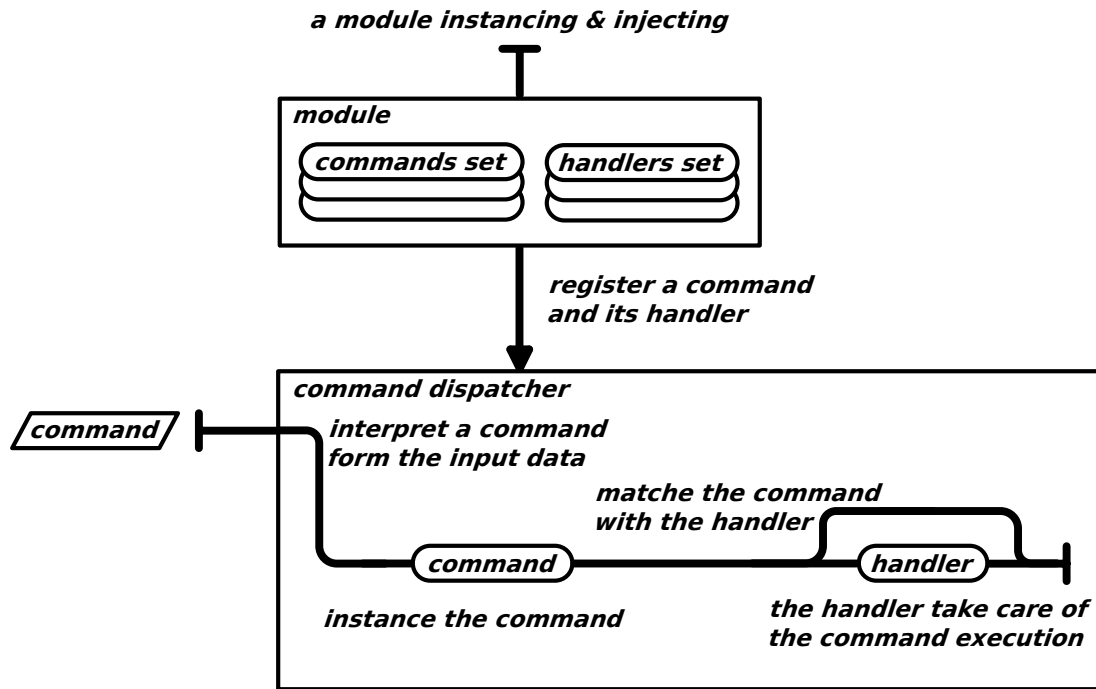


Figure 4.1: Graphical representation of command dispatching.

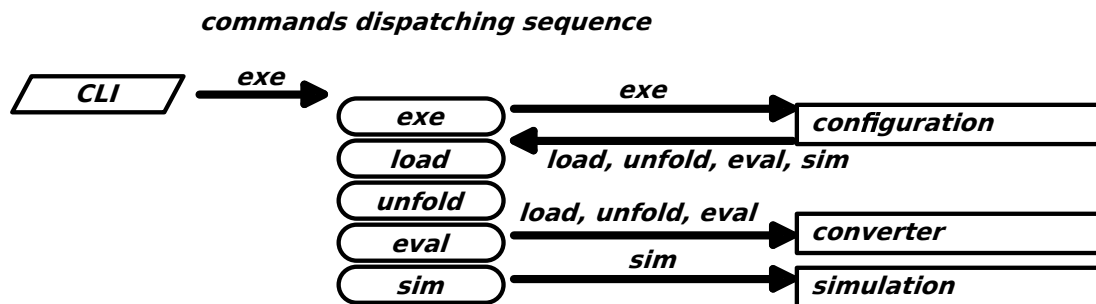


Figure 4.2: Flow of commands through Spike modules when a user types the command "exe".

Table 4.1 shows a summary of all commands currently available in Spike.

Table 4.1: List of Spike modules with their commands.

Module	Command	Description
Main	version	display version of Spike
CLI	help	display help for a given command
Configuration	exe	execute configuration script
Converter	load	load a model from a given file
	save	save a model to a given file
	prune	prune a model
	eval	evaluate constants
	unfold	unfold a coloured model
Simulation	sim	run a simulation of the model

4.1 Spike Functionality

Spike is a slim, but powerful brother of Snoopy [HHL+12]; it is the latest addition to the *PetriNuts* family of tools for modelling analysing and simulating a variety of related models, for which Petri nets are used as umbrella modelling paradigm. For more details see Figure 4.3.

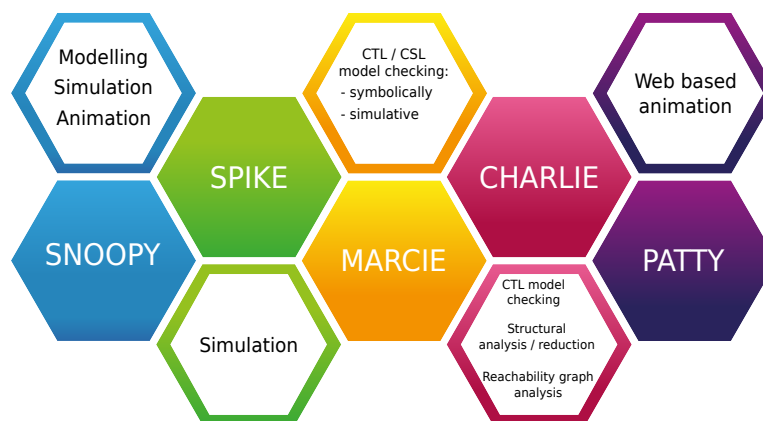


Figure 4.3: The *PetriNuts* framework consists of tools for modelling (Snoopy [HHL+12]), analysing (Marcie [HRS13], Charlie [HSW15]), simulating (Snoopy, Marcie, Spike [CH19]) and animating (Snoopy, Patty [Sch08]).

Spike deals with quantitative Petri nets, comprising stochastic, continuous and hybrid Petri nets, which are specifically tailored to the investigation of biochemical reaction networks. The Spike core features are presented in Figure 4.4 (page 80) and include: efficient and reproducible simulation experiments, the transformation between different exchange data formats and some basic model reductions.

4. SPIKE ARCHITECTURE

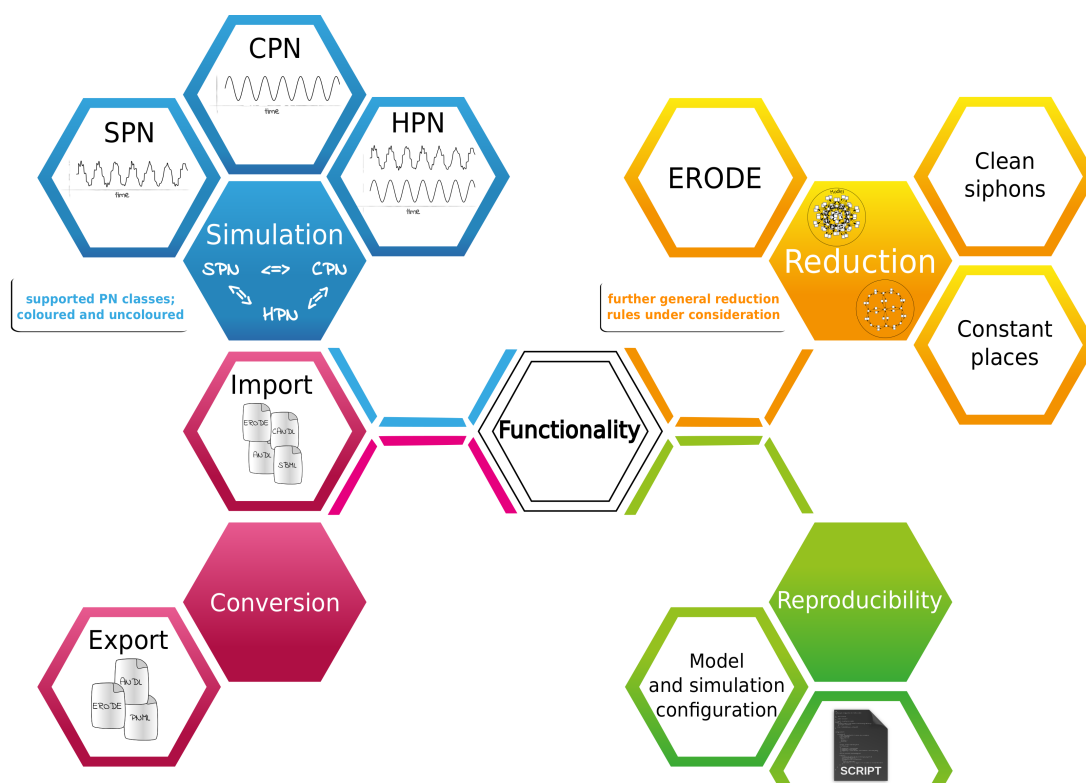


Figure 4.4: Overview of the Spike functionality, which includes efficient and reproducible simulation experiments, the transformation between different exchange data formats and some basic model reductions.

The Spike core features allow, among others, configuring the model (via parameters specifying arc weights, initial marking, kinetic parameters) and the simulator (via the usual, simulator-dependent options) over sets of arguments (parameter/option scanning). An argument is a value passed to a parameter or option. A set of argument sets triggers the so-called branching process. A new configuration branch is created for each argument set (if there is more than one). The set of configuration branches can be executed sequentially or in parallel. The simulation results can be saved in CSV files, which can be used later for analysis or visualization. They may comprise any user-defined combinations of traces over place markings, transition rates, and observers (auxiliary variables).

4.2 Simulation

The main focus of Spike lies in efficient and reproducible simulations. Depending on the configuration, Spike is able to run three basic types of simulations: stochastic, deterministic and hybrid [HLR+17], each comes with several algorithms.

Stochastic simulation follows basically the standard Gillespie algorithm; some algorithms apply approximation ideas for reasons of efficiency. All implementations are part of Snoopy's library of simulation algorithms:

- direct - Gillespie's stochastic simulation algorithm [Gil77],
- tauLeaping - τ -leaping [Gil01],
- deltaLeaping - δ -leaping [Roh17, Roh18],
- fau - fast adaptive uniformization [DHM+09, HRS+10, Roh17].

Deterministic simulation supports stiff/unstiff solvers ranging from simple fixed-step-size unstiff solvers (e.g. Euler) to more sophisticated variable-order, variable-step, multi-step stiff solvers (e.g. Backward Differentiation Formulas (BDF)). The ODE solvers BDF and ADAMS use the external library SUNDIAL CVODE [HBG+05]; all others are part of Snoopy's library of simulation algorithms:

- BDF - Backward Differentiation Formulas [HBG+05],
- ADAMS - Adams-Moulton [HBG+05],
- Classic - classical Runge-Kutta method (RK4) [VPT+02],
- RosenBrock - Rosenbrock method [VPT+02],
- Euler - Euler method (Runge-Kutta method, first order) [VPT+02],
- ModEuler - two-steps Euler method (Runge-Kutta method, second order) [VPT+02].

4. SPIKE ARCHITECTURE

Hybrid simulation allows for static or dynamic partitioning. In both cases, continuous transitions are simulated using an ODE solver, while stochastic transitions are simulated by the direct method of the Gillespie algorithm [HHL+12]. Static partitioning can be combined with `static`, `staticAcc`, `HRSSA`, or `HRSSAacc`. These are different strategies to synchronize the stochastic and continuous subnets. Dynamic partitioning always applies the exact method. The ODE solvers BDF and ADAMS use the external library SUNDIAL CVODE [HBG+05]; all others are part of Snoopy’s library of simulation algorithms:

- `static` - exact method [HR02, HH12],
- `staticAcc` - accelerated exact method [HH16],
- `HRSSA` - Hybrid Rejection-based Stochastic Simulation Algorithm [MPT16],
- `HRSSAacc` - accelerated HRSSA [HH18],
- `dynamic` - dynamic partitioning [HH12].

The simulation of stochastic, continuous and hybrid \mathcal{PN}^c models is supported by automatically unfolding them to their uncoloured counterparts.

A given model is simulated according to the specified simulation type, despite place and transition types in the model. That means all places and transitions are converted to the appropriate type. For example, if a user wants to run a stochastic simulation on a continuous model, all places and transitions are converted to the stochastic type. Likewise, for stochastic models to be simulated continuously, all stochastic transitions are converted to the continuous type, likewise for places.

4.3 Parallel Simulation

The evaluation of configuration may cause the split into separate branches A branching process is triggered by defining in the configuration a set of parameters to scan. The set of values is assigned to the configuration parameters. For each value in the set, a new configuration branch is created. Such a feature allows a configuration script to be split into separate branches, what results in multiple simulation configuration.

Example 4.2. The constant D has been defined. To set the size of the diffusion grid. With the help of the parameter scanning introduced in Spike, it is possible to reuse the same model and to set the range of values to scan for the constant D in the configuration script, e.g.:


```

1 ...
2 configuration: {
3   model: {
4     constants: {
5       all: {
6         D: [[3, 5, 7]];
7       }
8     }
9   }
10 }
11 ...

```

By using the *branching operator* `[[...]]`, the set of three values is assigned to the constant `D`. The number of branches depends on the size of the set. For each value in the set, Spike creates a new branch of the configuration script. In this case, Spike will split the configuration and create three branches.

The set of configuration branches can be executed sequentially or in parallel. Each branch is executed as a separate process of Spike. During running the simulation Spike creates two types of processes. One so-called master process and one or more worker processes. A high-level overview of performing parallel simulations is presented in Figure 4.5.

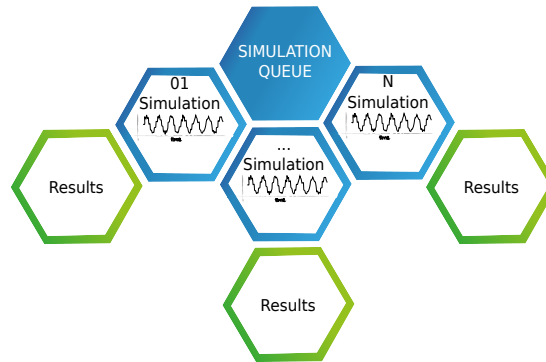


Figure 4.5: High level overview of performing parallel simulations. A master process orchestrates a queue of worker processes (simulations). Each simulation result is stored by a worker process.

4.4 Inter-Process Communication

Spike distinguishes two types of processes: the master and the worker. The master process acts as a broker that schedules the execution of simulation branches for the worker processes. The worker process is instantiated by the broker to execute a simulation task. After instantiating, a worker acts independently and communicates with the broker via network sockets [SFR03]. The communication is asynchronous and workers do not

4. SPIKE ARCHITECTURE

block each other while communicating with a broker. The *Boost.Asio* library [Koh21] was chosen for implementation as it is a cross-platform C++ library for network and low-level I/O programming. Despite the use of network sockets, Spike is currently only able to perform parallel simulation experiments on single host.

Figure 4.6, presents a simplified diagram of the life cycle of a broker and a worker process. After instantiating of Spike, the main process acts as the broker and the owner of the simulation experiment. The broker takes care of creating worker processes on a local machine. A worker process is responsible for executing exactly one branch of the simulation configuration. Depending on the option passed to Spike, a worker process can exit after finishing its job or can be reused. The reuse of a worker allows to speed-up initialization of a new simulation. It uses the resources acquired during instantiating and only needs to be initialized with a new configuration branch. The number of workers running in parallel depends on an option passed to Spike. If only one worker is allowed, then each simulation branch will be executed sequentially. In such a case, the broker waits for the worker to finish before outsourcing another configuration branch. Otherwise, the broker will instantiate workers up to the maximum number specified by the option of Spike. If the number of branches exceeds the number of workers, the broker will postpone outsourcing of the execution of the next branch until one of the currently running workers will finish its task.

The number of running worker processes is not equivalent to the number of running threads. The number of thread depends on the simulation algorithm. The stochastic simulation is an example, where the algorithm can be executed by utilizing multi-threading.

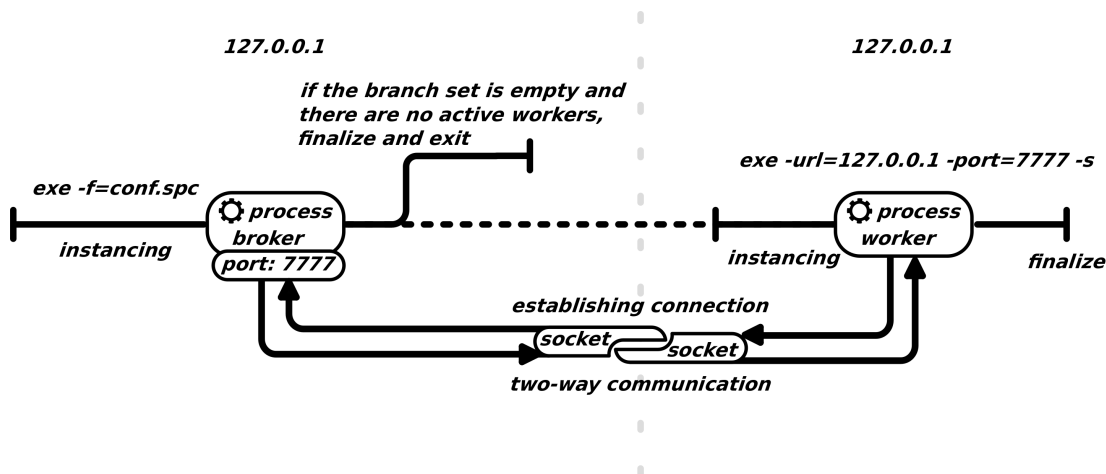


Figure 4.6: Simplified diagram of the life cycle of a broker and a worker process. Both processes are instantiated on the same local host. The broker process opens the default port 7777 of the local host (IP: 127.0.0.1) for inter-process communication. After establishing a connection with the worker process, the two-way inter-process communication begins.

The communication between a broker and a worker is done through networks sockets using a message pattern. When a message arrives at a receiver, a handler is invoked to process the message. The message format is as follows:

MSG#[DATA]::msg::end

and consists of four parts:

- MSG - is a string that names a message,
- # - a message-data separator,
- DATA - optional data,
- ::msg::end - marks end of a message.

Depending on the message, the DATA part can be optional or required. To mark this, the following convention is used:

- square brackets - [optional data],
- angle brackets - <required data>>,
- curly braces {default values},
- parenthesis (miscellaneous info).

The Table 4.2 contains the list of messages used in inter-process communication.

Table 4.2: List of messages.

Message	Description
GETCONF#::msg::end	request a configuration, sent from a worker to a broker;
GETCONF#<DATA>::msg::end	response with a configuration data on requests, sent from a broker to a worker;
SIMEND#::msg::end	sent to a broker notifies about finishing a simulation job, sent to a worker confirms end of a task and allows a worker to finalize;
LOG#<DATA>::msg::end	sent logging data to a logger;

Example 4.3. Figure 4.7 (page 86) presents a simplified scenario of an inter-process communication during the life cycle of a worker performing a simulation task. After instantiating, a worker asks a broker about the configuration. If a set of configuration branches to be executed is not empty, then the broker will respond and send a message with data that contains a configuration branch. After finishing a simulation task, the

4. SPIKE ARCHITECTURE

worker will send a message informing about the simulation end. If the queue is not empty and Spike is configured to reuse a worker process, the broker will send the next branch from the given set. Otherwise, a broker will send a message to the worker allowing to finish the work and to finalize. If the queue is not empty, a new worker process will be created.

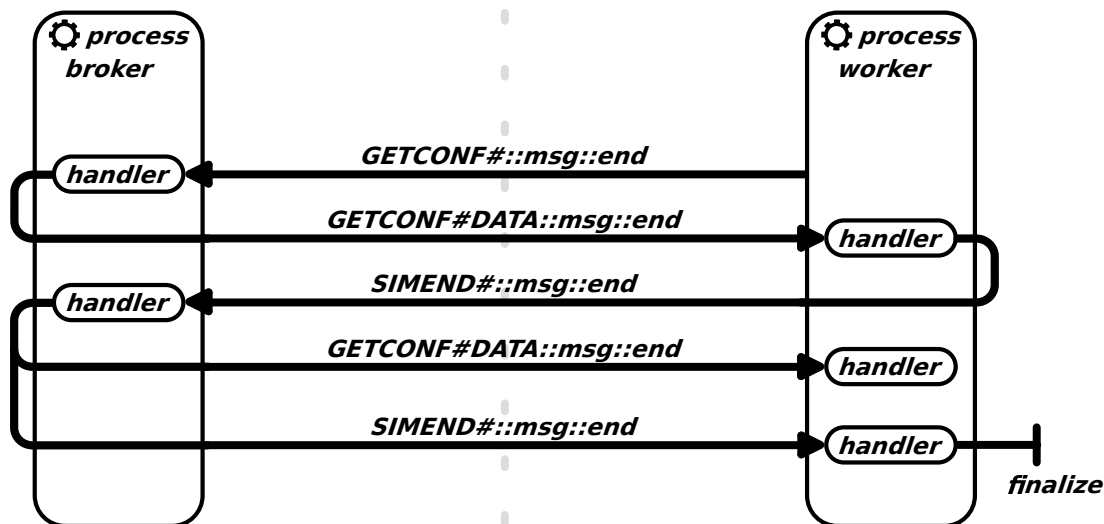


Figure 4.7: An example of an inter-process communication during the life cycle of a worker performing a simulation task.

4.5 Stepwise Simulation

The idea to implement a stepwise simulation arose from the need to introduce dynamic relaxation/constraint rules into the SIR model. The stepwise simulation allows for the dynamic adaptation of a model during the simulation runtime. It advances in a given time interval, in which the parameters of the model and its state can be adjusted/adapted after each simulation step. The adaptation is based on an evaluation of boolean conditions that may involve the current state of a model and the simulation.

The proposed algorithms: Algorithm 8 and Algorithm 9 allow reusing any simulator without its modification. The main idea is to apply a simulator for each time step. For each step, a simulator is reinitialized with the current state of a model and a constant time range. The time range corresponds to one simulation step and starts from zero and ends with a value of a time step size. After each simulation, the state of a model is stored and can be changed by predefined boolean conditions, which are evaluated for each time step.

Spike supports deterministic and stochastic stepwise simulation. The support of stepwise hybrid simulation is considered in future work.

Algorithm 8: Stepwise deterministic simulation algorithm.

Data: \mathcal{CPN} with initial state $X(\tau_0)$;
time interval $[\tau_0, \tau_{end}]$;
step size $\delta\tau$ where $\delta\tau < (\tau_{end} - \tau_0)$;

Result: trace of stored system states;

```

1 set of constants  $C = \emptyset$ ;
2 time  $\tau = \tau_0$ ;
3 state  $X(\tau) = \emptyset$ ;
4 while  $\tau < \tau_{end}$  do
5    $X(\tau) = \text{deterministic\_simulator}(\mathcal{CPN}, 0, \delta\tau, \delta\tau)$ ;
6    $X(\tau) \rightarrow \text{store}$ ; /* add  $X(\tau)$  to trace */
7    $X(\tau), C = \text{evaluate\_boolean\_conditions}(X(\tau), C)$ ;
8    $\text{set\_constants}(\mathcal{CPN}, C)$ ; /* apply new constant values to the model */
9    $\text{set\_makrings}(\mathcal{CPN}, X(\tau))$ ; /* apply new modified state to the model */
10   $\tau = \tau + \delta\tau$ ; /* determine next time point */
11 end

```

Algorithm 9: Stepwise stochastic simulation algorithm.

Data: \mathcal{SPN} with initial state $X(\tau_0)$;
time interval $[\tau_0, \tau_{end}]$;
step size $\delta\tau$ where $\delta\tau < (\tau_{end} - \tau_0)$;
number of runs R ;

Result: trace of stored system states;

```

1 run  $r = 0$ ;
2 average state  $A = \emptyset$ ;
3 while  $r < R$  do
4   set of constants  $C = \emptyset$ ;
5   time  $\tau = \tau_0$ ;
6   state  $X(\tau) = \emptyset$ ;
7   while  $\tau < \tau_{end}$  do
8      $X(\tau) = \text{stochastic\_simulator}(\mathcal{SPN}, 0, \delta\tau, \delta\tau)$ ;
9      $A(\tau) = A(\tau) + X(\tau)$ ;
10     $X(\tau), C = \text{evaluate\_boolean\_conditions}(X(\tau), C)$ ;
11     $\text{set\_constants}(\mathcal{SPN}, C)$ ; /* apply new constant values to the model */
12     $\text{set\_makrings}(\mathcal{SPN}, X(\tau))$ ; /* apply new modified state to the model */
13     $\tau = \tau + \delta\tau$ ; /* determine next time point */
14  end
15 end
16  $A = A/R$ ;
17  $A \rightarrow \text{store}$ ;

```

4.6 Reproducible Stochastic Simulation

Random number generator play a crucial role in any stochastic simulation. In order to guarantee the reproducibility of a stochastic simulation, the simulation library used internally by the *PetriNuts* framework had to be modified in a such way, that each of the simulation threads produce unique, reproducible results based on a main seed for all random number generators.

The process of instantiating the simulation threads is from Spike's point of view as follows. The main thread is configured with a seed and a number of threads, which are set in a configuration script. The seed initializes a random number generator which is used by the main thread. The main thread creates a pool of random seeds and a pool of threads. Each time when the random number generator of main thread is initialized with the same seed, it ensures to generate the same pool of seeds. The size of both pools is equal to the number of threads defined by a configuration. Each seed can be assigned to only one thread from the pool. A pair of seed/thread defines a simulation, which is added to the pool of simulation threads. After initializing, simulation threads are executed in parallel. The graphical representation of this process is represented in Figure 4.8.

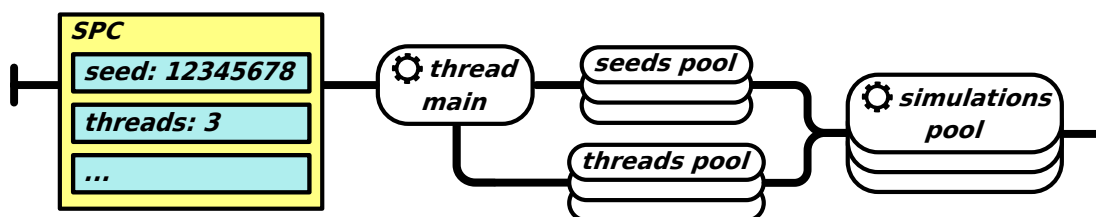


Figure 4.8: Graphical representation of instantiating the simulation threads.

4.7 Conversion

Spike supports the following data formats and conversion between them, as shown in Figure 4.9:

- ANDL and CANDL - human-readable formats for Petri nets and coloured Petri nets, respectively, used internally by the *PetriNuts* framework,
- SBML (Systems Biology Markup Language) - an XML-based representation format designed to exchange computational models within the systems biology community [Huc15],
- PNML - an XML-based interchange format for qualitative Petri nets [PNML] used within the Petri net community,

- ERODE - a tool for the evaluation and reduction of chemical reaction networks [CTT+17].

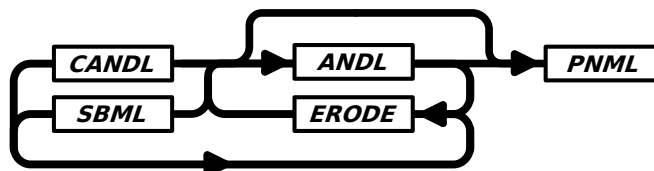


Figure 4.9: Data format conversions supported by Spike. Please note that any node in this diagram can be the entry point. For the PNML format, only import is allowed and no further conversions are possible.

4.8 IDD-based unfolding

Spike uses *IDD* (Interval Decision Diagrams) to efficiently unfold \mathcal{PN}^c [SRF+20]. *IDD*, first proposed in [LR95], belong to the symbolic data structures and can be seen as a generalization of the popular Binary Decision Diagrams (*BDD*). *BDD* are widely used to encode boolean functions, while *IDD* encode interval logic functions. Interval logic functions are boolean expressions involving atomic predicates defining integer intervals, e.g.: $x_1 \in [6, 8)$, $x_2 > 0$.

IDD are Directed Acyclic Graphs (*DAG*) with two types of nodes -- terminal and non-terminal ones. There are two terminal nodes (typically represented as boxes), labelled with 0 and 1, and the non-terminal nodes (typically represented as circles or ellipses) are labelled with the variables occurring in the interval logic function to be encoded. Non-terminal nodes may have an arbitrary number of outgoing arcs labelled with intervals of natural numbers (including zero) partitioning the set of natural numbers. Intervals have the form $[a, b)$; where the lower bound a is included in the interval $[a, b)$ and the upper bound b not. Note that intervals of the form $[a, \infty)$ are allowed as well, see Figure 4.10 for two examples.

4. SPIKE ARCHITECTURE

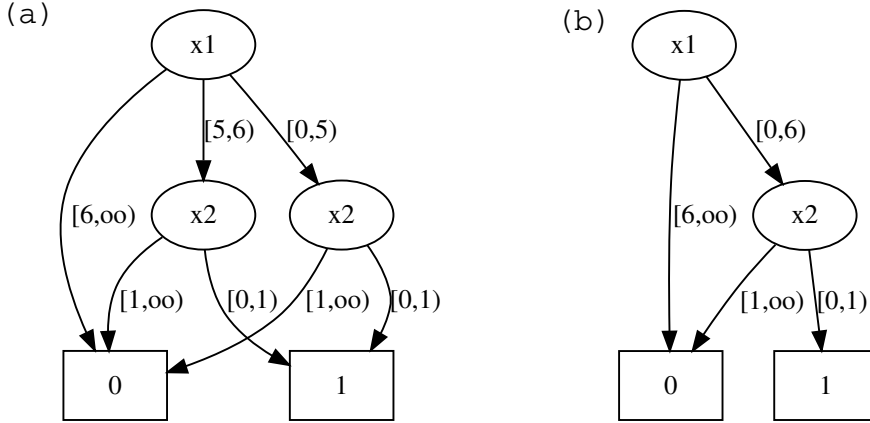


Figure 4.10: Two *IDDs* representing $f = (x_1 \in [0, 6] \wedge x_2 = 0)$; (a) not reduced; (b) reduced.

4.8.1 IDD Reduction

Reduced Ordered Interval Decision Diagrams (*ROIDD*) are a canonical representation for interval logic functions and often provide a compact representation in many application areas. An *IDD* is called reduced, if three 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 subgraphs.

Applying those rules to the *IDD* in Figure 4.10.(a) yields the reduced version presented in Figure 4.10.(b). The reduction is carried out by merging the two nodes labelled with x_2 (third rule) and merging/reduction of redundant arcs (first rule).

The variable ordering can have an impact on the size of a *ROIDD*. Finding an optimal ordering is generally infeasible, and even checking if a particular ordering is optimal is NP-complete [BW96, RK08]. There exist interval logic functions, that have *ROIDD* representations of exponential size for any variable ordering. Heuristics, taking into consideration that variables which depend on each other should be close together in the ordering, often bring good results.

4.8.2 Unfolding

The unfolding engine utilized by Spike uses shared *ROIDDs*, an implementation principle to keep several *ROIDDs* within one data structure. Technically speaking, a shared *ROIDD* is a single multi-rooted DAG representing a collection of interval logic functions. All functions in the collection must be defined over the same set of variables, using the same variable ordering. Thanks to the canonicity of *ROIDDs*, two functions

in the collection are identical if and only if the *ROI*DDs representing these functions have the same root in the shared *ROI*DD. The unfolding proceeds basically in three steps (according to [SRF+20]):

1. *Unfolding of coloured places* -- generates for each coloured place as many unfolded places as there are 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 $p\text{-}c$ 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 any isolated unfolded places* -- colours that are never used yield isolated places, which will never influence the net behaviour, even if initially holding tokens; thus they can be safely removed.

The first and last step are relatively easy. The core problem of efficient unfolding is to determine the transition instances, i.e. all bindings of values to the variables involved, potentially enabling coloured transitions. Fortunately, each coloured transition t can be considered separately, and the problem can be formulated as a constraint satisfaction problem (CSP), defined by:

- *the set of variables* -- all variables occurring on any arc adjacent to transition;
- *the domain of each variable* -- given by its (finite, discrete) colour set;
- *the constraints* -- any guards involved, which are all Boolean expressions.

To solve the CSP, a corresponding *IDD*, which represents the constraints (the so-called *constraint IDD*), is built stepwise bottom-up. First, the domain of each individual variable is represented as *IDD*; the only colour set type causing here problems is *union*. Next, the constraint *IDD* is constructed using standard *IDD* algorithms. The set of all paths going from the root to the terminal node 1 describes all solutions of the given constraint problem; typically, one path encodes more than one solution. Thus, all CSP solutions can be easily picked from the constraint *IDD*.

To deal with variables of union type, all different data types subsumed by the union type, each yielding one constraint *IDD*, need to be considered alternatively. In other words: if two variables of a union type subsuming three types (colour sets), the solution is obtained by considering nine constraint *IDD*s.

Guards, which may be arbitrarily complex, may not only serve as transition guards, but also help to conveniently define colour sets as subsets of previously defined colour

4. SPIKE ARCHITECTURE

sets or to specify the initial marking in a concise and scalable way. Both need to be considered when unfolding places. Likewise, guards also permit to specify colour-dependent transition rate functions or conditional colour expressions for arcs.

4.8.3 Algorithms

This section sketches an implementation of the *IDD* unfolding engine by a pseudocode description; see Algorithms 10--13.

Algorithm 10. The main procedure of the *IDD* unfolding engine follows the basic steps outlined in Subsection 4.8.2. Before unfolding the coloured places (line 18) and unfolding the coloured transitions (line 19), all colour-related net annotations have to be registered (lines 6--17). This comprises four categories of declarations: constants, colour sets, variables, and colour functions. Constants are crucial to design scalable and easily adjustable coloured Petri nets; thus they are often used in colour sets and colour functions.

The actual unfolding happens in Algorithms 11 and 12, which involves setting up and solving a CSP for every place and every transition, respectively. This is here done by the help of *IDDs*, but could be equally achieved by any other appropriate data structure. Algorithm 11 creates unfolded places, but does not add them to the unfolded net. Algorithm 12 creates unfolded transitions and their unfolded adjacent arcs and does indeed add them to the unfolded net. Afterwards, all unfolded places, which are involved in the unfolding of transitions, are actually added to the unfolded net in the final step (lines 20--24), which implicitly prunes the unfolded net by ignoring isolated places.

Algorithm 11. The unfolding of places can be done place by place and requires determining all colours of a place's colour set. Thus, the computational load for this unfolding step depends on the kind of colour sets supported. Colour sets known by the *PetriNuts* framework include the following.

- *Dot sets.* A *Dot* set contains one so-called *black* colour and is defined by a set of one constant value: *dot*.
- *Boolean sets.* A *Boolean* set is defined by a set of two Boolean constants: *true* and *false*.
- *String sets.* Are based on strings of characters surrounded by quotation marks, i.e.: "...". A string colour set is specified by a set of single elements, and may incorporate the usual set operations.
- *Integer sets.* Are based on natural numbers. An integer colour set can be specified by a set of single elements or valid ranges, and may incorporate the usual set operations.
- *Enumeration types* are treated as integer sets, where all elements are given by constants.

Algorithm 10: Unfold CPN

```

1 Net unfoldedNet;
2 placeRefTable  $\subset$  String  $\times$  Int  $\times$  Int =  $\emptyset$ ; /* (name, tokens, number of references) */
3 Environment env; /* some kind of registry */
4
5 proc unfoldNet (CPN net)
6   forall c  $\in$  net.constants do
7     env.registerConstant(c.name, c.expr);
8   end
9   forall cs  $\in$  net.colorsets do
10    env.registerColorset(cs.name, cs.expr);
11  end
12  forall v  $\in$  net.variables do
13    env.registerVariable(v.name, v.colorset);
14  end
15  forall cf  $\in$  net.color functions do
16    env.registerColorFunction(cf);
17  end
18  unfoldPlaces(net); /* Algorithm 2 */
19  unfoldTransitions(net); /* Algorithm 3 */
20  forall (place, tokens, ref)  $\in$  placeRefTable do
21    if ref > 0 then
22      unfoldedNet.addPlace(place, tokens);
23    end
24  end
25 end

```

- *Product sets*. Building on previously defined colour sets more complex, compound colour sets can be defined by means of the Cartesian product.
- *Subsets*. Given a previously defined colour set, it is possible to select specific elements characterised by a Boolean expression (*guard*). These guards are treated as implicit guards during the unfolding (line 4).

The computation of all colours for the colour set of a given place is achieved by constructing an IDD for the solution space (lines 8, 12). The solutions are obtained by following all paths to the IDD's terminal node 1 (supported by a corresponding iterator concept); each solution generates an unfolded place (lines 18-22).

The creation of unfolded places includes the generation of their initial marking according to the given marking expression (lines 6-11). Places which remain empty are created afterwards (lines 12-13). Please note, places are created, but not added yet to the unfolded net.

Algorithm 12. The unfolding of transitions can be done transition by transition and requires determining all variable bindings for every transition. To set up the corresponding CSP, the algorithm first iterates over all adjacent arcs (line 4), which are

4. SPIKE ARCHITECTURE

Algorithm 11: Unfold Places

```

1 proc unfoldPlaces (CPN net)
2   forall  $p \in \text{net.places}$  do
3     /* replace function call by its body guard used to describe subsets */
4     substituteColorFunctions( $p.markingExpr, env$ );
5     Guard  $g_{cs} = env.implicitGuard(p.colorset)$ ;
6     Set  $G_p = \{g_{cs}\}$ ;
7     /* separated by '++' */
8     forall  $expr \in p.markingExpr$  do
9       Set vars = collectVariables( $markingExpr, env$ );
10      IDDSolutionSpaceRepr S( $vars, expr.guard, env$ );
11      createPlaces( $p, S, expr.value, expr.color$ );
12       $G_p = G_p \cup \{expr.guard \cap g_{cs}\}$ ;
13    end
14    /* remaining places are empty */
15    IDDSolutionSpaceRepr S( $vars, \bigcap_{g \in G_p} \neg g, env$ );
16    createPlaces( $p, S, 0, expr.color$ );
17  end
18 proc createPlaces (Place p, IDDSolutionSpaceRepr S, ColExpr value, ColExpr color)
19   forall  $sol \in S$  do
20      $place_s = createPlace(p, color, sol, env)$ ;
21      $value_s = createValue(value, sol, env)$ ;
22      $placeRefTable = placeRefTable \cup \{(place_s, value_s, 0)\}$ ;
23   end
24 end

```

grouped into *conditions* (read arcs, inhibitory arcs, equal arcs, reset arcs) and *updates* (standard arcs connecting pre- and post-places). A transition guard may be additionally restricted by the implicit guards of any adjacent places with a subset colour set. Thus, those implicit guards have to be collected (lines 5--7). Next, all variables involved in any adjacent arc or transition guard are collected (line 9), which then permits to create the IDD representation of the solution space of the given CSP (line 10).

Next, the CSP solutions are evaluated by iterating over the solution space, following all paths to the *IDD*'s terminal node 1 (lines 11--23). Every solution generally generates a set of arcs, whereby the unfolding of arcs always preserves the arc type; a coloured read arc will always be unfolded to read arcs. If there are no arcs for a given CSP solution, no unfolded transition is created (line 20--22).

Unfolded places will be ignored in Algorithm 10, if they are never connected to any transition. Thus, the entry in the *placeRefTable* is updated by removing the previous tuple and adding a new tuple with the number of references (usage of this place) increased by 1 (line 33).

Algorithm 12: Unfold Transitions

```

1 proc unfoldTransitions (CPN net)
2   forall  $t \in \text{net.transitions}$  do
3     Guard  $G_a = \emptyset$ ;
4     /* preparation step */
5     forall  $(p, \text{arcType}, \text{arcExpr}) \in t.conditions \cup t.updates$  do
6       /* guard used to describe subsets */
7       Guard  $g_p = \text{env.implicitGuard}(p.colorset)$ ;
8       /* replace function call by its body */
9       substituteColorFunctions(arcExpr, env);
10       $G_a = G_a \cup \{\text{arcExpr.guards} \cap g_p\}$ ;
11    end
12    Set vars = collectVariables( $t.conditions \cup t.updates \cup t.guard$ , env);
13    IDDSolutionSpaceRepr  $S(\text{vars}, t.guard \cap (\bigcup_{g \in G_a} g), \text{env})$ ;
14    /* creation step */
15    forall  $sol \in S$  do
16      Set arcs;
17      forall  $(p, \text{arcType}, \text{arcExpr}) \in t.conditions \cup t.updates$  do
18        arc = createArc( $p, \text{arcType}, sol, \text{arcExpr.guard}$ ,
19           $\text{arcExpr.value}, \text{arcExpr.color}$ );
20        if arc  $\neq \text{null}$  then
21          arcs = arcs  $\cup \{\text{arc}\}$ ;
22        end
23      end
24      if arcs  $\neq \emptyset$  then
25        unfoldedNet.addTransition(createTransition( $t, sol, \text{env}$ ), arcs);
26      end
27    end
28  end
29
30 proc createArc (Place  $p$ , ArcType  $\text{arcType}$ , Solution  $sol$ , Guard  $\text{guard}$ , ColExpr  $\text{value}$ ,
31   ColExpr  $\text{color}$ )
32   /* guard used to describe subsets */
33   Guard  $g_p = \text{env.implicitGuard}(p.colorset)$ ;
34   if  $sol \models \text{guard} \cap g_p$  then
35      $\text{place}_s = \text{createPlace}(p, \text{color}, sol, \text{env})$ ;
36      $\text{value}_s = \text{createValue}(\text{value}, sol, \text{env})$ ;
37      $\text{placeRefTable} = \text{placeRefTable} - \{(\text{place}_s, \text{value}_s, n)\} \cup \{(\text{place}_s, \text{value}_s, n+1)\}$ ;
38
39     return Arc( $\text{place}_s, \text{arcType}, \text{value}_s$ );
40   else
41     return null;
42   end
43 end

```

4. SPIKE ARCHITECTURE

Algorithm 13: IDD solution space representation

```

1 class IDDSolutionSpaceRepr
2   IDD solutions;
3   proc constructor (Set vars, Guard guard, Environment env)
4     createVariableOrder(vars, guard);
5     solutions = 1;                                     /* the universe */
6     /* create the potential solution space */
7     forall  $v \in vars$  do
8       Intset is = env.getIntColorset(v);
9       solutions = solutions  $\cap$  makeIDD(is, env);
10    end
11    /* create the actual solution space */
12    solutions = solutions  $\cap$  makeIDD(guard, env);
13  end
14  proc makeIDD (Guard guard, Environment env)
15    if  $g \equiv g_1 \wedge g_2$  then return makeIDD(g1, env)  $\cap$  makeIDD(g2, env) ;
16    if  $g \equiv g_1 \vee g_2$  then return makeIDD(g1, env)  $\cup$  makeIDD(g2, env) ;
17    if  $g \equiv \neg g_1$  then return  $1 - \text{makeIDD}(g_1, \text{env})$  ;
18    if  $g \equiv f_1 \circ f_2 : \circ \in \{=, \neq, \leq, <, >, \geq\}$  then
19      Set vars = collectVariables(f1)  $\cup$  collectVariables(f2);
20      IDD S =  $\emptyset$ ;                                     /* empty set */
21      forall  $v \in vars$  do
22        Intset is = env.getIntColorset(v);
23        S = S  $\cup$  makeIDD(is, env);
24      end
25      return ExtractAP(S, g);
26  end

```

Algorithm 13. This pseudocode is a data structure and provides the algorithm to construct an *IDD* representing the solution space for a given CSP, characterized by a set of variables and a guard in the context of the coloured net to be unfolded. The algorithm starts with choosing the variable order (line 4). A good variable order often depends on the specific guard involved; thus the guard occurs as parameter and is evaluated by the procedure *createVariableOrder* to determine, which variables are close to each other. Next, the potential solution space is constructed by combining the colour sets of all variables involved (lines 6--9), which is afterwards restricted to the actual solution space by considering the guard (line 10). The actual *IDD* construction (lines 12--25) follows the standard *IDD* algorithms, see [ST11, Sch14]. The sub-procedure *ExtractAP* (Extract atomic proposition, line 23) extracts all states in *S* fulfilling the guard *g*, represented as *IDD*; for details see [Sch14], Algorithm 6.

The developed implementation is equipped with an iterator enabling the efficient

iteration over all solutions, which is used in Algorithms 11 and 12. As a special feature, the iterator automatically updates the environment only with regard to changed variable values.

This algorithm is not *IDD*-specific. The type *IDD* could be replaced just by some set type and the algorithm will work. Although *IDDs* often yield a very compact representation of sets and permit very efficient manipulation algorithms, it may be worth considering explicit or other symbolic data structures.

4.8.4 The *elemOf* Operator and Boolean Colour Set

The implemented *IDD* unfolder supports expressions that may involve the *elemOf* operator. It checks the membership of a certain colour in a colour set, which returns true if the colour is a member of the colour set, otherwise it returns false. An expression that uses the *elemOf* operator can be assigned to constrain transitions, arcs and colour-dependent rate functions. If the *elemOf* operator is applied to a subset of a colour set, then a given expression can be substituted by an explicit constrain, that defines the subset and the *elemOf* operator is applied on the main colour set. The new expression is combined by the logical operator $\&$, e.g.:

for a given colour set and its subset

$$\begin{aligned} \text{enumPopulation} &= \{A, B, C, D, E, F\}; \\ \text{PopulationAB} &= \text{Population}[x = A || x = B]; \end{aligned}$$

the expression

$$x \text{ elemOf PopulationAB}$$

is equivalent to

$$(x = A || x = B) \ \&\& \ x \text{ elemOf Population} .$$

Drawbacks of the implementation. Currently, the implementation of the *elemOf* operator has the following drawbacks:

- An auxiliary variable is implicitly created for each colour set of the right-hand operand. An auxiliary variable has an impact on the size of the calculated *IDD*, as is shown in Figure 4.14 (page 104), and slows down its evaluation.
- If the *elemOf* operator is applied on a subset of a colour set, then a given expression is substituted by an explicit constraint that defines the subset and the *elemOf* operator is applied on the main colour set. Such a substitution is inefficient and

4. SPIKE ARCHITECTURE

may add extra non-terminal nodes to a resulting *IDD*. The process of such a substitution is presented in Figure 4.14.(c), (f) (page 104) and the resulting *IDD* in Figure 4.14.(g) (page 104).

- Only one *elemOf* operator is allowed in an expression as the substitution of such expression

$$x \text{ elemOf } PopulationAC \ \&\& \ x \text{ elemOf } PopulationAB$$

where

$$\begin{aligned} PopulationAB &= Population[x = A || x = B]; \\ PopulationAC &= Population[x = A || x = C]; \end{aligned}$$

is unreliable and results in

$$((x = A || x = C) || (x = A || x = B)) \ \&\& \ x \text{ elemOf } Population$$

where the constraints of the two subsets are implicitly joined by the operator $||$.

Example 4.4. The model in Figure 4.11 (page 99) is a variation of the model in Figure 2.6 (page 18). It presents a more flexible solution to specify colour-dependent rate functions. In this model the colour set *Population* is extended by new species: *C*, *D*, *E* and *F*. It contains one subpopulation *PopulationAB* defined as a subset of the colour set *Population*. The colour-dependent rate functions are defined with the help of the *elemOf* operator, which allows to assign a specific rate function based on a subset of the population, e.g.: the following expression

$$\begin{aligned} [x \text{ elemOf } PopulationAB] &: MassAction(k_infect[x]) \\ [x \text{ elemOf } PopulationF] &: MassAction(k_infect_A + k_infect_B) \end{aligned}$$

states that, if the value of the variable *x* belongs to the subpopulation *PopulationAB*, then the reaction fires with the rate specified by the function

$$MassAction(k_infect[x]),$$

where $[...]$ is the indexing operator. With the help of the indexing operator, kinetic parameters defined as constants can be accessed by a value of defined colours. The indexing operator requires the definition of constants with a colour value at the end of a constant name separated by an underscore, e.g: *k_infect_A* where *A* is the colour

value. If the value of the variable x belongs to the subpopulation $PopulationF$, then the reaction fire with the rate specified by the function

$$MassAction(k_infect_A + k_infect_B) .$$

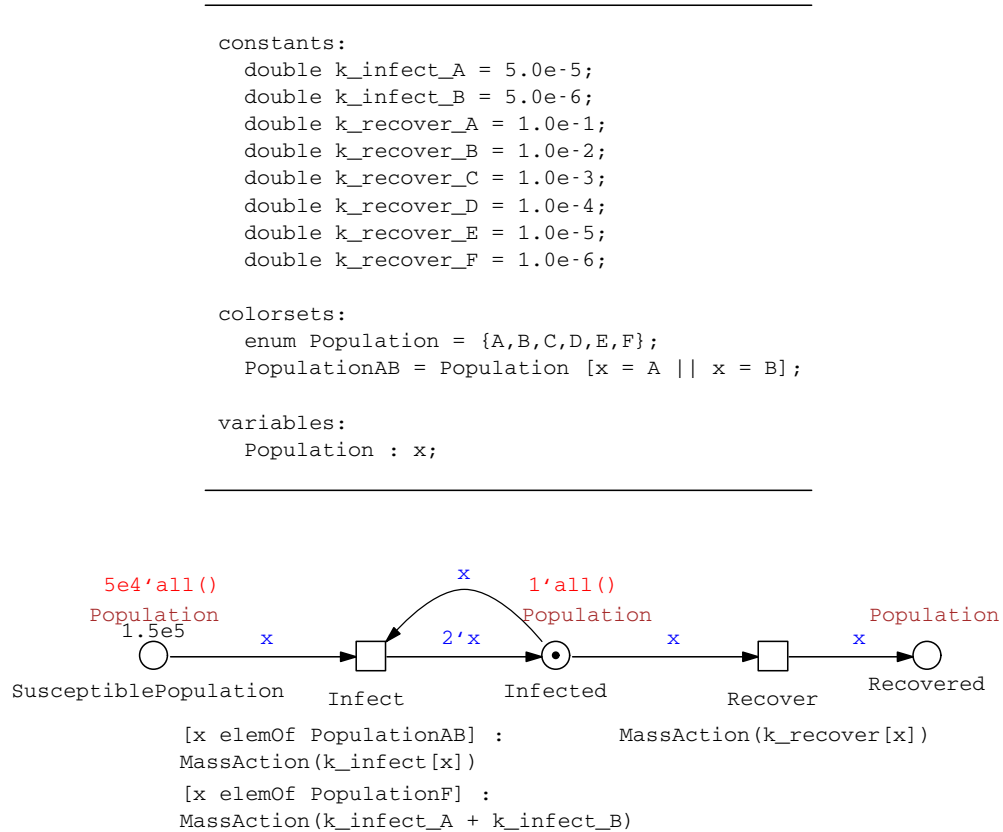


Figure 4.11: Coloured SPN SIR model with a more flexible solution to specify colour-dependent rate functions.

Example 4.5. The model in Figure 4.12 (page 100), is a variation of the model in Figure 4.11 (page 99). It presents a more advanced use of the *elemOf* operator. In addition to the previous example, the *elemOf* operator is used to constrain transitions and the model contains two additional subpopulation: $PopulationF$ and $PopulationABF$. The transition *Infect* is constrained and accepts species from the subpopulation $PopulationABF$ what is expressed by the expression

$$x \text{ elemOf } PopulationABF .$$

In this case, the transition will accept and fire only for the species A, B and F . The

4. SPIKE ARCHITECTURE

transition *Recover* is also constrained by the expression

$$x \text{ elemOf } \textit{PopulationAB} ,$$

and accepts species that belong to the subpopulation *PopulationAB*. The model comprises also one additional transition *Recover_F*, which fires only if the colour value of the variable *x* belongs to the *PopulationF*, what is defined by the constraint expression

$$x \text{ elemOf } \textit{PopulationF} .$$

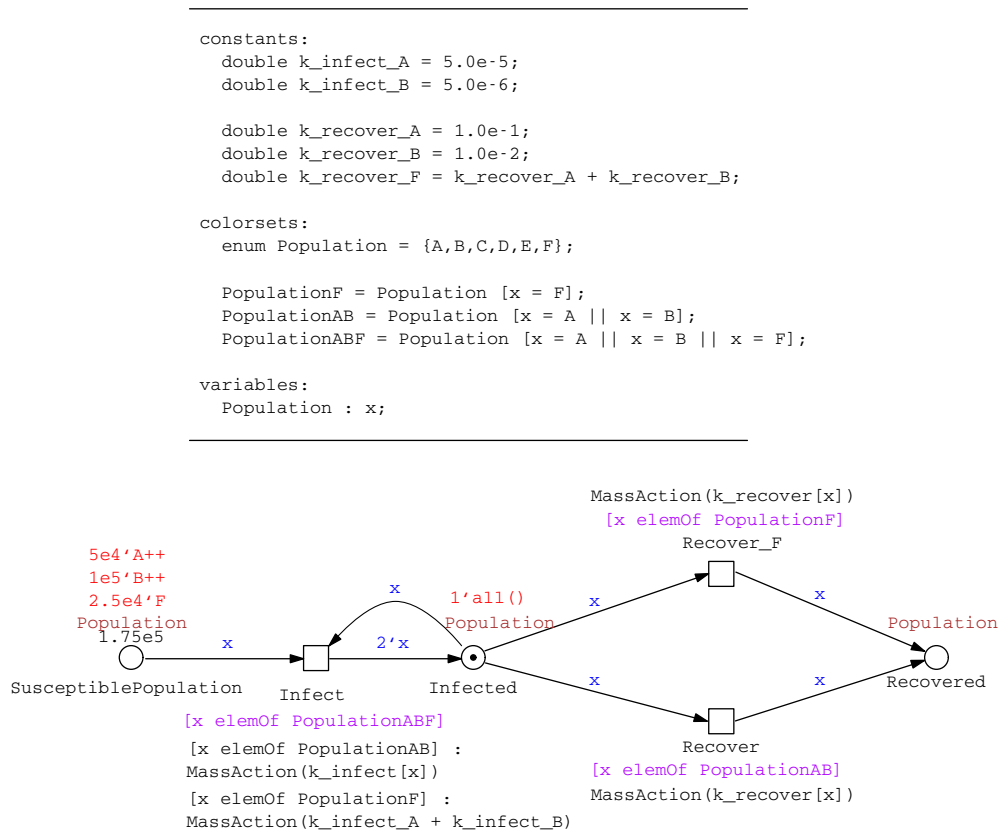


Figure 4.12: SIR model, as coloured *SPN* driven by *elemOf* expressions.

In coloured Petri nets each token has a colour and thus each place must have a colour set. The newly introduced *Boolean* colour set extends the implemented *IDD* unfold engine. The *Boolean* colour set is defined by a set of coloured values represented by two Boolean constants: *true* and *false*; it supports the following boolean operators:

& - AND - denoted $x&y$;

| - OR - denoted $x|y$;

! - NOT - denoted $!x$.

The denoted expression values can be expressed by a truth table, see Table 4.3.

Table 4.3: The truth table of *IDD* Boolean expression.

x	y	$x&y$	$x y$	$!x$
0	0	0	0	1
1	0	0	1	0
0	1	0	1	1
1	1	1	1	0

Example 4.6. The use of the *Boolean* colour set can simplify colour expressions, which can be used to control the firing of transitions. The model in Figure 4.13 (page 102) is a variation of the model in Figure 4.12 (page 100). In this model one additional place *AllowInfect* is introduced. The place is of type *Boolean* and with the help of Boolean b it ensures sequential occurrence of the *Infection* and *Recovery* process of an individual population. It is a form of control of a mutual exclusion process, where firing of the transition *Infect* is followed by firing of the transition *Recover* for a single population.

Example 4.7. The stepwise computation of the constraint *IDD* to find all instances for the single transition *Recover_F* is documented in Figure 4.14 (page 104) and comprises the following steps:

- (a) Encoding the entire enumeration colour set $Population = \{A, B, C, D, E, F\}$ automatically involves a mapping of the enumerated constants to integer identifiers; thus, the integer identifier 0 represents *A*, 1 stands for *B*, 2 for *C*, 3 for *D*, 4 for *E* and 5 for *F*.
- (b) Encoding the entire colour set *Boolean*, specified by the list of the boolean constants $\{true, false\}$, automatically involves a mapping of boolean constants to integer identifiers; thus, the integer identifier 0 represents *false*, 1 stands for *true*.
- (c) Constraining the colour *Population* given in (a) to $x \text{ elemOf } Population$, which is represented by the implicit auxiliary variable *aux*.

4. SPIKE ARCHITECTURE

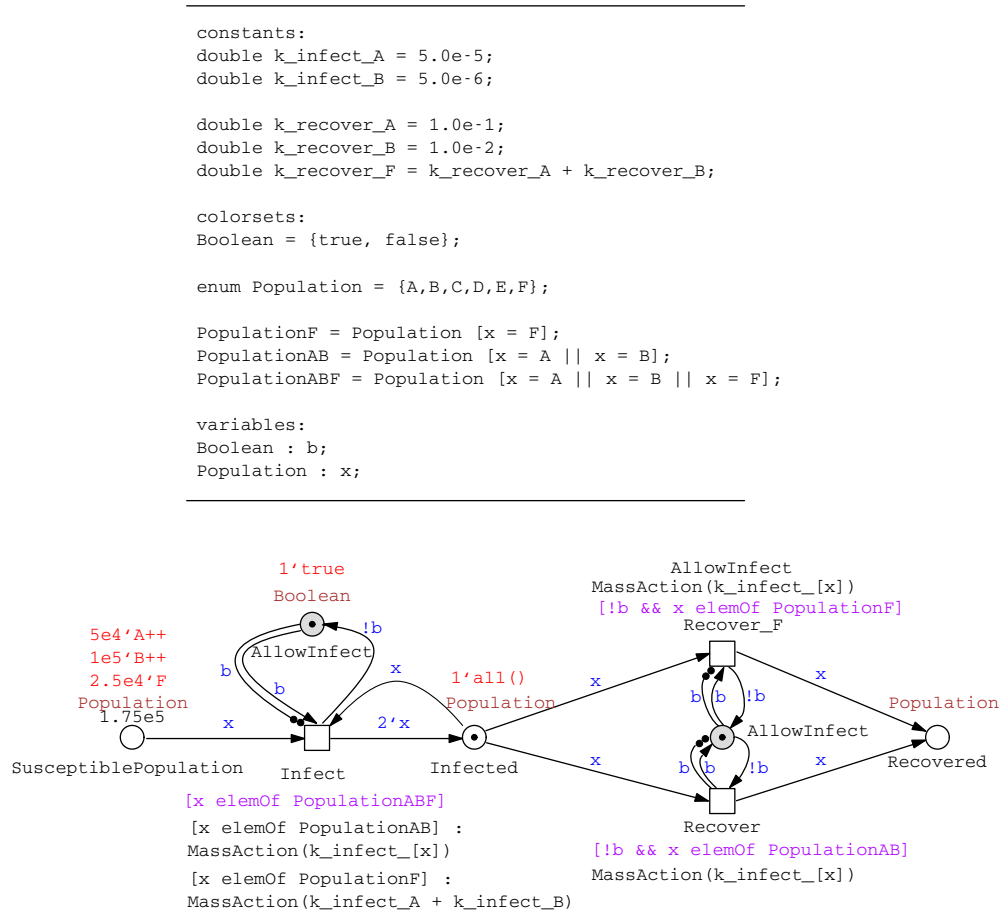


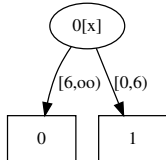
Figure 4.13: SIR model, as coloured *SPN* driven by *elemOf* expressions and *Boolean* colour set. The palce *AllowInfect* is of type *Boolean* and is represented by two logical places given in grey. A set of logical places, having the same name, refers to one and the same place; it serves only to simplify the representation of the model.

- (d) Constraining the colour set *Boolean* given in (b) to $b = false$ yields the subrange comprising false.
- (e) Combining (c) and (d) by the logical operator $\&$ yields the *IDD* with the solution space containing all elements of the entire colour set *Population* and the variable b set to *true*.
- (f) Constraining the colour *Population* given in (a) to $x = F$ yields the subrange comprising the single value F , represented by its identifier 5.
- (g) Combining (e) and (f) by the logical operator $\&$ yields the final result that defines all possible bindings for the transition *Recover_F* according to its guard $!b \ \&\& \ x \ \text{elemOf} \ PopulationF$. There is one path going to the terminal node 1. It encodes the value binding $(false, F)$, giving one uncoloured transition instance for the coloured transition *Recover_F*.

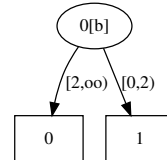
All *IDDs* in Figure 4.14 (page 104) have been generated by a logging mechanism integrated in the unfolding engine for debugging purposes and visualized with Graphviz [GN00]. Non-terminal nodes are labelled with the variable index and the variable name (in square brackets).

4. SPIKE ARCHITECTURE

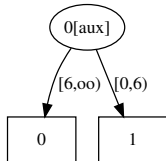
(a) Population = {A,B,C,D,E,F}



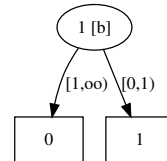
(b) Boolean = {true, false}



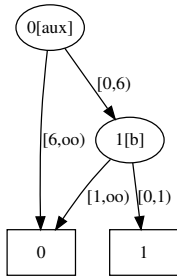
(c) x elemOf Population



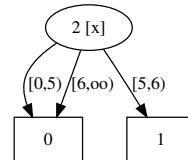
(d) b = false



(e) !b && x elemOf Population



(f) x=F



(g) !b && (x=F) && x elemOf Population ≡ !b && x elemOf PopulationF

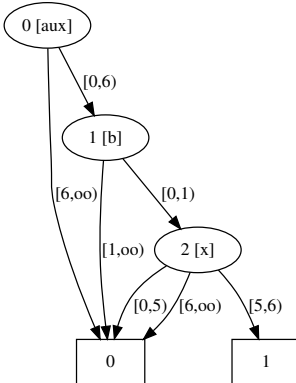


Figure 4.14: Stepwise IDD computation to find all instances (bindings) for the single transition *Recover_F* of the model in Figure 4.13. The constrain expression $!b \ \&\& \ x \ \text{elemOf} \ \text{PopulationF}$ is substituted by the expression $!b \ \&\& \ (x = F) \ \&\& \ x \ \text{elemOf} \ \text{Population}$.

4.9 Closing Remarks

Spike is an efficient tool for the reproducible execution of parallel simulation experiments of biochemical reaction networks. This chapter has described the main architecture of Spike. The modular structure and the mechanism of intermodule communication allows to easily extend Spike by new modules. The main functionalities of Spike allow importing and exporting \mathcal{PN} models in various formats. An imported coloured model can be unfolded using *IDD*-based unfolding, which is integrated in the internally developed `dssd_util` library used by Snoopy, Marcie and Spike. During the work on Spike the *Boolean* colour set and the *elmeOf* operator were introduced into *IDD*-based unfolding. The newly introduced colour set and operator allow simplifying coloured expressions. To perform a simulation, Spike uses an internally developed simulation library; it is able to run three basic types of simulations: stochastic, deterministic and hybrid, where each comes with several algorithms. Spike is supported by a scripting language (*SPC*), which allows for designing reproducible simulation experiments that can be executed in parallel. Additionally, *SPC* allows the execution of a simulation in a stepwise manner.

Open Issues and Future Works. Spike can be improved in many ways, as it lacks some features that should be addressed in future work:

- Model reduction - Spike allows for the basic reduction of a \mathcal{PN} model. It is able to structurally reduce a model by pruning clean siphons and constant places. However, this basic reduction methods are insufficient. The growing amount of experimental data and expressive power of the colour annotations leads to the development of complex models. A complex model represented by \mathcal{PN}^c needs to be unfolded before its simulation.

After unfolding, the number of nodes can be much larger than in its coloured counterpart. Reduction of a model may yield a more optimized (in terms of size) model, provide insights into structural properties and reduces a simulation overhead. The main challenge of a reduction is to preserve the main three properties of a \mathcal{PN} model: liveness, reversibility and boundedness. The two simplest techniques that preserve the main three properties are pruning of clean siphons and constant places.

- Model decomposition - decomposition of \mathcal{PN} model into basic subnets. Decomposition can be done by network structure or through type, if the \mathcal{PN} is hybrid. The process of clustering should be aided through manual selection / specification of cluster set as well as through an automatic / algorithm approach. The model decomposition will allow for distributed simulation of the decomposed model. Such functionality should speed up the simulation of large models - more research needs to be done to get a clear answer.

4. SPIKE ARCHITECTURE

- Distributed simulation - Spike is able to perform parallel executions of simulation experiments on single host. Future work should consider implementation of distributed simulation, which can speed-up the execution of an experiment in the following example cases:
 - (a) - a simulation experiment contains a set of exhaustive simulations - in this case each simulation can be distributed over a network of computing peers, where each peer performs a single simulation.
 - (b) - a parallel simulation of a decomposed model - in a such case each component of the model is distributed over a network of computing peers, where each peer performs a single, parallel, synchronized simulation for the received model component.
- Parameter optimization - Optimization through a simulation can be used as a search method [CM97] for the best candidates of input variables among all valid alternatives at any system state. By adopting heuristic evaluation, it is possible to reduce a search space without explicitly evaluating each possibility. Spike features such as parameters scanning and parallel execution of configuration branches make Spike suitable for this task. However, all these features are not sufficient to perform parameter optimization. Future work should consider embedding the optimization strategy directly into Spike.

The following chapter discusses use cases with complete examples that illustrate most of the functionalities of *SPC*.

5

Use Cases

To illustrate the functionality of Spike and its configuration script language *SPC* (introduced in Chapter 3) the following three use cases are discussed:

1. Benchmarking - designing of benchmarking experiments;
2. Simulation of adaptive models - stepwise simulation;
3. Spike as a backend simulator for simulative parameter optimization;

5.1 Benchmarking

Depending on the configuration, a given model is simulated according to the specified simulation type, regardless of the model type. Such functionality allows to design benchmarking experiments, with the main goal to compare the computational complexity of models and / or the performance of the simulation algorithms. A benchmarking experiment can be designed in the following two ways:

1. to firmly compare the performance of different simulation methods using a well characterized comparative set of models,
2. to firmly compare the computational complexity of models (representing the same system) using a set of simulations.

This allows to determine the strengths of each simulation method or model, respectively.

Essential information about designing of benchmarking experiments are provided in [WSC+19] where a set of guidelines is introduced. Based on this, the following guidelines should be considered when designing a benchmarking experiment:

- Define the purpose and scope of the benchmark - how comprehensive the benchmark should be.

5. USE CASES

- Select (or design) representative data sets - number and types of data sets to be included.
- Choose appropriate parameter values - amount of tuning parameters.
- Evaluate methods according to key quantitative performance metrics - numbers and types of performance metrics.
- Interpret results and provide recommendations - generality versus specificity of recommendations.
- Follow best practices for reproducible research, by making code and data publicly available.

The examples below show that designing a benchmarking experiment is relatively easy with Spike, which has the ability to scan its configuration options.

Example 5.1. To benchmark simulation types over a given model, the following scenario represented by Algorithm 14 can be used.

Algorithm 14: Use case: Benchmarking of simulation types.

```
1 Load model;
2 Determine model configuration;
3 for each simulation type do
4   Determine simulation configuration;
5   Create new configuration branch;
6   Run simulation;
7   Store results;
8 end
```

The following implementation of this scenario is intended to compare the deterministic and stochastic simulations results of the SIR model (see Figure 2.6, page 18). The scenario comprises four main steps:

- (a) Specification of the model source; lines 2 – 4.
- (b) Specification of the simulation name. The name depends on the named import and the type of simulation; line 10. It is used to define the unique names of the files, which contain the simulation results.
- (c) Declaration of the simulation type list. The list contains the configuration of two types of the simulation: stochastic and continuous (deterministic), for each of them a new branch of the simulation configuration will be created and a separated simulation performed. In the configuration of the stochastic simulation, the seed value is specified to reproduce the simulation traces presented in Figure 5.1; lines 15 – 48.

- (d) Configuration of how to store the simulation results. It allows specifying which the simulation traces are to be recorded - in this case, all paces related to population B. The name of the resulting file depends on the variables defined in the configuration, what allows to generate a unique name; lines 51 – 62.

```

1 // Import model
2 import: {
3   from: "./model/SIR-SPNC.cand1";
4 }
5
6 configuration: {
7
8   simulation:
9   {
10    name: "BENCHMARK1:" << import.name << "_" << type; // Name of a simulation
11    /*
12     * Branching:
13     * Scanning over simulation types
14     */
15    type: [[
16      // Stochastic simulation
17      stochastic: {
18        solver:
19        direct: {
20          threads: 1;
21          runs: 50;
22          // reproducing stochastic simulation resulting traces
23          seed: 2589244515;
24        }
25        single: true;
26        avg: true;
27      },
28      // Continuous Simulation
29      continuous: {
30        solver:
31        BDF: {
32          /*
33           * Define new variable "runs" that
34           * is used in the export file name
35           */
36          runs: "CONT";
37          semantic: "adapt";
38          iniStep: 0.1;
39          linSolver: "CVDense";
40          relTol: 1e-5;
41          absTol: 1.0e-10;
42          autoStepSize: false;
43          reductResultingODE: true;
44          checkNegativeVal: false;
45          outputNoiseVal: false;
46        }
47      }
48    ]];
49
50    interval: 0:200:50;

```

5. USE CASES

```
51   export: {
52     // Array of places to save (if empty export all)
53     places: ["*_B.*"]; // []; // all places
54     //transitions: []; // all transitions
55     csv: {
56       sep: ";"; // Separator
57       file: "./data/"
58         << name
59         << "_" << configuration.simulation.type.solver.runs
60         << ".csv"; // File name
61     }
62   }
63
64 }
65 }
```

It is worth noting the difference between the resulting traces in Figure 5.1.(a) and 5.1.(b) (page 111). This can be explained by decay of a disease when all specimen in an infected population recovered without spreading a disease (isolation) onto a susceptible population. This event cannot be clearly seen in the traces of the stochastic simulation, as they show the averaged results of 50 runs. It is similar in the case of the deterministic simulation. This event is superseded, since the traces are results of approximating the solutions of $ODEs$. To spot this event, it is necessary to look on a single run of a stochastic simulation. The Figure 5.1.(c) (page 111) presents the resulting trace of the single run selected from the set of the stochastic simulation runs. The selected 33th run explains the notable difference between resulting trace. It shows decay events of a disease (when all specimen in an infected population have recovered) in the populations.

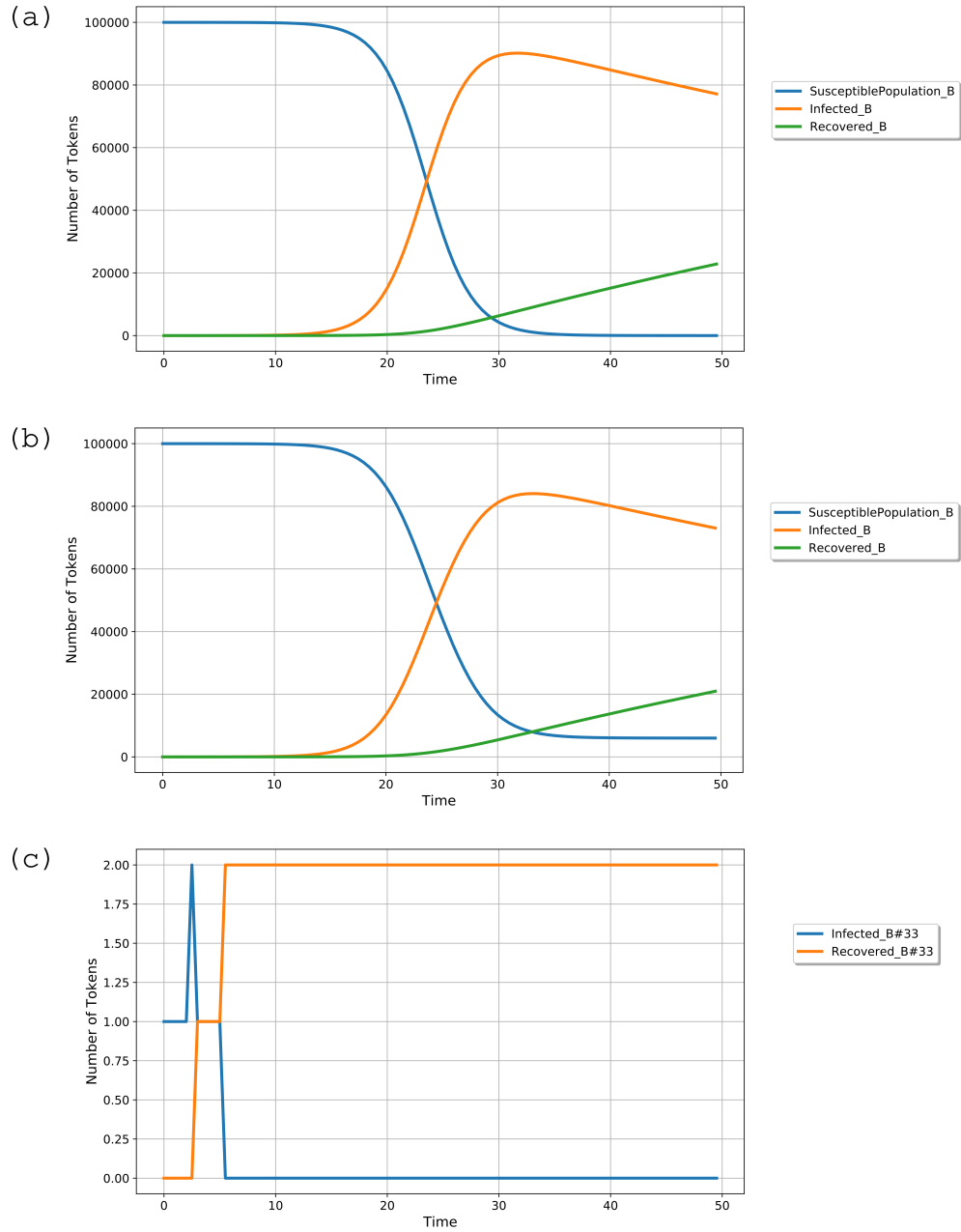


Figure 5.1: Comparison of simulation traces of the model in Figure 2.6, which are related to the population B; the deterministic (continuous) simulation (a) versus average of 50 stochastic simulation runs (b) and one of the single runs of the averaged stochastic simulation (c); where $SusceptiblePopulation$ is set to $5e4A + 1e5B$; $k_{infect_a} = 5.0e - 5$; $k_{infect_b} = 5.0e - 6$; $k_{recover_a} = 1.0e - 1$; $k_{recover_b} = 1.0e - 2$. The selected 33th run explains the notable difference between resulting traces. It shows decay events of a disease (when all specimen in an infected population have recovered) in the populations.

5. USE CASES

Example 5.2. The following scenario represented by Algorithm 15 can be used to benchmark models over one simulation type.

Algorithm 15: Use case: Benchmarking of simulation models.

```
1 for each model do
2   Load model;
3   Determine model configuration;
4   Determine simulation configuration;
5   Create new configuration branch;
6   Run simulation;
7   Store results;
8 end
```

The following scenario implementation is intended to compare the SIR model (see Figure 5.2.(a), page 114) with its SEIR extension (see Figure 5.2.(b), page 114). SEIR is an extension of the SIR model with one additional compartment representing population of exposed species (E), which are infected but not yet infectious. Upon being infected, individuals will move to this sub-population and remain there for an incubation period before moving to the infected population.

The scenario comprises four main steps:

- (a) Declaration of a list of models to import. The list contains the configuration of two named imports: *a* and *b*, for each of them a new branch of the simulation configuration will be created and a separated simulation performed; lines 5 – 12.
- (b) Specification of the simulation name. The name depends on the named import; line 18. It is used to define the unique names of the files, which contain the simulation results.
- (c) Declaration of simulation type. lines 19 – 27.
- (d) Configuration of how to store the simulation results. It allows specifying which the simulation traces are to be recorded. The name of the resulting file depends on the variables defined in the configuration, what allows to generate a unique name; lines 31 – 44.

```

1  /*
2  * Branching:
3  * Scanning over models to import
4  */
5  import: [[
6    SIR:{
7      from: "./model/SIR-SPNC.candl";
8    },
9    SEIR:{
10     from: "./model/SEIR-SPNC.candl";
11   }
12  ]];
13
14  configuration: {
15
16    simulation:
17    {
18      name: "BENCHMARK:" << import ; // Name of a simulation
19      type:
20        // Stochastic simulation
21        stochastic: {
22          solver:
23          direct: {
24            threads: 1;
25            runs: 3;
26          }
27        }
28
29      interval: 0:200:100;
30
31      export: {
32        // Array of places to save (if empty export all)
33        places: []; // [] // all places
34        // transitions: [] // all transitions
35        csv: {
36          sep: ";"; // Separator
37          file: "./data/"
38            << name << "-"
39            << configuration.simulation.type << "-"
40            << configuration.simulation.type.solver
41            << "-" << configuration.simulation.type.solver.runs
42            << ".csv"; // File name
43        }
44      }
45    }
46  }
47 }

```

Both models are configured with the same set of constants, with one exception that the constant $k_{incubation}$ is used in the SEIR model as the kinetic parameter of the *Incubation* transition. The results of the simulation are presented in Figure 5.3. It can be seen that the additional compartment that represents exposed species, slows down the spread of the disease since during this period, species in the incubation state do not take in the infection process making it is less violent.

5. USE CASES

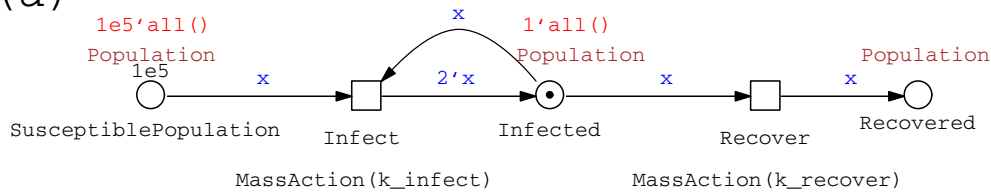
```

constants:
  double k_infect      = 5.0e-5;
  double k_recover    = 1.0e-1;
  double k_incubation = 1.0e-1;
  int pop_size = 1;

colorsets:
  enum Population = {1..pop_size};

variables:
  Population : x;
  
```

(a)



(b)

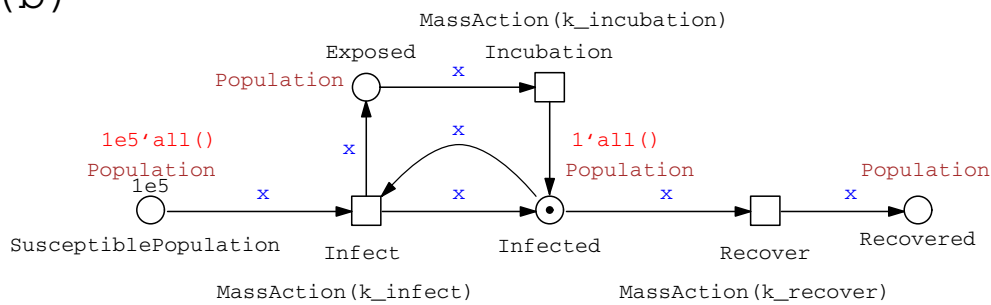


Figure 5.2: The SIR (a) and the SEIR (b) model used to simulate the spread of the disease. The SEIR model extends the SIR model by one additional place *Exposed* and the transition *Incubation*. Those two additional nodes create the compartment representing the population of exposed species (E), which are infected but not yet infectious.

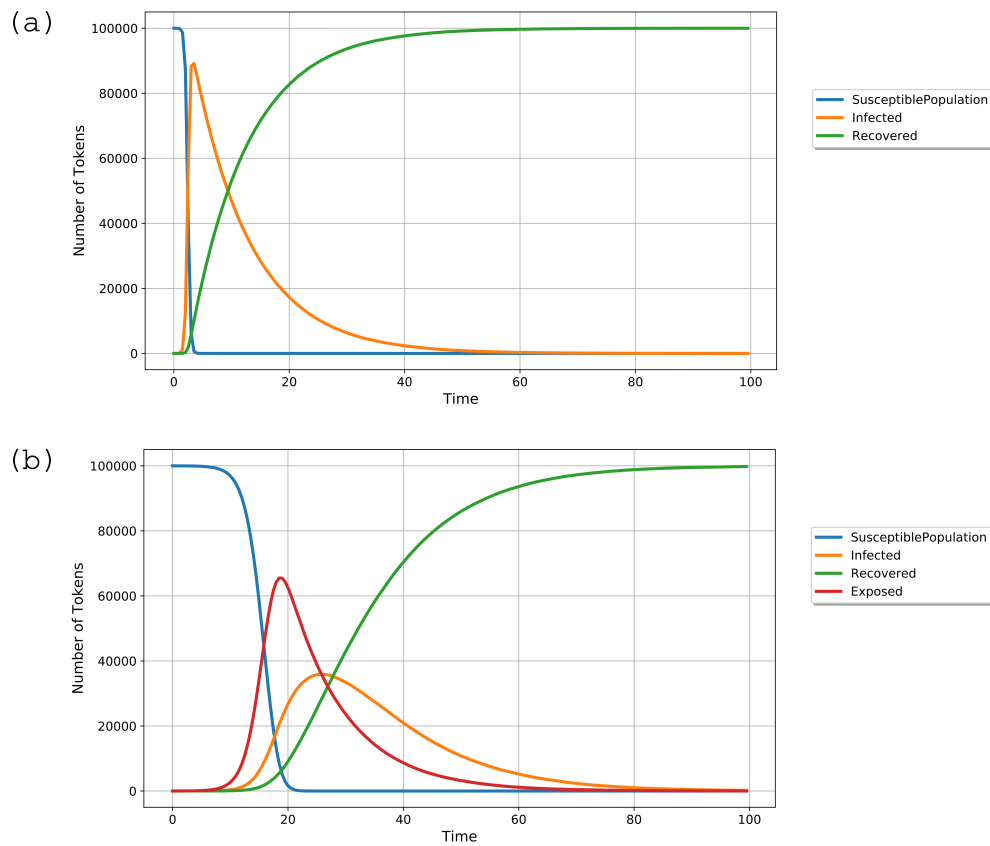


Figure 5.3: Simulation results of the SIR and the SEIR model presented in Figure 5.2.(a),(b), respectively. As it can be seen, the additional compartment in the SEIR model that represents exposed species, slows down the spread of the disease since during this period, species in the incubation state do not take in the infection process making it less violent.

5. USE CASES

Example 5.3. The following scenario represented by Algorithm 16 comprises both scenarios from the two previous examples. It allows to benchmark a set of models over a given set of simulation types. A separated simulation will be performed for each unique combination of *model – simulation type*.

Algorithm 16: Use case: Benchmarking of simulation types and models.

```
1 for each model do
2   Load model;
3   Determine model configuration;
4   for each simulation type do
5     Determine simulation configuration;
6     Create new configuration branch;
7     Run simulation;
8     Store results;
9   end
10 end
```

The following implementation of this scenario compare the performance of hybrid simulation algorithms. It is a simplified version of the experiment presented in [HH18]. The scenario comprises four main steps:

- (a) Declaration of a list of models to import. The list contains the configuration of two named imports: *HPNC2* and *HPNC*, for each of them a new branch of the simulation configuration will be created and a separated simulation performed; lines 2 – 9;
- (b) Adjust the model parameters; lines 12 – 18.
- (c) Specification of the simulation name. The name depends on the type of simulation algorithm; line 21. It is used to define the unique names of the files, which contain simulation results.
- (d) Declaration of the simulation algorithm list. The list contains the configuration of four hybrid simulation algorithms: *static*, *staticAcc*, *HRSSA* and *HRSSAacc*. For each of them a new branch of the simulation configuration will be created and a separated simulation performed; lines 27 – 89.
- (e) Configuration of how to store the simulation results. It allows specifying, which the simulation traces are to be recorded. The name of the resulting file depends on the variables defined in the configuration, what allows to generate a unique name; lines 94 – 105.

```
1 // Import - exactly one model
2 import: [[
3   HPNC2: {
4     from: "./model/SIR-HPNC2.candl";
5   },
6   HPNC: {
7     from: "./model/SIR-HPNC.candl";
8   }
9 ]];
10
11 configuration: {
12   model: {
13     constants: {
14       all: {
15         pop_size: 2;
16       }
17     }
18   }
19   simulation:
20   {
21     name: "BENCHMARK:" << import << ":" << type.solver; // Name of a simulation
22     /*
23     * Branching:
24     * Scanning over simulation algorithms
25     */
26     type: hybrid: {
27       solver: [[
28         static: {
29           threads: 1;
30           runs: 3;
31           odeSolver: "BDF";
32           iniStep: 0.1;
33           linSolver: "CVDense";
34           relTol: 1e-5;
35           absTol: 1.0e-10;
36           autoStepSize: true;
37           reductResultingODE: true;
38           checkNegativeVal: false;
39           outputNoiseVal: false;
40         },
41         staticAcc: {
42           threads: 1;
43           runs: 3;
44           odeSolver: "BDF";
45           iniStep: 0.1;
46           linSolver: "CVDense";
47           relTol: 1e-5;
48           absTol: 1.0e-10;
49           autoStepSize: true;
50           reductResultingODE: true;
51           checkNegativeVal: false;
52           outputNoiseVal: false;
53         },

```

5. USE CASES

```
54     HRSSA: {
55         threads: 1;
56         runs: 3;
57         odeSolver: "BDF";
58         iniStep: 0.1;
59         linSolver: "CVDense";
60         relTol: 1e-5;
61         absTol: 1.0e-10;
62         autoStepSize: true;
63         reductResultingODE: true;
64         checkNegativeVal: false;
65         outputNoiseVal: false;
66         //
67         fluctRatio: 0.2; // Fluct ratio
68         // Apply monitored places
69         applyMonPlaces: true;
70     },
71     HRSSAacc: {
72         threads: 1;
73         runs: 3;
74         odeSolver: "BDF";
75         iniStep: 0.1;
76         linSolver: "CVDense";
77         relTol: 1e-5;
78         absTol: 1.0e-10;
79         autoStepSize: true;
80         reductResultingODE: true;
81         checkNegativeVal: false;
82         outputNoiseVal: false;
83         //
84         fluctRatio: 0.2; // Fluct ratio
85         // Apply monitored places
86         applyMonPlaces: true;
87         applyInterfacePlaces: true;
88     }
89 ];
90 }
91
92 interval: 0:200:500;
93
94 export: {
95     // Array of places to save (if empty export all)
96     places: [];//[];// all places
97     transitions: ["Travel.*"];// all transitions
98     csv: {
99         sep: ";";// Separator
100        file: "./data/"
101            << name << "_"
102            << configuration.simulation.type.solver.runs
103            << ".csv";// File name
104    }
105 }
106 }
107 }
```

For this simple benchmark experiment two hybrid models are used. Each of them is a specially tailored variation of the SIR model. The models are statically partitioned. Model in Figure 5.4.(a) (page 119) contains two disjoint clusters, not connected by any interface transitions. Clusters have the form of the separated \mathcal{CPN}^C and the \mathcal{SPN}^C SIR model. The model in Figure 5.4.(b) (page 119) consists of two clusters: the

deterministic and stochastic clusters are connected by one interface transition *Recover*. The experiment demonstrates (see Table 5.1) that for a statically partitioned model with absence of interface, reactions the accelerated *HRSSA* algorithm (*HRSSAacc*) indeed improves the hybrid simulation performance. The number of interface reactions influences the performance of hybrid simulation. Their firings affect the system state of the deterministic regime. For each interface reaction the ODE solver needs to be reinitialised when one of these reactions occurs.

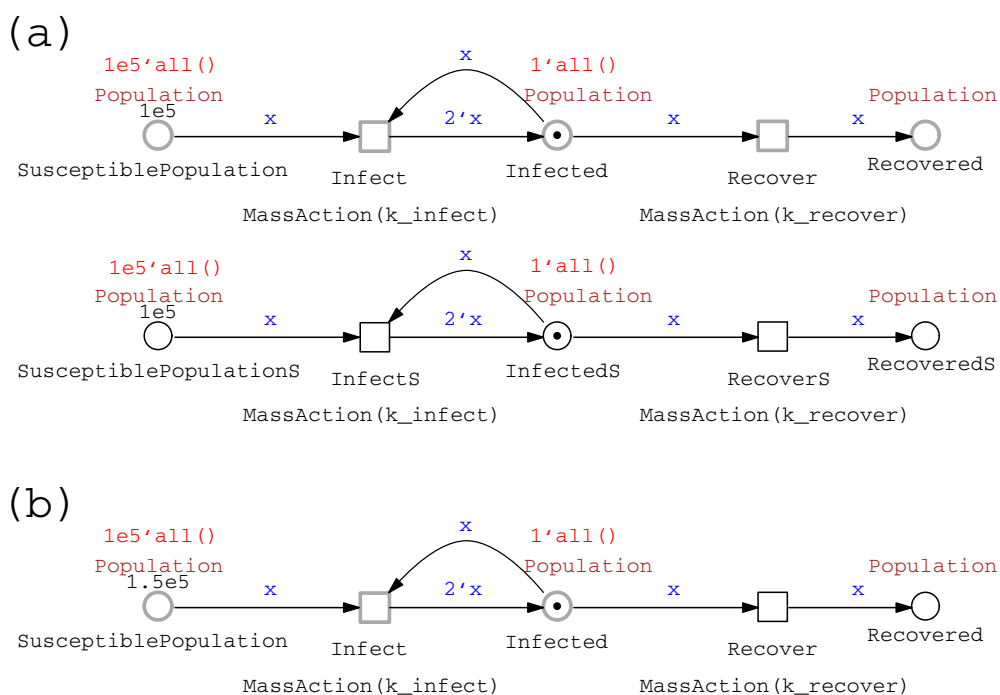


Figure 5.4: To compare the performance of hybrid simulation algorithms, the model contains two disjoint SIR subnets in the form of \mathcal{CPN}^c and \mathcal{SPN}^c . Initial values of the kinetic constants: $k_{\text{infect}} = 5.0e - 5$; $k_{\text{recover}} = 1.0e - 1$

Table 5.1: The performance of hybrid simulation algorithms supported by Spike.

Model \ Algorithm	static	staticAcc	HRSSA	HRSSAacc
(a)	30.1	13.1	21.8	1.6
(b)	18.0	5.7	3.0	3.1

Time [s]

Runtime of the four hybrid algorithms carried out on a PC with 2.6 GHz Core i7-6700HQ processor and 32GB memory. The simulations performed on models presented in Figure 5.4. Results are given in seconds [s].

5.2 Simulation of Adaptive Models

The simulation of adaptive models is an important part for all adaptive systems. In Spike, a discrete-time adaptive modelling system (stepwise simulation) scans the state of the model, which provides feedback. Based on the feedback the model internal parameters are automatically adjusted by means of predefined conditions. The model adjustment is controlled by a feedback loop which provides the generic mechanism for self-adaptation. In Spike the control feedback loop (see Figure 5.5) is based on the model presented in [BSG+09]. It comprises four implicit components/activities:

- simulation - collects data from the model executed and its current state,
- analyse - analyses the data to infer trends and identify symptoms,
- decide - decides how to act on the model executed based on analyses which are defined by conditions,
- alter model - alters model parameters based on decisions.

All the activities can be defined in the configuration of a stepwise simulation with the help of an *SPC* script.

A stepwise simulation advances in the given time range. Based on the current state of the model and the simulation, it enables a model to be adapted after each simulation step. As presented in [KCR+09, Kan12], such an approach can be used to dynamical change the simulation algorithm what allows performing hybrid simulation. Based on the analysis of the system state, transitions can be clustered and assigned to different simulation algorithms. Currently, during the runtime of a stepwise simulation, the dynamic change of a simulation algorithm (by changing its type or configuration) is not supported by Spike. This feature is considered in future work.

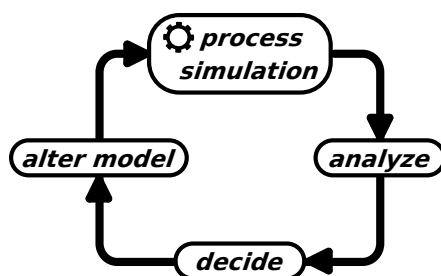


Figure 5.5: Control feedback loop implicitly embedded in Spike’s stepwise simulation.

Example 5.4. The infection rate changes dynamically during a pandemic for various reasons. Depending on the pandemic situation, one of the reasons could be rules that restrict or relax social distancing. After applying the rules, the infection rate does not change immediately, instead it changes over a range of time. During this period, the

infection rate decreases or increases, depending on whether the restriction or relaxation rules are applied. The change of the infection rate depends on the current state of the system and requires appropriate adjustment of the model parameters during the simulation.

The following configuration of the simulation experiment is defined over the model in Figure 2.3 and comprises the following main steps:

- (a) Specification of the source of the model; lines 10 – 12.
- (b) Reconfiguration of the model by setting the new value of the kinetic constant k_{infect_a} and new initial state of the susceptible population $SusceptiblePopulation_A$; lines 18 – 27.
- (c) Configuration of the simulation by setting up its type and solver; lines 42 – 51.
- (d) Configuration of the simulation time; line 53.
- (e) Configuration of the stepwise simulation; lines 61 – 141. This step allows to define the control feedback loop through the *SPC* script used for self-adaptation of the model. In the lines 67 – 68 two auxiliary variables k_{infect_lo} and k_{infect_hi} are defined which store the values of the kinetic parameters of the restriction and relaxation rules. The lines 72 – 78 define the time period $iWin$ in which the restriction and relaxation rules will be smoothly applied; lines 126 – 135. It means that the infection kinetic parameter will progressively reach the value defined by k_{infect_lo} in the case of restriction; lines 96 – 104; and the value defined by k_{infect_hi} in the case of relaxation; lines 110 – 117.
- (f) Configuration of how to store the simulation results. It allows specifying which the simulation traces are to be recorded. The name of the resulting file depends on the variables defined in the configuration, what allows to generate a unique name; lines 144 – 166.

The control feedback loop is realized within the *do* object which is evaluated after each step of a simulation. After each simulation step the system state is accessible by using the following predefined objects and variables:

- *simulation* - this object allows to read the time and the step of the simulation, see line 89;
- *place* - this object allows read/write access to the model places, see in the line 69. The place *Infected_A* (*place.Infected_A*) is used in the boolean expression of the conditional block that checks if the number of infected species is greater than 40% of the susceptible population;

5. USE CASES

- *constant* - this object allows read/write access to the model constants, see in the line 102. The constant *k_infect_a* (*constant.k_infect_a*) is used in the expression that calculates a new value of the variable *dWinStepSize*.

The combination of conditional blocks and expressions allows analysing the system state (boolean expression of conditional blocks). Depending on the analyses, decisions can be made (conditional blocks), which actions to perform, e.g. alter the model, set a new variable value, etc.

The simulation results are presented in Figure 5.6. It can be clearly seen if the size of the infected population *Infected_A* is greater than 40% of the susceptible population *SusceptiblePopulation_A* then the restriction rules are applied (a). In this case, the infection kinetic parameter *k_infect_a* is progressively decreasing (b). Similarly in the case of applying relaxation rules. If the size of the infected population is less than 20% of the susceptible population, then the relaxation rules are applied and the infection kinetic parameter progressively increases.

5.2 Simulation of Adaptive Models

```
1 /**
2  * Example configuration of a stepwise simulation
3  */
4
5 // - line comment
6 /*
7  - block comment
8 */
9 // Import - exactly one model
10 import: {
11     from: "./model/SIR-SPN.and1";
12 }
13
14
15 configuration: {
16     model: {
17         constants: {
18             all: {
19                 k_infect_a: 5.0e-5;
20             }
21         }
22     }
23     places: {
24         SusceptiblePopulation_A: 20000;
25     }
26 }
27
28 simulation:
29 {
30     /*
31     * This is example variable that is added
32     * to the log
33     */
34     varExample: model.places.SusceptiblePopulation_A;
35     // Name of a simulation
36     name: "SIR";
37     /*
38     * Set up a simulation
39     */
40     type:stochastic: {
41         solver:
42         direct: {
43             threads: 1;
44             runs: 3;
45             //seed: 2413805201;
46         }
47         single: true; // Single
48         //avg: false; // Default set to true
49     }
50
51     interval: 0:200:100;
52
53 }
```

5. USE CASES

```
54  /*
55  * Stepwise simulation
56  * Description: Depending on the current number of
57  * infected specimens set restriction or relaxation
58  * rules by applying change of infection kinetic rates
59  * in a given time frame.
60  */
61  onStep: enabled: {
62    /*
63     * Kinetic parameters of
64     * the restriction and relaxation
65     * rules
66     */
67    k_infect_lo: 1.0e-9;
68    k_infect_hi: 5.0e-5;
69
70    k_infect:observe: constant.k_infect_a;
71    iInitailSusceptiblePopulation_A: place.SusceptiblePopulation_A;
72    iTimeFrame: 2;// Stretch factor
73    /*
74     * Calculate the window size that stretch over
75     * the full time frames defined by iTimeFrame
76     */
77    iWin: (interval.splitting / (interval.end - interval.start)) *
78          iTimeFrame;
79    iWinStep: 0;
80    bFirst: false;
81    bRelax: false;
82    dWinStepSize: 0;
83
84    LOG = "END_INIT";
85    /*
86     * Smoothed stepwise lockdown and relaxation
87     */
88    do: {
89      LOG = "step:" << simulation.step;
90      LOG = "time:" << simulation.time;
91      /*
92       * Change infection rate if the number of
93       * infected specimens is > than 40% of susceptible
94       * population
95       */
96      if(place.Infected_A > iInitailSusceptiblePopulation_A * 0.4
97         && !bFirst) {
98
99         bFirst = true;
100        iWinStep = 0;
101        // Distance & step size
102        dWinStepSize = (constant.k_infect_a - k_infect_lo) / iWin;
103        bRelax = false;
104      }
105      /*
106       * Change infection rate if the number of
107       * infected specimens is < than 20% of susceptible
108       * population
109       */
110      else if(place.Infected_A < iInitailSusceptiblePopulation_A *
111              0.2 && !bRelax && bFirst) {
112
113        iWinStep = 0;
114        // Distance & step size
115        dWinStepSize = (constant.k_infect_a - k_infect_hi) / iWin;
116        bRelax = true;
117      }

```

5.2 Simulation of Adaptive Models

```
118 // ABS - absolut value
119 if(dWinStepSize < 0) {
120     dWinStepSize = -dWinStepSize;
121 }
122 /*
123  * Adjust the kinetic parameter according
124  * to the position in the time frame
125  */
126 if(iWinStep < iWin) {
127     if(!bRelax) {
128         constant.k_infect_a = constant.k_infect_a -
129             dWinStepSize;
130     } else if(bRelax) {
131         constant.k_infect_a = constant.k_infect_a +
132             dWinStepSize;
133     }
134     iWinStep = iWinStep + 1;
135 }
136 // Set the value of the observed variable
137 k_infect = constant.k_infect_a;
138 // Logging extra information
139 LOG = "bRelax: " << bRelax;
140 }
141 }
142
143
144 export: {
145     // Array of places to save (if empty export all)
146     places: []; // all places
147     //places:c: []; // all coloured places
148     //places:u: []; // uncoloured places
149     transitions: []; // all transitions
150     //transitions:c: []; // all coloured transitions
151     //transitions:u: []; // all uncoloured transitions
152     observers: [];
153     csv: {
154         sep: ";"; // Separator
155
156         file: "./data/"
157             << import.name << "_"
158             << configuration.simulation.type << "_"
159             << configuration.simulation.type.solver
160             << "_" << configuration.simulation.type.solver.runs
161             << "_" << configuration.model.constants.all.k_infect_a
162             << "_"
163             << configuration.model.places.SusceptiblePopulation_A
164             << "_FIELSE-step.csv"; // File name
165     }
166 }
167 }
168 }
169
170 log: {
171     sim.varExa: configuration.simulation.varExample;
172 }
```

5. USE CASES

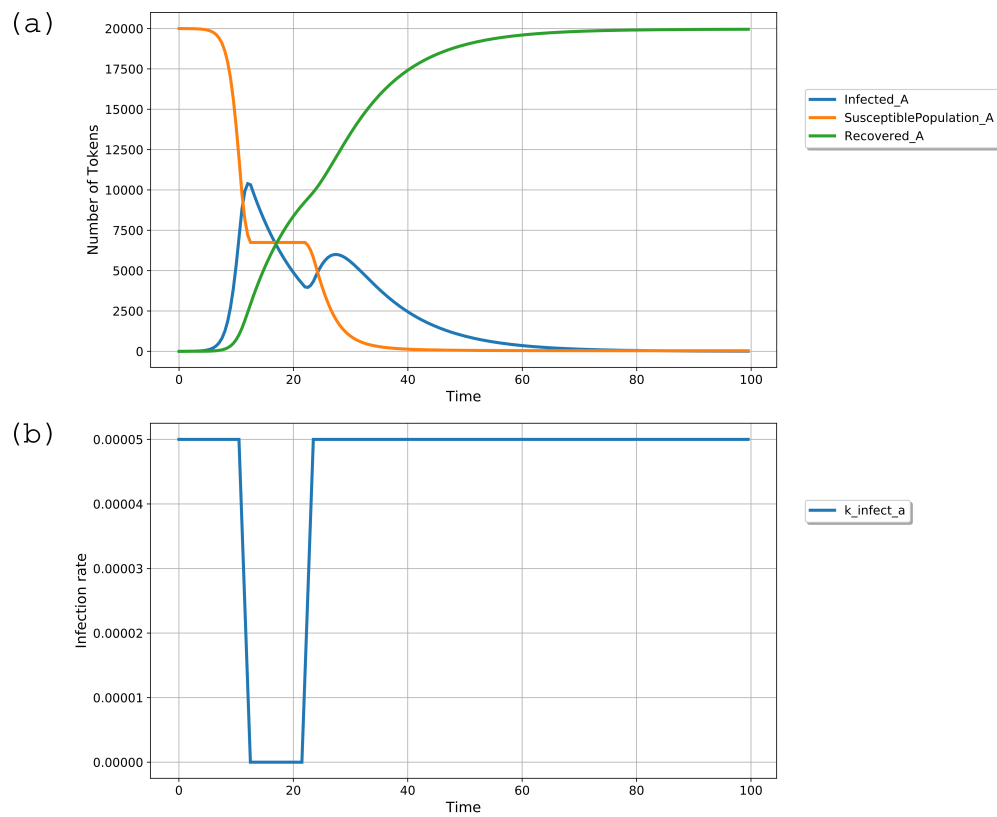


Figure 5.6: Simulation traces of stepwise simulation. It can be clearly seen if the size of the infected population *Infected_A* is greater than 40% of the susceptible population *SusceptiblePopulation_A* then the restriction rules are applied (a). In this case the infection kinetic parameter k_{infect_a} is progressively decreasing (b). Similarly in the case of applying relaxation rules. If the size of the infected population is less than 20% of the susceptible population, then the relaxation rules are applied and the infection kinetic parameter progressively increases.

Example 5.5. This follow-up example represents an experiment that aims to compare the effects of applying two different relaxation time frames:

1. *bigbang* - after applying the relaxation rules, the infection rate changes immediately (in one simulation step),
2. *smooth* - after applying the relaxation rules, the infection rate changes smoothly over a time range (multiple simulation steps).

To achieve this, the configuration of the experiment utilizes the feature of scanning variable values.

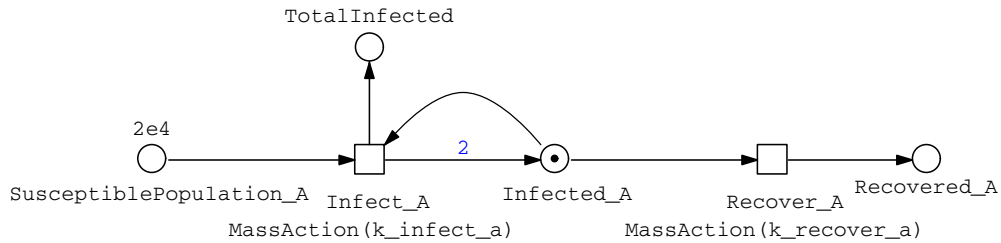


Figure 5.7: The SIR model used in the example is a variation of the SIR models in Figure 2.3. It contains one population *A* and the additional place *TotalInfected*, which holds the number of total infections. Initial values of the kinetic constants: $k_{infect_a} = 5.0e - 5$; $k_{recover_a} = 1.0e - 1$

The following configuration of the simulation experiment is defined over the model in Figure 5.7 and comprises the following main steps:

- (a) Specification of the source of the model; lines 11 – 13.
- (b) Reconfiguration of the model by setting the new value of the kinetic constant k_{infect_a} and new initial state of the susceptible population *SusceptiblePopulation_A*; lines 19 – 28.
- (c) Configuration of the simulation by setting up its type and solver; lines 38 – 51.
- (d) Configuration of the simulation time; line 53.
- (e) Configuration of the stepwise simulation; lines 61 – 168. This step allows to define the control feedback loop through *SPC* script used for self-adaptation of the model. In the lines 67 – 68 two auxiliary variables k_{infect_lo} and k_{infect_hi} are defined which store the values of kinetic parameters of the restriction and relaxation rules. The lines 76 – 85 define the stretch factors for calculating time periods of the restriction and relaxation rules. The definition involves parameter scanning which allows defining values for the two cases *bigbang* and *smooth*, respectively. The parameter scanning will trigger branching, which leads to two separate simulation branches.

5. USE CASES

The lines 90 – 95 define two time periods *win.iRest* and *win.iRelax* in which the restriction and relaxation rules will be applied; lines 145 – 157. They stretch over the full time frames defined by *timeFrame.iRestriction* and *timeFrame.iRelaxation*, respectively. It means that the infection kinetic parameter will progressively reach the value defined by *k_infect_lo* in the case of restriction; lines 113 – 121; and the value defined by *k_infect_hi* in the case of relaxation; lines 127 – 136.

- (f) Configuration of how to store the simulation results. It allows specifying which the simulation traces are to be recorded. The name of the resulting file depends on the variables defined in the configuration, what allows to generate a unique name; lines 170 – 186.

The simulation results are presented in Figure 5.8. It can be clearly seen that the simulation results depend on the size of the relaxation window. The number of infected species *Infected_A* rise to the:

- higher value in the case of *bigbang* - immediate introduction of relaxation, where the size of the relaxation window is set to zero (*win.iRelax* = 0),
- lower value in the case of *smooth* - step by step introduction of relaxation, where the size of the relaxation window equals twelve (*win.iRelax* = 12).

It is worth noting that the total number of infections is the same in both cases. The size of the restriction windows is the same in both cases, and it equals four (*win.iRest* = 4).

5.2 Simulation of Adaptive Models

```
1 /**
2  * Example configuration of a stepwise simulation:
3  * BIGBANG vs SMOOTH relaxation
4  */
5
6 // - line comment
7 /*
8  - block comment
9 */
10 // Import - exactly one model
11 import: {
12   from: "./model/SIR-SPN-BIGBANG.and1";
13 }
14
15
16
17 configuration: {
18
19   model: {
20     constants: {
21       all: {
22         k_infect_a: 5.0e-5;
23       }
24     }
25     places: {
26       SusceptiblePopulation_A: 20000;
27     }
28   }
29
30   simulation:
31   {
32
33     // Name of a simulation
34     name: "SIR";
35     /*
36     * Set up a simulation
37     */
38     type: continuous: {
39       solver:
40       BDF: {
41         semantic: "adapt";
42         iniStep: 0.1;
43         linSolver: "CVDense";
44         relTol: 1e-5;
45         absTol: 1.0e-10;
46         autoStepSize: false;
47         reductResultingODE: true;
48         checkNegativeVal: false;
49         outputNoiseVal: false;
50       }
51     }
52
53     interval: 0:200:100;
```

5. USE CASES

```
54  /*
55  * Stepwise simulation
56  * Description: Depending on the current number of
57  * infected specimens set restriction or relaxation
58  * rules by applying change of infection kinetic rates
59  * in a given time frame.
60  */
61  onStep: enabled: {
62    /*
63     * Kinetic parameters of
64     * the restriction and relaxation
65     * rules
66     */
67    k_infect_lo: 1.0e-9;
68    k_infect_hi: 5.0e-5;
69
70    k_infect:observe: constant.k_infect_a;
71    iInitailSusceptiblePopulation_A: place.SusceptiblePopulation_A;
72    /*
73     * Scan over stepwise simulation variables values
74     */
75    // Stretch factors
76    timeFrame: [[
77      bigbang: {
78        iRestriction: 2;
79        iRelaxation: 0;
80      },
81      smooth: {
82        iRestriction: 2;
83        iRelaxation: 6;
84      }
85    ]];
86    /*
87     * Calculate two windows size that stretch over the full time frames
88     * defined by timeFrame.iRestriction and timeFrame.iRelaxation
89     */
90    win: {
91      iRest: (interval.splitting / (interval.end - interval.start)) *
92            timeFrame.iRestriction;
93      iRelax: (interval.splitting / (interval.end - interval.start)) *
94            timeFrame.iRelaxation;
95    }
96    iWinStep: 0;
97    bFirst: false;
98    bRelax: false;
99    dWinStepSize: 0;
100
101    LOG = "END_INIT";
102    /*
103     * Smoothed stepwise lockdown and relaxation
104     */
105    do: {
106      LOG = "step:" << simulation.step;
107      LOG = "time:" << simulation.time;
108      /*
109       * Change infection rate if the number of
110       * infected specimens is > then 40% of susceptible
111       * population
112       */
113      if(place.Infected_A > iInitailSusceptiblePopulation_A * 0.4 && !bFirst) {
114        bFirst = true;
115        iWinStep = 0;
116        // Distance & step size
117        if(win.iRest != 0) {
118          dWinStepSize = (constant.k_infect_a - k_infect_lo) / win.iRest;
119        }
120        bRelax = false;
121      }
122    }
```


5.2 Simulation of Adaptive Models

```
122     /*
123     * Change infection rate if the number of
124     * infected specimens is < then 20% of susceptible
125     * population
126     */
127     else if(place.Infected_A < iInitailSusceptiblePopulation_A * 0.2 && !bRelax &&
128           bFirst) {
129
130         iWinStep = 0;
131         // Distance & step size
132         if(win.iRelax != 0) {
133             dWinStepSize = (constant.k_infect_a - k_infect_hi) / win.iRelax;
134         }
135         bRelax = true;
136     }
137     // ABS - absolut value
138     if(dWinStepSize < 0) {
139         dWinStepSize = -dWinStepSize;
140     }
141     /*
142     * Adjust the kinetic parameter according
143     * to the position in the time frame
144     */
145     if(!bRelax) {
146         if(iWinStep < win.iRest) {
147             constant.k_infect_a = constant.k_infect_a - dWinStepSize;
148         } else if(win.iRest == 0) {
149             constant.k_infect_a = k_infect_lo;
150         }
151     } else if(bRelax) {
152         if(iWinStep < win.iRelax) {
153             constant.k_infect_a = constant.k_infect_a + dWinStepSize;
154         } else if(win.iRelax == 0) {
155             constant.k_infect_a = k_infect_hi;
156         }
157     }
158     iWinStep = iWinStep + 1;
159
160     // Set the value of the observed variable
161     k_infect = constant.k_infect_a;
162     // Logging extra informations
163     LOG = "place.Infected_A: " << place.Infected_A;
164     LOG = "k_infect: " << k_infect;
165     LOG = "dWinStepSize: " << dWinStepSize;
166     LOG = "END_DO";
167 }
168 }
169
170 export: {
171     // Array of places to save (if empty export all)
172     places: [];// all places
173     //transitions: [];// all transitions
174     observers: [];
175     csv: {
176         sep: ";";// Separator
177         file: "./data/"
178             << import.name << "_"
179             << configuration.simulation.type << "_"
180             << configuration.simulation.type.solver
181             << "_" << configuration.model.constants.all.k_infect_a
182             << "_" << configuration.model.places.SusceptiblePopulation_A
183             << "_" << configuration.simulation.onStep.timeFrame
184             << "-step.csv";// File name
185     }
186 }
187 }
188 }
```

5. USE CASES

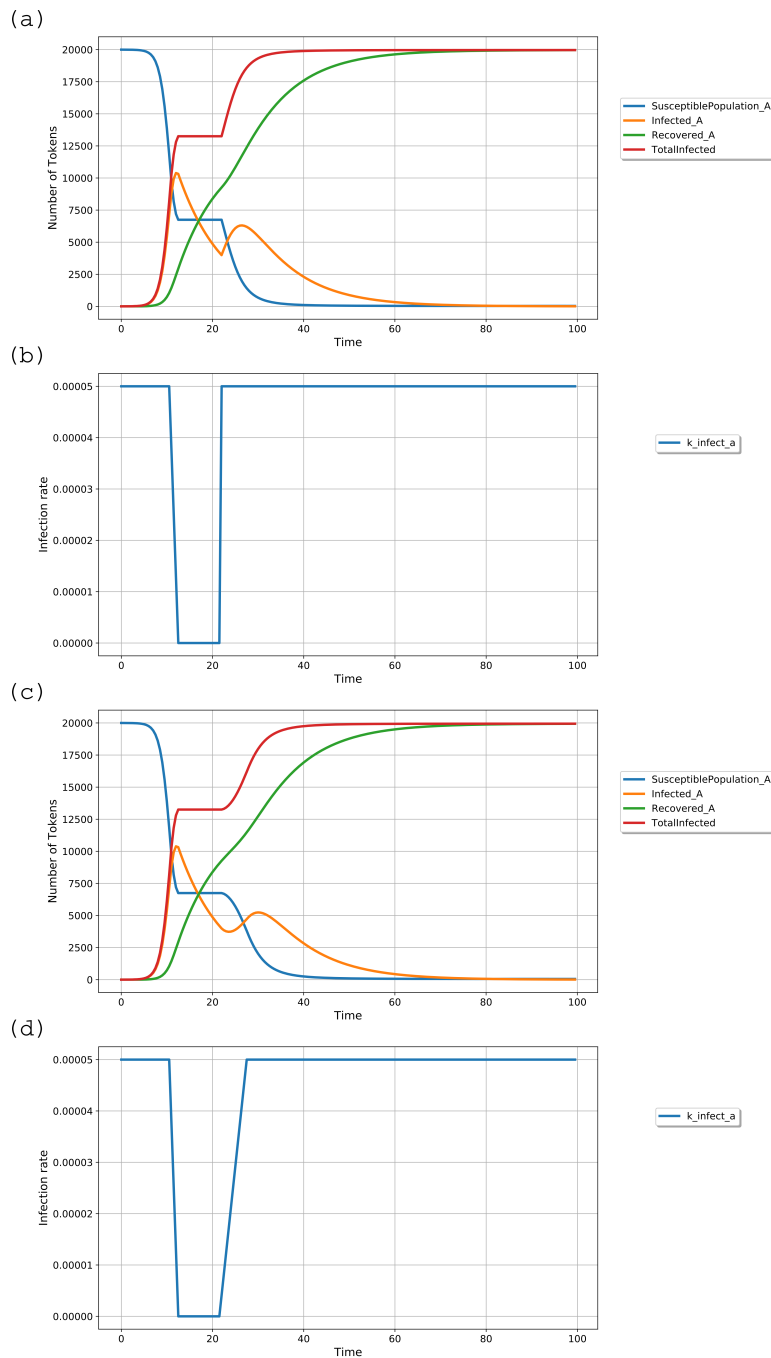


Figure 5.8: The simulation results depend on the size of relaxation window. The number of infected species *Infected.A* rises rapidly and to a higher value in the case of *bigbang* - immediate introduction of relaxation, where the infection kinetic parameter k_{infect_a} is immediately decreased (a), and the size of the relaxation window is set to zero ($win.iRelax = 0$) (b) or rise slowly and to a lower value in the case of *smooth* - step by step introduction of relaxation, where the infection kinetic parameter k_{infect_a} is progressively decreasing (c), and the size of the relaxation window equals twelve ($win.iRelax = 12$) (d). It is worth noting that the total number of infections (*TotalInfected*) is the same in both cases. The size of restriction windows is the same in both cases, and it equals four ($win.iRest = 4$).

5.3 Spike as a Backend Simulator for Parameter Optimization

Parameter optimization is required when during developing a model some parameters are not known yet or uncertain. A simulative approach can be applied in order to optimize / estimate unknown values of model parameters. In this case, it is necessary to carry out a series of simulation experiments and compare the obtained results with the help of a fitness function (also known as the evaluation function), which evaluates how close a given solution is to the optimum solution of the given problem. The fitness function can have various forms and can be related not only to time series but also to expectations (e.g. steady state), temporal logic, etc. Depending on the number of uncertain parameters and the number of parameter values, the size of the simulation set can be large and is equal to the product expressed by equation (5.1)

$$R \prod_{i=1}^m n(i), \quad (5.1)$$

where R is the number of repetitive simulation runs, m is the number of uncertain parameters and $n(i)$ is the number of values of the i th parameter.

Optimization through a simulation can be used as a search method [CM97] for the best candidates of input variables among all valid alternatives at any system state. By adopting heuristic evaluation it is possible to reduce a search space without explicitly evaluating each possibility. Spike does not directly support a heuristic parameter optimization. An optimization strategy must be implemented separately, while Spike performs the simulation task. Nevertheless, Spike features such as parameters scanning and parallel execution of configuration branches, makes it suitable for the task of brute force optimization. In the following two scenarios of parameter optimization, it is shown how Spike can be used to perform the simulation task.

Brute force. This is a straightforward approach where a new simulation is performed for each combination of parameters. After performing all possible simulations, the best matching results should be selected using a fitness function. This approach is presented by Algorithm 17.

Example 5.6. The brute force approach can be useful as a quick method if the size of the search space for each variable values is relatively small. The simulation tasks can be easily performed by Spike, exploiting the branching of simulation configurations (parameter scanning).

To find the best fitted result, the evaluation function represented by Algorithm 18 can be used. The given algorithm calculates the average percentage difference between two data series.

5. USE CASES

Algorithm 17: Use case: Brute force multiple parameter optimization.

```
1 Load model;
2 Determine simulator configuration;
3 for each unique combination of parameter values do
4     Determine model configuration;
5     Create new configuration branch;
6     Run simulation;
7     Save results of the simulation;
8 end
9 for each stored results do
10    if results not fitted then
11        Remove results;
12    end
13 end
```

Algorithm 18: Fitness function: comparison of two data traces.

```
Data: time series  $TS$ ;  
        simulation data trace  $DT$ ;  
Result: fitness value  $dFit$ ;  
1 double  $dFit$ ;  
2 for each integer index  $iIdx$  in range of  $|TS|$  do  
3     if  $TS[iIdx] + DT[iIdx] \neq 0$  then  
4          $dFit = dFit + |(TS[iIdx] - DT[iIdx]) / (TS[iIdx] + DT[iIdx])| / 2 * 100$ ;  
5     end  
6 end
```

5.3 Spike as a Backend Simulator for Parameter Optimization

For the purpose of this example, the reference data (see Figure 5.10 on the page 135) has been generated by the simulation of the model in Figure 5.9 with the following set of kinetic constants: $k_{infect_b} = 5.0e - 6$; $k_{recover_b} = 1.0e - 2$. In this example, the trace *Recover_B* can be used as the reference time series in the fitness function.

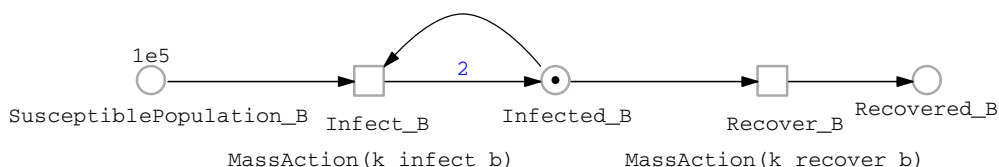


Figure 5.9: SIR model used in the example, with the following set of kinetic constants: $k_{infect_b} = 5.0e - 6$; $k_{recover_b} = 1.0e - 2$. The simulation results are presented in Figure 5.10.

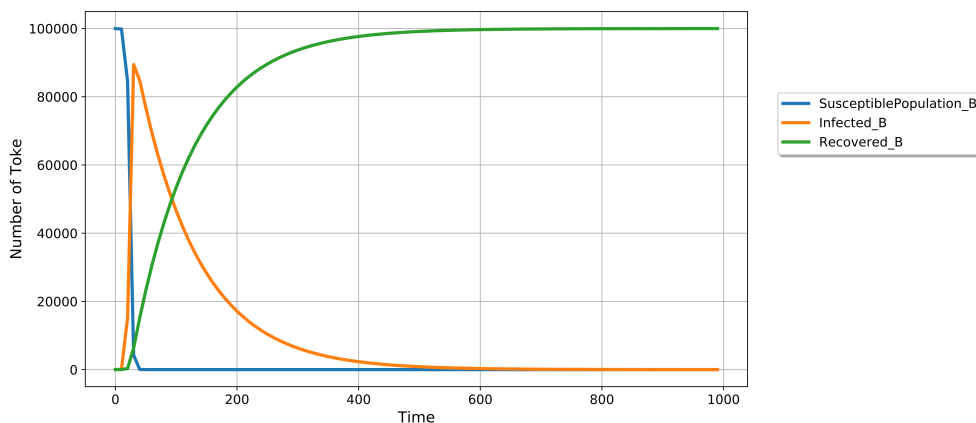


Figure 5.10: Simulation traces of the deterministic simulation of the SIR model in Figure 5.9, which are used as the reference data trace in the examples of parameter optimization.

By combining Algorithm 17 and 18, it is easy to select a most fitted simulation trace from the set of simulation traces. The set is generated by the following configuration of the simulation experiment which is defined over the model in Figure 5.9 (page 135) and comprises the following main steps:

- (a) Specification of the source of the model; lines 1 – 3.
- (b) Reconfiguration of the model by setting the range of parameters for scanning the kinetic constant k_{infect_b} and $k_{recover_b}$; line 8.
- (c) Configuration of the simulation by setting up its type and solver; lines 14 – 26.
- (d) Configuration of the simulation time; line 28;
- (e) Configuration of how to store the simulation results. It allows specifying which the simulation traces are to be recorded. The name of the resulting file depends

5. USE CASES

on the variables defined in the configuration, what allows to generate a unique name; lines 30 – 43.

The beginning of the search space of the kinetic constant k_{infect_b} is set to $1.0e - 6$ and will be scanned with a step of $1.0e - 6$ to the end of the search space $1.0e - 5$. Similarly, for the kinetic constant $k_{recover_b}$ the search space is limited by two real values $1.0e - 3$ and $1.0e - 1$ and will be scanned with a step of $1.0e - 3$. In this case, the configuration of parameter scanning will result in 990 simulation branches. For this example, the proposed range of values to be scanned is artificially adjusted to ensure that one of the simulation branches contains the desired configuration values for the kinetic parameters.

```
1 import: {
2   from: "./model/SIR-SPN.and1";
3 }
4
5 configuration: {
6   model: {
7     constants: { all: {
8       k_infect_b: [[1.0e-6:1.0e-6:1.0e-5]]; k_recover_b: [[1.0e-3:1.0e-3:1.0e-1]];
9     }}
10  }
11
12 simulation: {
13   name: "SIR-BRUTEFORCE-OPTIMIZATION";
14   type: continuous: {
15     solver: BDF: {
16       semantic: "adapt";
17       iniStep: 0.1;
18       linSolver: "CVDense";
19       relTol: 1e-5;
20       absTol: 1.0e-10;
21       autoStepSize: false;
22       reductResultingODE: true;
23       checkNegativeVal: false;
24       outputNoiseVal: false;
25     }
26   }
27
28   interval: 0:100:1000;
29
30   export: {
31     places: [];
32     csv: {
33       sep: ",";
34       file: "./simresults/"
35         << name << "_"
36         << configuration.simulation.type << "_"
37         << configuration.simulation.type.solver << "_"
38         << "TST"
39         << "_" << configuration.model.constants.all.k_infect_b
40         << "_" << configuration.model.constants.all.k_recover_b
41         << ".csv";
42     }
43   }
44 }
45 }
```

5.3 Spike as a Backend Simulator for Parameter Optimization

Heuristic. The brute force approach does not reduce the search space of parameters values. For each combination of parameters, a new simulation is performed, which is both time and resources consuming. Frequently, the size of the value search space is too large and prevents the optimization over a finite period of time. In such case, a heuristic method is better suitable. In [DG08], a genetic algorithm is used to drive the optimization strategy of model parameters. The DIRECT method and its derivatives [JPS93], as presented in [GK10], also suite well as the optimization strategy in the use case presented by Algorithm 19. As shown in Fig. 5.11 (page 137), simulation results are provided for an optimization strategy which reduces the search space of parameter values by checking the fitness of a set of parameters and provides a set of best fitted parameter values as a feedback to the model.

Algorithm 19: Use case: Heuristic multiple parameter optimization.

```
1 Load model;  
2 Determine simulator configuration;  
3 repeat  
4   for each unique combination of parameter values do  
5     Determine model configuration;  
6     Create new configuration branch;  
7     Run simulation;  
8   end  
9   Run optimization strategy;  
10 until space reduced;
```

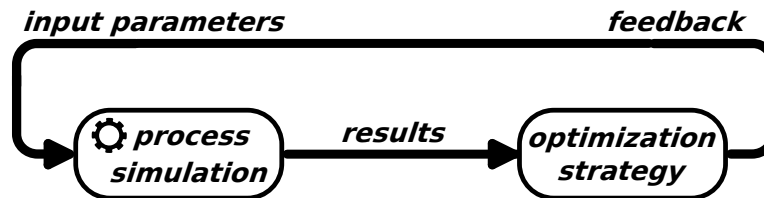


Figure 5.11: Optimization through simulation - simulation results are provided to an optimization strategy, which reduces the space of parameter values by checking the fitness of set of parameters and provides a set of best fitted parameter values for the model as feedback.

Example 5.7. In this example Spike is used as backend simulator, which performs the simulation task. The control loop and the optimization strategy is implemented by the use of the Python programming language [VD09], as Spike does not currently support heuristic parameter optimization directly.

The optimization strategy is implemented as a genetic algorithm [BSH17] with the help of the Python library *geneticalgorithm* [Sol13], which was slightly modified for the purposes of the example. All script source code can be found in Appendix B.

5. USE CASES

The genetic algorithm [Hol75, Jon75] is a black-box optimization technique which belongs to the class of random-based evolutionary algorithms. It is characterized by three main features:

- population - is a set of solutions from which new solutions are to be generated;
- fitness - is associated with each solution which evaluates how close a given solution is to the optimum solution of the given problem;
- variation - is a random process, which, based on a fitness value, performs random variations on individual solutions in order to generate a new population.

Base on the feedback, provided by each run of the optimization strategy, a new configuration of the model is determined. The newly determined model configuration is combined with the following configuration template.

```
1 import: {
2   from: "../model/SIR-SPN.and1";
3 }
4
5 configuration: {
6
7   model: [[%s]];
8
9   simulation: {
10    name: "SIR";
11    type: continuous: {
12     solver:
13     BDF: {
14      semantic: "adapt";
15      iniStep: 0.1;
16      linSolver: "CVDense";
17      relTol: 1e-5;
18      absTol: 1.0e-10;
19      autoStepSize: false;
20      reductResultingODE: true;
21      checkNegativeVal: false;
22      outputNoiseVal: false;
23    }
24  }
25
26   interval: 0:100:1000;
27
28   export: {
29     places: [];
30     csv: {
31      sep: ",";
32      file: "simresults/" << name << "_"
33          << configuration.model
34          << ".csv";
35    }
36  }
37 }
38 }
```


5.3 Spike as a Backend Simulator for Parameter Optimization

The combination takes place in line 7, where the placeholder string (*%s*) is replaced by the newly determined model configuration, e.g.:

```
1 inhab0: {  
2   constants: {all: { k_infect_b: 0.577220; k_recover_b: 1.454898; } }  
3 }
```

As in the previous example, the trace *Recover_B* from the generated data set (see Figure 5.10, page 140) has been used as the reference time series in the fitness function. The progress of the best performed experiment runs for two experiment set-ups is presents in Figure 5.12. The main difference between those two experiments is the size of the search space. In the first case (see Figure 5.12.(a), page 140), the search space is set to $k_{infect_b} \in \langle 1.0e - 6, 1.0e - 5 \rangle$ and $k_{recover_b} \in \langle 1.0e - 3, 1.0e \rangle$. In the second case (see Figure 5.12.(b), page 140), the size of the search space is much wider, and it is set to $k_{infect_b} \in \langle 1.0e - 7, 1.0 \rangle$ and $k_{recover_b} \in \langle 1.0e - 7, 1.0 \rangle$. In both cases the genetic algorithm performed 20 iterations for the population of size 50. The achieved accuracy for the case (a) is 0.0119032134008 and for (b) 3.104465461. The experiment results are presented in Figure 5.13(a) and (b), which can be compared with the reference data (c).

5. USE CASES

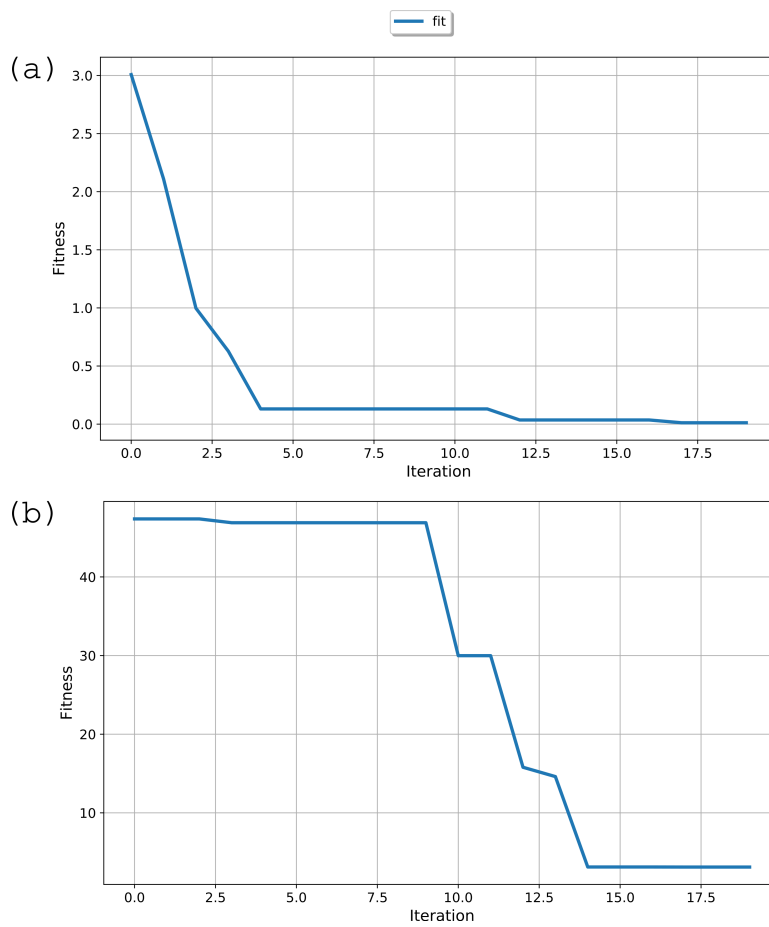


Figure 5.12: Progress of optimization of two experiment set-ups: (a) - *iteration* = 20, *population size* = 50, *mutation probability* = 0.4, search space: $k_{infect.b} \in \langle 1.0e - 6, 1.0e - 5 \rangle$, $k_{recover.b} \in \langle 1.0e - 3, 1.0e \rangle$, $FIT = 0.0119032134008$; (b) - *iteration* = 20, *population size* = 50, *mutation probability* = 0.6, search space: $k_{infect.b} \in \langle 1.0e - 7, 1.0 \rangle$, $k_{recover.b} \in \langle 1.0e - 7, 1.0 \rangle$, $FIT = 3.104465461$.

5.3 Spike as a Backend Simulator for Parameter Optimization

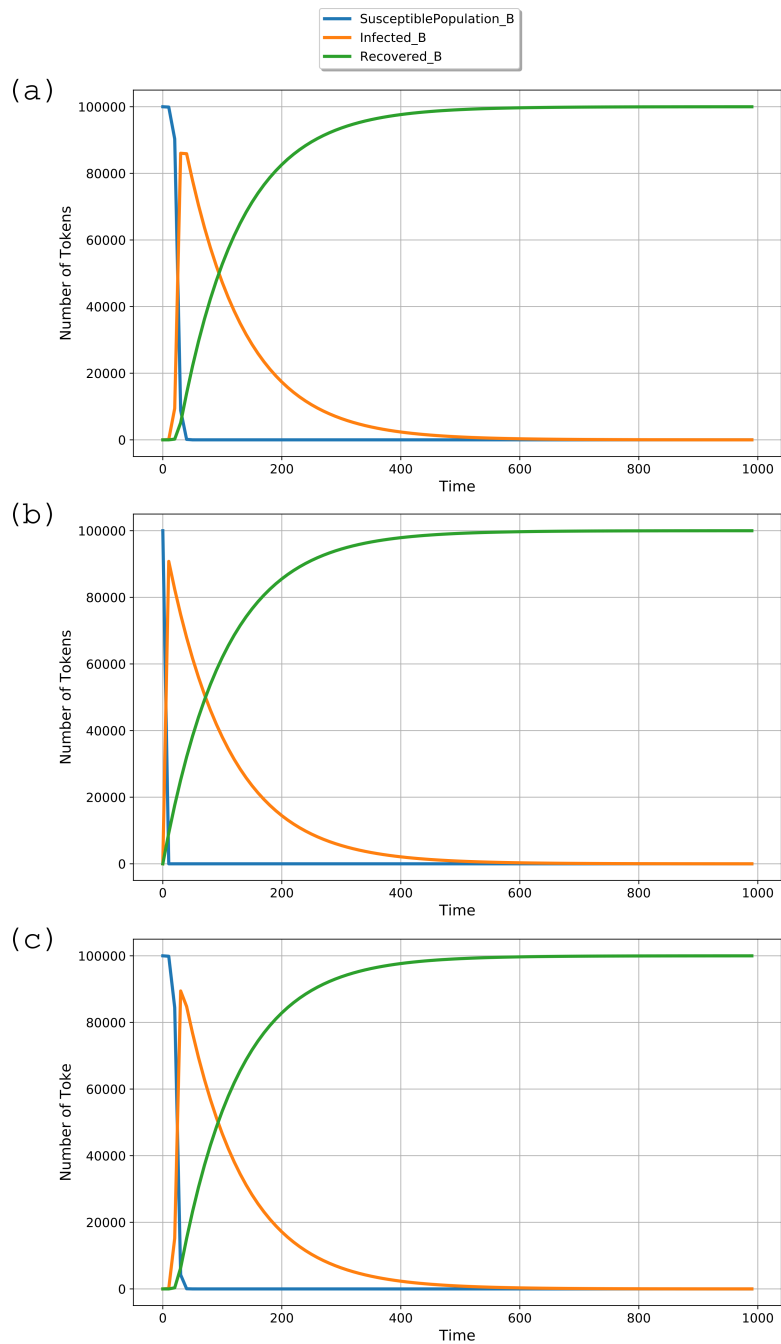


Figure 5.13: Simulation traces of the deterministic simulation of the SIR model in Figure 5.9; (a) - $FIT = 0.0119032134008$, $k_{infect_b} = 4.7277170068E-06$, $k_{recover_b} = 0.009999231$ and search space: $k_{infect_b} \in \langle 1.0e-6, 1.0e-5 \rangle$, $k_{recover_b} \in \langle 1.0e-3, 1.0e \rangle$; (b) - $FIT = 3.104465461$, $k_{infect_b} = 0.6702367369$, $k_{recover_b} = 0.0096678283$ and search space: $k_{infect_b} \in \langle 1.0e-7, 1.0 \rangle$, $k_{recover_b} \in \langle 1.0e-7, 1.0 \rangle$; (c) - the reference data trace, $k_{infect_b} = 5.0e-6$, $k_{recover_b} = 1.0e-2$.

5. USE CASES

5.4 Closing Remarks

The flexibility of *SPC* allows Spike to design and perform simulation experiments in very efficient ways. The first example shows the power of scanning of model parameters and simulation options. Through the branching of a configuration, Spike can perform a set of simulation experiments in parallel. The second and third examples focus on the stepwise simulation. Spike's stepwise simulation feature allows designing simulation experiments of adaptive models by adjusting dynamically the model parameters. The fourth example shows how to perform model parameter optimization by embedding Spike in a third party application. All of these examples illustrate main features of Spike, but certainly do not cover all use cases in which Spike can be used.

6

Conclusions and Outlook

6.1 Conclusions

Spike is an efficient tool for the reproducible execution of parallel simulation experiments of biochemical reaction networks. The modular structure of Spike and the mechanism of intermodule communication allows to easily extend Spike by new modules. The main functionalities of Spike allow to import and export \mathcal{PN} models in various formats. An imported coloured model can be unfolded using *IDD*-based unfolding, which is integrated in the internally developed `dssd_util` library used by Snoopy, Marcie and Spike. During the work on Spike, the *Boolean* colour set and the *elmeOf* operator were introduced into the *IDD*-based unfolding. The newly introduced colour set and operator increase the expressive power of the colour annotations by simplifying coloured expressions. To perform a simulation, Spike uses an internally developed simulation library; it is capable to run three basic types of simulations: stochastic, deterministic and hybrid, where each comes with several algorithms. Spike is supported by the scripting language (*SPC*), which allows for designing reproducible simulation experiments, that can be executed in parallel. Additionally, *SPC* allows the execution of a simulation in a stepwise manner.

The main goal of *SPC* is to efficiently support reproducible simulation experiments. *SPC* has a human-readable format and allows configuring a model, a simulation and observers. Additionally, it enables to define the export of simulation results. Through the branching of configuration it is possible to set up the scanning of model parameters and simulation options. The branches of a configuration are loosely coupled (they only have in common a high-level/parent configuration) and can be executed in parallel. *SPC* supports adaptive stepwise simulation, which allows for reconfiguring model parameters based on the current state of a model and a simulation. All of this allows Spike to efficiently perform reproducible experiments.

6. CONCLUSIONS AND OUTLOOK

6.2 Outlook

Spike and its configuration scripting language \mathcal{SPC} can be improved in many ways. Certain features are missing that should be addressed in future work.

- Full support of arrays - currently \mathcal{SPC} supports only the declaration of arrays as they are used only to set values of some configuration options. Accessing of array elements will allow to reduce the number of declared variables and to collect and organize data in many useful ways.
- Conditional loop blocks - condition loops allow certain parts of a program to be run multiple times while a condition remains true. Support of a conditional loop block in connection with arrays will be very handy. This will facilitate the processing of data during stepwise simulation.
- Temporal logic - the temporal logic is focused on formulas that use temporal operators to describe how static conditions change over the time. Support of the temporal logic syntax will allow to conveniently express how to alter a model after each simulation step, based on the current state of a model and a simulation.
- Model reduction - Spike allows for the basic reduction of a \mathcal{PN} model. It is able to structurally reduce a model by pruning clean siphons and constant places. However, this basic reduction methods are insufficient. The growing amount of experimental data and expressive power of the colour annotations leads to the development of complex models. A complex model represented by \mathcal{PN}^c needs to be unfolded before its simulation. After unfolding, the number of nodes can be much larger than in its coloured counterpart. Reduction of a model may yield a more optimized (in terms of size) model, provide insights into structural properties and reduces a simulation overhead. The main challenge of a reduction is to preserve the main three properties of a \mathcal{PN} model: liveness, reversibility and boundedness. The two simplest techniques that preserve the main three properties are pruning of clean siphons and constant places.
- Model decomposition - decomposition of \mathcal{PN} model into basic subnets. Decomposition can be done by network structure or through type, if the \mathcal{PN} is hybrid. The process of clustering should be aided through manual selection / specification of cluster set as well as through an automatic / algorithm approach. The model decomposition will allow for distributed simulation of the decomposed model. Such functionality should speed up the simulation of large models - more research needs to be done to get a clear answer.
- Distributed simulation - Spike is able to perform parallel executions of simulation experiments on single host. Future work should consider implementation of distributed simulation, which can speed-up the execution of an experiment in the following example cases:

- (a) - a simulation experiment contains a set of exhaustive simulations - in this case each simulation can be distributed over a network of computing peers, where each peer performs a single simulation.
 - (b) - a parallel simulation of a decomposed model - in a such case each component of the model is distributed over a network of computing peers, where each peer performs a single, parallel, synchronized simulation for the received model component.
- Parameter optimization - Optimization through a simulation can be used as a search method [CM97] for the best candidates of input variables among all valid alternatives at any system state. By adopting heuristic evaluation, it is possible to reduce a search space without explicitly evaluating each possibility. Spike features such as parameters scanning and parallel execution of configuration branches make Spike suitable for this task. However, all these features are not sufficient to perform parameter optimization. Future work should consider embedding the optimization strategy directly into Spike, which will be a complementary feature of parameter scanning. This will allow Spike to optimize a set of model parameters through an embedded optimization strategy.

6.3 Availability

Spike is developed in C++ and available for Linux, Mac/OSX and Windows. Binaries are statically linked and can be downloaded from Spike's website <https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Spike>, which provides also documentation, installation instructions and a set of examples. The source code of Spike is available in the GitHub repository: <https://github.com/PetriNuts/spike> under [GPLv3] licence.

6.4 Acknowledgement

The author would like to gratefully acknowledge George Assaf ¹, David Gilbert ² and his team, Monika Heiner ¹ and Robert Walton ³ for their active contributions in beta testing Spike.

¹Brandenburg University of Technology Cottbus-Senftenberg

²Brunel University London

³University of Oxford

References

- [ACT+05] ALFONSI, A.; CANCÈS, E.; TURINICI, G.; VENTURA, B.; HUISINGA, W.: Adaptive simulation of hybrid stochastic and deterministic models for biochemical systems. In: *ESAIM: Proc.* 14 (2005), pp. 1--13, cite: 29, 32
- [Bar18] BARBA, L. A.: *Terminologies for Reproducible Research*. 2018. -- Available at <https://arxiv.org/abs/1802.03311>, cite: 38
- [BBD+16] BREITWIESER, L.; BAUER, R.; MEGLIO, A. D.; JOHARD, L.; KAISER, M.; MANCA, M.; MAZZARA, M.; RADEMAKERS, F.; TALANOV, M.: The biodynamo project: Creating a platform for large-scale reproducible biological simulations. In: *arXiv preprint arXiv:1608.04967* (2016), cite: 41
- [BSH17] BOZORG-HADDAD, O.; SOLGI, M.; LOÁICIGA, H.: *Meta-heuristic and evolutionary algorithms for engineering optimization*. John Wiley & Sons, 2017, cite: 137
- [BSG+09] BRUN, Y.; SERUGENDO, G.; GACEK, C.; GIESE, H.; KIENLE, H.; LITOU, M.; MÜLLER, H.; PEZZÈ, M.; SHAW, M.: Engineering self-adaptive systems through feedback loops. In: *Software engineering for self-adaptive systems*. Springer, 2009, pp. 48--70, cite: 43, 44, 120
- [BW96] BOLLIG, B.; WEGENER, I.: Improving the variable ordering of OBDDs is NP-complete. In: *IEEE Transactions on computers* 45 (1996), Nr. 9, pp. 993--1002, cite: 90
- [Boe15] C. BOETTIGER: An introduction to Docker for reproducible research. In: *ACM SIGOPS Operating Systems Review* 49 (2015), Nr. 1, pp. 71--79, cite: 41
- [CH19] CHODAK, J.; HEINER, M.: Spike -- Reproducible Simulation Experiments with Configuration File Branching. In: BORTOLUSSI, Luca (Eds.); SANGUINETTI, Guido (Eds.): *Computational Methods in Systems Biology* Volume 11773. Cham: Springer, LNCS, 2019, pp. 315--321, cite: 2, 77, 79
- [CM97] CARSON, Y.; MARIA, A.: Simulation optimization: methods and applications. In: *Proceedings of the 29th conference on Winter simulation* IEEE Computer Society, 1997, pp. 118--126, cite: 106, 133, 145
- [CTT+16] CARDELLI, L.; TRIBASTONE, M.; TSCHAIKOWSKI, M.; VANDIN, A.: Symbolic computation of differential equivalences. In: *ACM SIGPLAN Notices* Volume 51 ACM, 2016, pp. 137--150, cite: 34
- [CTT+17] CARDELLI, L.; TRIBASTONE, M.; TSCHAIKOWSKI, M.; VANDIN, A.: ERODE: a tool for the evaluation and reduction of ordinary differential equations. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* Springer, 2017, pp. 310--328, cite: 34, 89
- [CW85] CARDELLI, L.; WEGNER, P.: On Understanding Types, Data Abstraction, and Polymorphism. In: *Computing Survey* 17 (1985), Nr. 4, pp. 471--522, cite: 14
- [CY19] CANON, R.S.; YOUNGE, A.: A case for portability and reproducibility of HPC containers. In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)* IEEE, 2019, pp. 49--54, cite: 42
- [DG08] DONALDSON, R.; GILBERT, D.: A Model Checking Approach to the Parameter Estimation of Biochemical Pathways. In: M. HEINER . A.M. UHRMACHER (Eds.): *Computational Methods in Systems Biology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 269--287, cite: 137
- [DHM+09] DIDIER, F.; HENZINGER, T. A.; MATEESCU, M.; WOLF, V.: Fast adaptive uniformization of the chemical master equation. In: *2009 International Workshop on High Performance Computational Systems Biology* IEEE, 2009, pp. 118--127, cite: 81
- [DTT+16] DRAWERT, B.; TROGDON, M.; TOOR, S.; PETZOLD, L.; HELLANDER, A.: Molns: A cloud platform for interactive, reproducible, and scalable spatial stochastic computational experiments in systems biology using pyurdm. In: *SIAM Journal on Scientific Computing* 38 (2016), Nr. 3, pp. C179--C202, cite: 42
- [ECMA-404] ECMA 2017 the JSON Data Interchange Syntax. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>, cite: 48
- [EU14] EWALD, R.; UHRMACHER, A.M.: SESSL: A Domain-Specific Language for Simulation Experiments. In: *ACM Trans. Model. Comput. Simul.* 24 (2014), Nr. 2. -- ISSN 1049--3301, cite: 40
- [GB00] GIBSON, M.; BRUCK, J.: Exact stochastic simulation of chemical systems with many species and many channels. In: *J. Phys. Chem.* 105 (2000), pp. 1876 -- 89, cite: 29
- [GCP+06] GRIFFITH, M.; COURTNEY, T.; PECCOUD, J.; SANDERS, W.: Dynamic partitioning for hybrid simulation of the bistable HIV-1 transactivation network. In: *Bioinformatics* 22 (2006), Nr. 22, pp. 2782--2789, cite: 32
- [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, cite: 1
- [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, cite: 25
- [Gil77] GILLESPIE, D.: Exact stochastic simulation of coupled chemical reactions. In: *J. Phys. Chem.* 81 (1977), Nr. 25, pp. 2340--2361, cite: 25, 81
- [Gil01] GILLESPIE, D.: Approximate accelerated stochastic simulation of chemically reacting system. In: *J. Chem. Phys.* 115 (2001), pp. 1716--1733, cite: 29, 81

REFERENCES

- [Gil07] GILLESPIE, D.: Stochastic simulation of chemical kinetics. In: *Annu Rev Phys Chem.* 58 (2007), Nr. 1, pp. 35--55, cite: 29, 32
- [GK10] GRIFFIN, J.D.; KOLDA, T.G.: Asynchronous parallel hybrid optimization combining DIRECT and GSS. In: *Optimization Methods and Software* 25 (2010), Nr. 5, pp. 797--817, cite: 137
- [GL79] GENRICH, H.J.; LAUTENBACH, K.: The analysis of distributed systems by means of predicate/transition-nets. In: *Semantics of Concurrent Computation*. Springer, 1979, pp. 123--146, cite: 14
- [GN00] GANSNER, E.; NORTH, S.: An open graph visualization system and its applications to software engineering. In: *Software: practice and experience* 30 (2000), Nr. 11, pp. 1203--1233, cite: 103
- [GPLv3] GNU General Public License. Version 3. Free Software Foundation. Available at <http://www.gnu.org/licenses/gpl.html>, cite: 145
- [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, cite: 81, 82
- [Her13] HERAJY, M.: *Computational Steering of Multi-Scale Biochemical Networks*, BTU Cottbus, Dep. of CS, PhD thesis, January 2013, cite: 12, 22, 25, 33, 34
- [HH12] HERAJY, M.; HEINER, M.: Hybrid Representation and Simulation of Stiff Biochemical Networks. In: *J. Nonlinear Analysis: Hybrid Systems* 6 (2012), November, Nr. 4, pp. 942--959, cite: 13, 82
- [HH16] HERAJY, M.; HEINER, M.: Accelerated Simulation of Hybrid Biological Models with Quasi-disjoint Deterministic and Stochastic Subnets. In: E CINQUEMANI, A. D. (Eds.): *Proc. 5th Int. Workshop on Hybrid Systems Biology (HSB 2016)* Volume 9957, Springer, LNBI, October 2016, pp. 20--38, cite: 82
- [HH17] 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, cite: 31, 34
- [HH18] HERAJY, M.; HEINER, M.: An Improved Simulation of Hybrid Biological Models with Many Stochastic Events and Quasi-Disjoint Subnets. In: M. RABE, A. JUAN, N. MUSTAFEE, A. SKOOGH, S. JAIN, B. JOHANSSON (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, cite: 13, 82, 116
- [HHL+12] HEINER, M.; HERAJY, M.; LIU, F.; ROHR, C.; SCHWARICK, M.: Snoopy -- A Unifying Petri Net Tool. In: *ATPN 2012*, Springer, LNCS 7347, 2012, pp. 398--407, cite: 2, 79, 82
- [Hil17] HILL, D.: Numerical Reproducibility of Parallel and Distributed Stochastic Simulation Using High-Performance Computing. In: *Computational Frameworks*. Elsevier, 2017, pp. 95--109, cite: 38
- [HLM14] HARKO, T.; LOBO, F. S. N.; MAK, M. K.: Exact analytical solutions of the Susceptible Infected Recovered (SIR) epidemic model and of the SIR model with equal death and birth rates. In: *arXiv e-prints* (2014), März, pp. arXiv:1403.2160, cite: 9
- [HLR+17] HERAJY, M.; LIU, F.; ROHR, C.; HEINER, M.: Snoopy's Hybrid Simulator: a Tool to Construct and Simulate Hybrid Biological Models. In: *BMC Systems Biology* (2017). -- published: July 28, 2017, cite: 81
- [HLR+18] HERAJY, M.; LIU, F.; ROHR, C.; HEINER, M.: Coloured Hybrid Petri Nets: an Adaptable Modelling Approach for Multi-scale Biological Networks. In: *Computational Biology and Chemistry* 76 (2018), pp. 87--100, cite: 17
- [HNW93] HAIRER, E.; NØRSETT, S.; WANNER, G.: *Springer Series in Comput. Mathematics*. Volume 8: *Solving ordinary differential equations I: nonstiff problems*. Springer-Verlag, 1993, cite: 31
- [Hol75] HOLLAND, J.H. *Adaptation in Natural and Artificial Systems*. 1975, cite: 138
- [HR02] HASELTINE, E.; RAWLINGS, J.: Approximate simulation of coupled fast and slow reactions for stochastic chemical kinetics. In: *J. Chem. Phys.* 117 (2002), Nr. 15, pp. 6959--6969, cite: 29, 32, 82
- [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, LNCS, June 2013, pp. 389--399, cite: 2, 35, 37, 79
- [HRS+10] HEINER, M.; ROHR, C.; SCHWARICK, M.; STREIF, S.: A Comparative Study of Stochastic Analysis Techniques. In: *Proc. 8th International Conference on Computational Methods in Systems Biology (CMSB 2010)*, ACM digital library, September 2010, pp. 96--106, cite: 81
- [HSG+06] HOOPS, S.; SAHLE, S.; GAUGES, R.; LEE, C.; PAHLE, J.; SIMUS, N.; SINGHAL, M.; XU, L.; MENDES, P.; KUMMER, U.: COPASI--a complex pathway simulator. In: *Bioinformatics* 22 (2006), Nr. 24, pp. 3067--3074, cite: 2
- [HSW15] HEINER, M.; SCHWARICK, M.; WEGENER, J.: Charlie -- An Extensible Petri Net Analysis Tool. In: DEVILLERS, Raymond (Eds.); VALMARI, Antti (Eds.): *Application and Theory of Petri Nets and Concurrency* Volume 9115. Cham: Springer, LNCS, 2015, pp. 200--211, cite: 2, 79
- [Huc15] HUCKA, M.: Systems biology markup language (SBML). In: *Encyclopedia of Computational Neuroscience* (2015), pp. 2943--2944, cite: 40, 88
- [HW96] HAIRER, E.; WANNER, G.: *Springer Series in Comput. Mathematics*. Volume 14: *Solving ordinary differential equations II: stiff and differential-algebraic problems*. Springer-Verlag, 1996, cite: 31
- [Jav92] JÁVOR, A.: Demon controlled simulation. In: *Mathematics and Computers in Simulation* 34 (1992), Nr. 3-4, pp. 283--296, cite: 43
- [Jen92] JENSEN, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol 1, Basic Concepts*. Berlin Heidelberg: Springer, 1992, cite: 18

REFERENCES

- [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, cite: 16
- [Jon75] JONG, K. A. D.: *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, PhD thesis, University of Michigan, PhD thesis, 1975, cite: 138
- [JPS93] JONES, D. R.; PERTTUNEN, C. D.; STUCKMAN, B. E.: Lipschitzian optimization without the Lipschitz constant. In: *Journal of optimization Theory and Applications* 79 (1993), Nr. 1, pp. 157--181, cite: 137
- [Kan12] KANG, P.: Modular implementation of dynamic algorithm switching in parallel simulations. In: *Cluster Computing* 15 (2012), Nr. 3, pp. 321--332, cite: 44, 120
- [KCC05] KURKOWSKI, S.; CAMP, T.; COLAGROSSO, M.: MANET simulation studies: the incredibles. In: *ACM SIGMOBILE Mobile Computing and Communications Review* 9 (2005), Nr. 4, pp. 50--61, cite: 38
- [KCR+09] KANG, P.; CAO, Y.; RAMAKRISHNAN, N.; RIBBENS, C.; VARADARAJAN, S.: Modular implementation of adaptive decisions in stochastic simulations. In: *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 995--1001, cite: 44, 120
- [KH96] KELLING, C.; HOMMEL, G.: Rare event simulation with an adaptive "RESTART" method in a Petri net modeling environment. In: *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems* IEEE, 1996, pp. 229--234, cite: 44
- [KKV04] KRAUSE, F.; KIND, C.; VOIGTSBERGER, J.: Adaptive modelling and simulation of product development processes. In: *CIRP Annals* 53 (2004), Nr. 1, pp. 135--138, cite: 44
- [KLP06] KORDON, F.; LINARD, A.; PAVIOT-ADET, E.: Optimized colored nets unfolding. In: *International Conference on Formal Techniques for Networked and Distributed Systems* Springer, 2006, pp. 339--355, cite: 35
- [KMS04] KIEHL, T.; MATTHEYSES, R.; SIMMONS, M.: Hybrid simulation of cellular behavior. In: *Bioinformatics* 20 (2004), pp. 316--322. -- ISSN 1367--4803, cite: 32
- [Koh21] KOHLHOFF, C.: *Boost.asio*. 2021. -- Available at https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio.html, cite: 84
- [KR12] KÖSTER, J.; RAHMANN, S.: Snakemake - a scalable bioinformatics workflow engine. In: *Bioinformatics* 28 (2012), Nr. 19, pp. 2520--2522, cite: 42
- [KRV15] KRUPITZER, C.; ROTH, F.; VANSYCKEL, S.; SCHIELE, G.; BECKER, C.: A survey on engineering approaches for self-adaptive systems. In: *Pervasive and Mobile Computing* 17 (2015), pp. 184--206, cite: 43
- [Kur72] KURTZ, T.G.: The relationship between stochastic and deterministic models for chemical reactions. In: *The Journal of Chemical Physics* 57 (1972), Nr. 7, pp. 2976--2978, cite: 25
- [Kur81] KURT, J.: Coloured Petri nets and the invariant-method. In: *Theoretical computer science* 14 (1981), Nr. 3, pp. 317--336, cite: 14
- [KW85] KLIR, G.; WAY, E.: Reconstructability analysis: aims, results, open problems. In: *Systems Research* 2 (1985), Nr. 2, pp. 141--163, cite: 43
- [KWD+04] KUMMER, O.; WIENBERG, F.; DUWIGNEAU, M.; J; SCHUMACHER; KÖHLER, M.; MOLDT, D.; RÖLKE, H.; VALK, R.: An Extensible Editor and Simulation Engine for Petri Nets: Renew. In: *ATPN 2004*, Springer, LNCS 3099, 2004, pp. 484--493, cite: 2
- [LCP+08] LI, H.; CAO, Y.; PETZOLD, L.; GILLESPIE, D.: Algorithms and software for stochastic simulation of biochemical reacting systems. In: *Biotechnol. Progr.* 24 (2008), Nr. 1, pp. 56--61, cite: 29
- [LHG19] LIU, F.; HEINER, M.; GILBERT, D.: Coloured Petri nets for multilevel, multiscale, and multidimensional modelling of biological systems. In: *Briefings in Bioinformatics* 20 (2019), Nr. 3, pp. 877--886. -- Published: 03 November 2017, cite: 14
- [LHR12] 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, cite: 40
- [Liu12] LIU, F.: *Colored Petri Nets for Systems Biology*, BTU Cottbus, Dep. of CS, PhD thesis, January 2012, cite: 1, 14, 17, 20, 21
- [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, cite: 89
- [MA99] MCADAMS, H.; ARKIN, A.: It's a noisy business! Genetic regulation at the nanomolar scale. In: *Trends in Genetics* 15 (1999), Nr. 2, pp. 65 -- 69, cite: 25, 31
- [MCK+18] MEDLEY, J.K.; CHOI, K.; KÖNIG, M.; SMITH, L.; GU, S.; HELLERSTEIN, J.; SEALFON, S.C.; SAURO, H.M.: Tellurium notebooks environment for reproducible dynamical modeling in systems biology. In: *PLoS computational biology* 14 (2018), Nr. 6, pp. e1006220, cite: 42
- [Mei19] MEINSMÄ, G.: Dimensional and scaling analysis. In: *SIAM review* 61 (2019), Nr. 1, pp. 159--184, cite: 24
- [MH91] MOHAMED, G.; HERMAN, T.: Adaptive programming. In: *IEEE Transactions on Software Engineering* 17 (1991), Nr. 9, pp. 911--921, cite: 43
- [MPT16] MARCHETTI, L.; PRIAMI, C.; THANH, V.H.: HRSSA-Efficient hybrid stochastic simulation for spatially homogeneous biochemical reaction networks. In: *Journal of Computational Physics* 317 (2016), pp. 301--317, cite: 82
- [MRH12] MARWAN, W.; ROHR, C.; HEINER, M.: Petri nets in Snoopy: A unifying framework for the graphical display, computational modelling, and simulation of bacterial regulatory networks. In: HELDEN, Jv (Eds.); TOUSSAINT, A (Eds.); THIEFFRY, D (Eds.): *Methods in Molecular Biology -- Bacterial Molecular Networks* Volume 804. Humana Press, 2012, Chapter 21, pp. 409--437, cite: 12
- [MRU11] MAUS, C.; RYBACKI, S.; UHRMÄCHER, A.M.: Rule-based multi-level modeling of cell biological systems. In: *BMC systems biology* 5 (2011), Nr. 1, pp. 1--20, cite: 40

REFERENCES


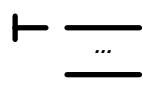




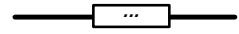

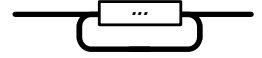
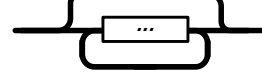


- [MSD+06] MCINTOSH, E.; SCHMIDT, F.; DE DINECHIN, F. et al.: Massive tracking on heterogeneous platforms. In: *9th International Computational Accelerator Physics Conference*, 2006, cite: 41
- [Mur89] MURATA, T.: Petri nets: Properties, analysis and applications. In: *Proceedings of the IEEE* 77 (1989), Nr. 4, pp. 541--580, cite: 1, 34
- [NAP19] NATIONAL ACADEMIES OF SCIENCES, ENGINEERING, AND MEDICINE: *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press, 2019, cite: 38
- [OSW69] OPPENHEIM, I.; SHULER, K.; WEISS, G.: Stochastic and deterministic formulation of chemical rate equations. In: *The Journal of Chemical Physics* 50 (1969), Nr. 1, pp. 460--466, cite: 25
- [Pah09] PAHLE, J.: Biochemical simulations: stochastic, approximate stochastic and hybrid approaches. In: *Brief Bioinform* 10 (2009), Nr. 1, pp. 53--64, cite: 25, 29, 31, 32
- [Pet62] PETRI, C.A.: *Kommunikation mit Automaten*, Universitt Hamburg, PhD thesis, 1962, cite: 7
- [PNML] PETRI NET MARKUP LANGUAGE (PNML): *Systems and software engineering -- High-level Petri nets -- Part 2: Transfer format*. 2009. -- ISO/IEC 15909--2:2011, cite: 88
- [PJC14] POLASEK, P.; JANOUSEK, V.; CESKA, M.: Petri Net Simulation as a Service. In: *PNSE@ Petri Nets*, 2014, pp. 353--362, cite: 2
- [VD09] ROSSUM, G. V.; DRAKE, F.L.: *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, cite: 137
- [RFC7159] Internet Engineering Task Force (IETF) 2018 The JavaScript Object Notation (JSON) Data Interchange Format. <https://tools.ietf.org/html/rfc7159>, cite: 48
- [RK08] RICE, M.; KULHARI, S.: A survey of static variable ordering heuristics for efficient BDD/MDD construction. In: *University of California, Tech. Rep* (2008), pp. 130, cite: 90
- [Roh17] ROHR, C.: *Simulative analysis of coloured extended stochastic Petri nets*, BTU Cottbus, Dep. of CS, PhD thesis, January 2017, cite: 10, 22, 25, 26, 81
- [Roh18] ROHR, C.: Discrete-Time Leap Method For Stochastic Simulation. In: *Fundamenta Informaticae* 160 (2018), Nr. 1-2, pp. 181--198. -- accepted for publication: May, 16 2017, cite: 81
- [Sch08] SCHULZ, K.: *An Extension of the Snoopy Software to Process and Manage Petri Net Animations (in German)*, BTU Cottbus, Dep. of CS, Bachelor thesis, November 2008, cite: 2, 79
- [Sch14] SCHWARICK, M.: *Symbolic on-the-fly analysis of stochastic Petri nets*, BTU Cottbus, Dep. of CS, PhD thesis, 2014, cite: 96
- [SFH99] STEINFELD, J.; FRANCISCO, J.; HASE, W.: *Chemical kinetics and dynamics*. Prentice Hall Upper Saddle River, NJ, 1999, cite: 23
- [SFR03] STEVENS, W.; FENNER, B.; RUDOFF, A.: *Unix Network Programming: The Sockets Networking API*. Volume Volume 1. 3rd Edition. Addison-Wesley Professional, 2003, cite: 83
- [SK05] SALIS, H.; KAZNESSIS, Y.: Accurate hybrid stochastic simulation of a system of coupled chemical or biochemical reactions. In: *J. Chem. Phys* 122 (2005), Nr. 5, cite: 32
- [SNT+13] SANDVE, G.K.; NEKRUTENKO, A.; TAYLOR, J.; HOVIG, E.: Ten simple rules for reproducible computational research. In: *PLoS Comput Biol* 9 (2013), Nr. 10, pp. e1003285, cite: 38, 39
- [Sol13] SOLGI, R.M. *geneticalgorithm*. <https://github.com/rmsolgi/geneticalgorithm>. 2020, cite: 137
- [SRH16] SCHWARICK, M.; ROHR, C.; HEINER, M.: MARCIE Manual / Brandenburg University of Technology Cottbus, Department of Computer Science. 2016 (02-16). -- Technical Report, cite: 40
- [SRF+20] SCHWARICK, M.; ROHR, C.; LIU, F.; ASSAF, G.; CHODAK, J.; HEINER, M.: Efficient Unfolding of Coloured Petri Nets Using Interval Decision Diagrams. In: *International Conference on Applications and Theory of Petri Nets and Concurrency* Springer, 2020, pp. 324--344, cite: 89, 91
- [ST11] SCHWARICK, M.; TOVCHIGRECHKO, A.: IDD-based model validation of biochemical networks. In: *Theoretical Computer Science* 412 (2011), Nr. 26, pp. 2884--2908, cite: 96
- [Val78] VALK, R.: Self-modifying nets, a natural extension of Petri nets. In: *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*. London, UK.; Springer-Verlag, 1978, pp. 464--476, cite: 10
- [VV+91] VILLEN-ALTAMIRANO, M.; VILLEN-ALTAMIRANO, J. et al.: RESTART: A method for accelerating rare event simulations. In: *Queueing, performance and Control in ATM* (1991), pp. 71--76, cite: 44
- [VPT+02] VETTERLING, T. W.; PRESS, W. H.; TEUKOLSKY, S. A.; FLANNERY, B. P.; BRIAN, P.: *Numerical Recipes Example Book (C++)*: *The Art of Scientific Computing*. Cambridge University Press, 2002, cite: 81
- [WAB+11a] WALTEMATH, D.; ADAMS, R.; BEARD, D. A.; BERGMANN, F.T.; BHALLA, U.S.; BRITTEN, R.; RANDALL; CHELLIAH; VIJAYALAKSHMI; COOLING; T, Michael; COOPER; JONATHAN; CRAMPIN; J, Edmund et al.: Minimum information about a simulation experiment (MIASE). In: *PLoS computational biology* 7 (2011), Nr. 4, pp. e1001122, cite: 38
- [WAB+11b] WALTEMATH, D.; ADAMS, R.; BERGMANN, F.T.; HUCKA, M.; KOLPAKOV, F.; MILLER, A.K.; MORARU, I.I.; NICKERSON, D.; SAHLE, S.; SNOEP, J.L. et al.: Reproducible computational biology experiments with SED-ML-the simulation experiment description markup language. In: *BMC systems biology* 5 (2011), Nr. 1, pp. 198, cite: 40
- [WHU17] WARNKE, T.; HELMS, T.; UHRMACHER, A.M.: Reproducible and flexible simulation experiments with ML-Rules and SESSL. In: *Bioinformatics* 34 (2017), 11, Nr. 8, pp. 1424--1427. -- ISSN 1367--4803, cite: 40
- [WSC+19] WEBER, L.; SAELENS, W.; CANNODT, R.; SONESON, C.; HAPFELMEIER, A.; GARDNER, P.; BOULESTEIX, A.; SAEYS, Y.; ROBINSON, M.: Essential guidelines for computational method benchmarking. In: *Genome biology* 20 (2019), Nr. 1, pp. 1--12, cite: 107
- [WUK+04] WOLKENHAUER, O.; ULLAH, M.; KOLCH, W.; CHO, K.: Modeling and simulation of intracellular dynamics: choosing an appropriate framework. In: *IEEE Trans. Nanobiosci.* 3 (2004), Nr. 3, pp. 200--207, cite: 29
- [ZDS14] ZUNKE, S.; DSOUZA, V.: Json vs xml: A comparative performance analysis of data exchange formats. In: *Int J Comput Sci Netw* 3 (2014), Nr. 4, pp. 257--261, cite: 49

Appendices






Appendix A

Grammar of Configuration Script

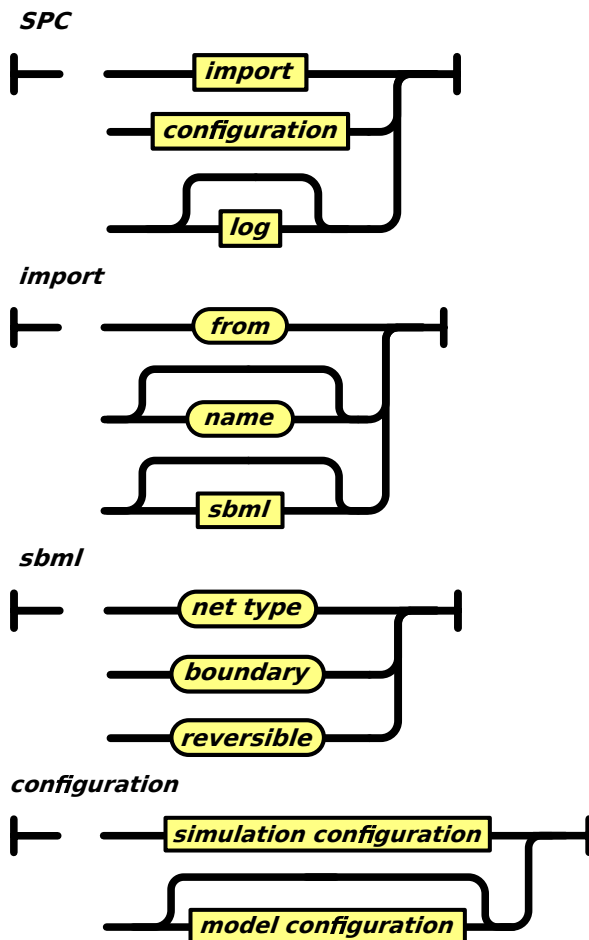
A.1 Graphical notations

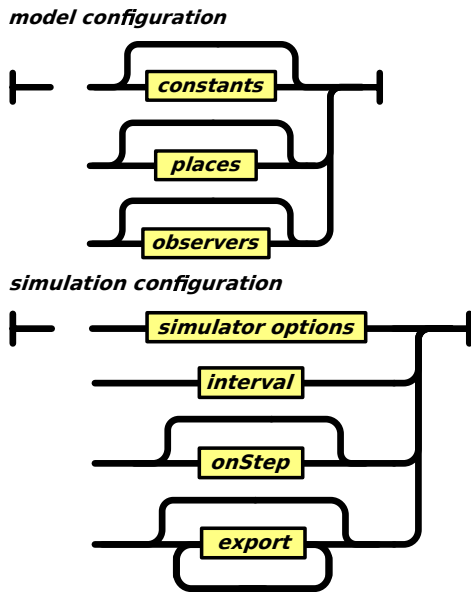
	definition entry point;
	parallel entry point - states that all entry points/paths must be chosen;
	definition end point;
	path - path to proceed;
	split path - path splitting states, only one direction can be chosen;
	join paths - combined paths become one;
	one occurrence of an entity;
	zero or one occurrence;
	one or many occurrences;
	zero or many occurrences;
	<i>SPC</i> properties - - properties used to define/configure an experiment;
	grammar definitions

A. GRAMMAR OF CONFIGURATION SCRIPT

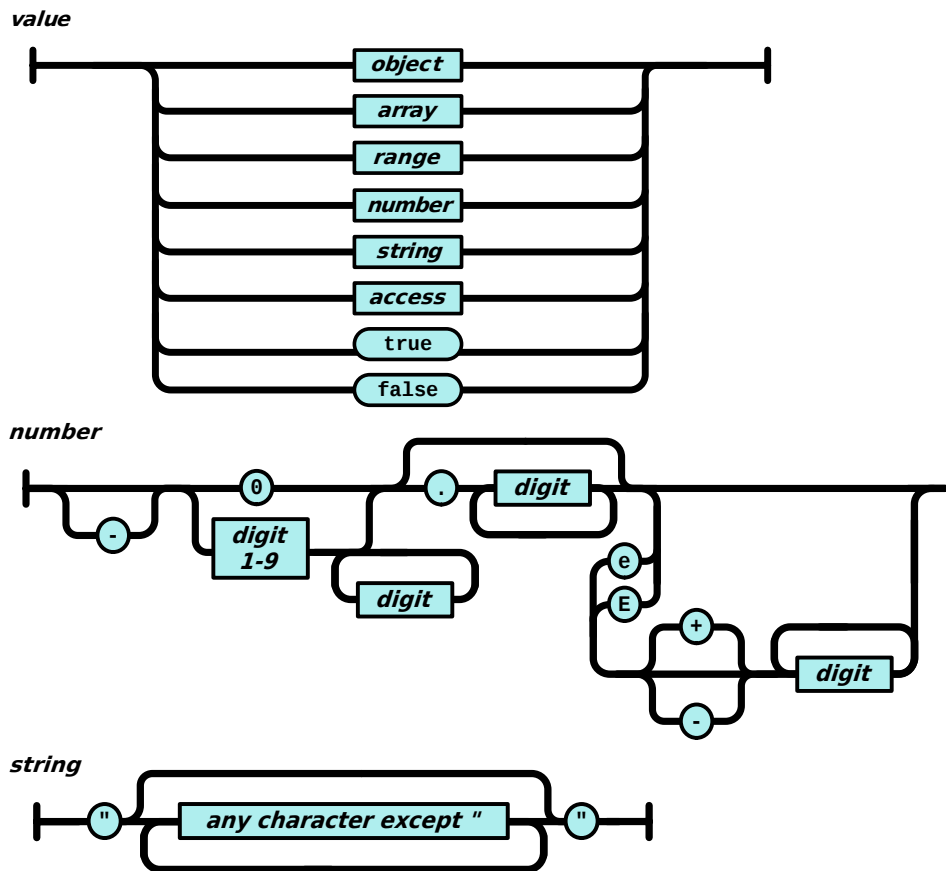
	SPC basic property - represents a property that does not associate any other properties;
	SPC complex property - represents a property that associate one or more basic properties;
	grammar operator - represents an operator;
	grammar literal - represents a constant/fixed value;
	grammar rule - represents a meta variable/nonterminal symbol;

A.2 Main *SPC* Objects



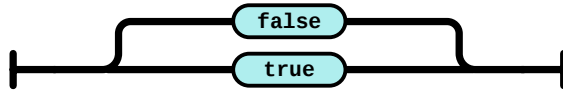


A.3 Basic definitions

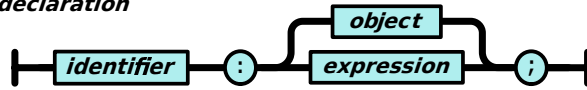


A. GRAMMAR OF CONFIGURATION SCRIPT

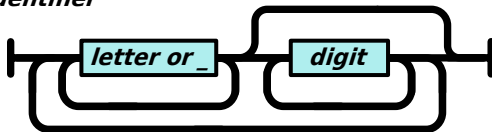
boolean value



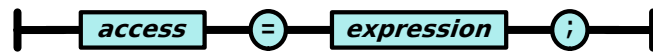
declaration



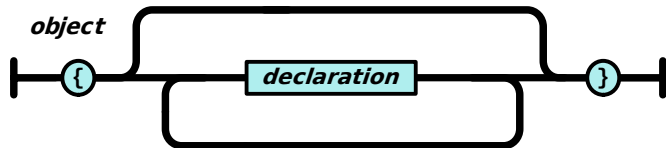
identifier



assign



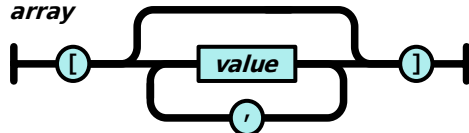
object



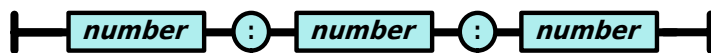
access



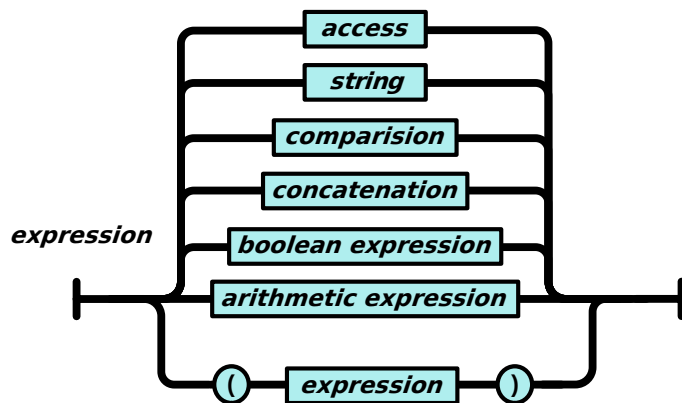
array



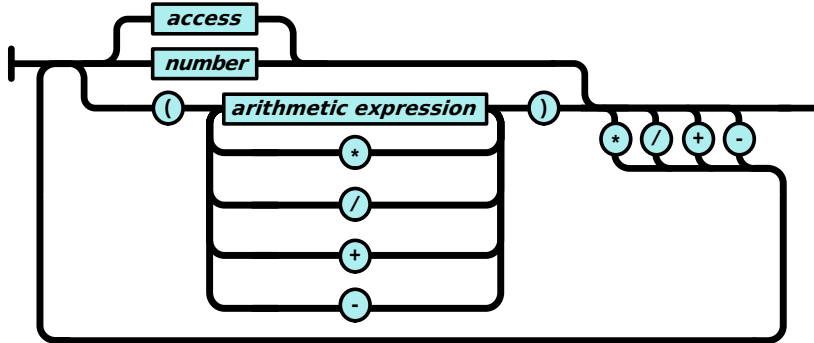
range



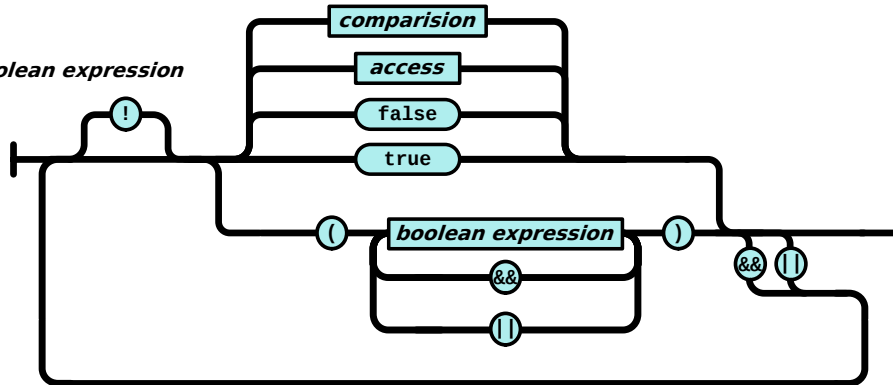
A.4 Expressions



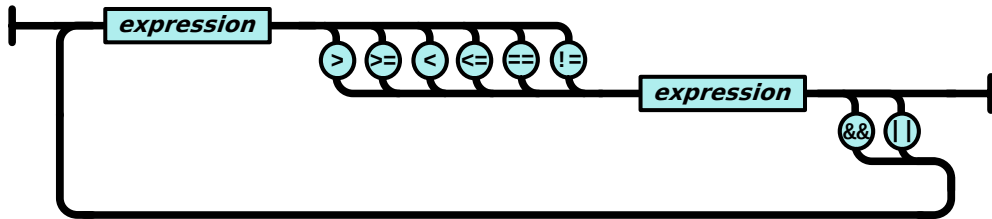
arithmetic expression



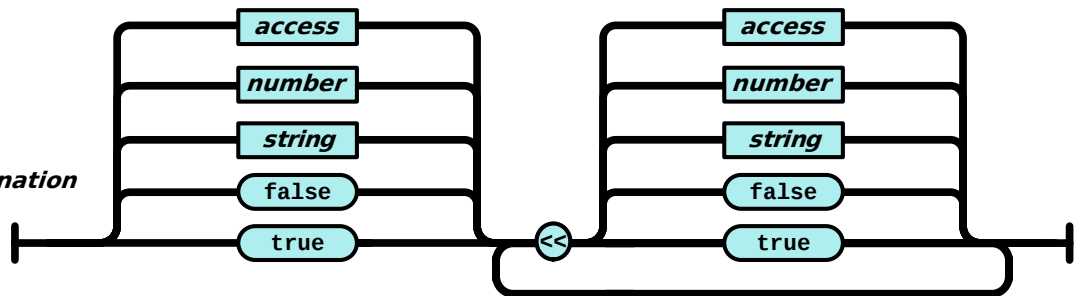
boolean expression



comparision

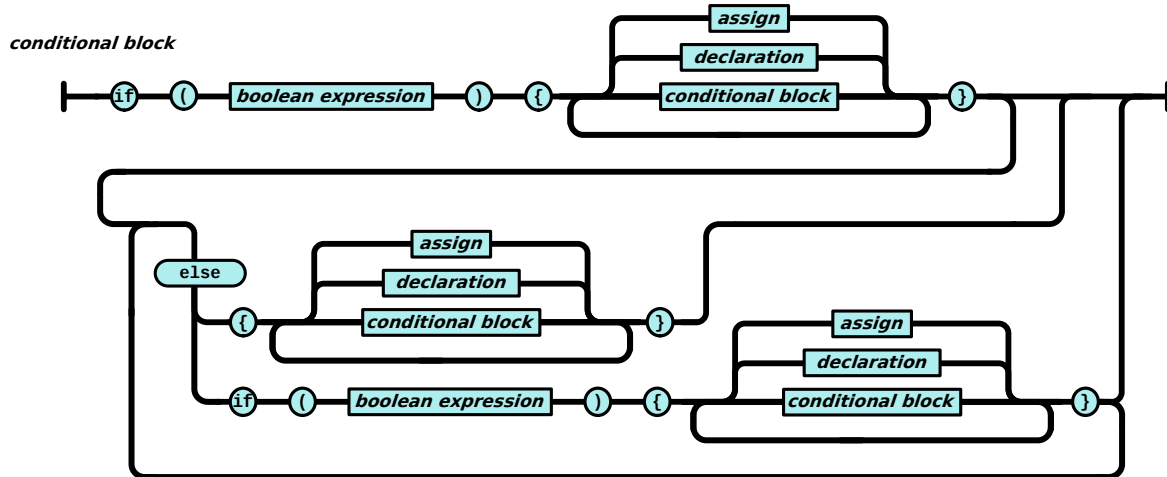


concatenation

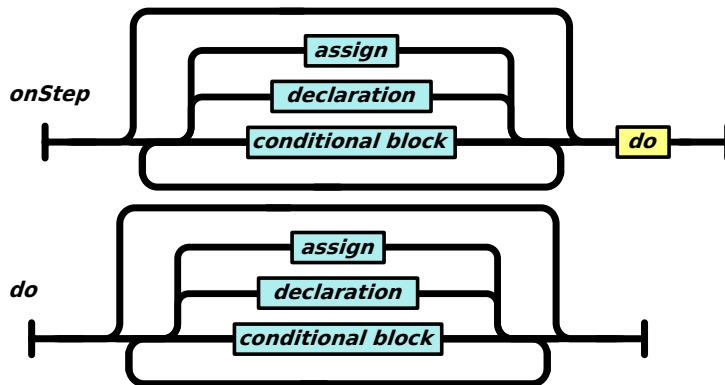


A. GRAMMAR OF CONFIGURATION SCRIPT

A.5 Conditional Block



A.6 onStep



Appendix B

Source Code: Heuristic Method of Parameter Optimization

B.1 SIR Model In ANDL Format: SIR-SPN.andl

```
1 spn [SIR-SPN]
2 {
3 constants:
4 all:
5 double k_infect_a = 5.0e-5;
6 double k_infect_b = 5.0e-6;
7 double k_recover_a = 1.0e-1;
8 double k_recover_b = 1.0e-2;
9
10 places:
11 discrete:
12 Infected_A = 1;
13 Infected_B = 1;
14 Recovered_A = 0;
15 Recovered_B = 0;
16 SusceptiblePopulation_A = 50000;
17 SusceptiblePopulation_B = 100000;
18
19 transitions:
20 stochastic:
21 Infect_0
22 :
23 : [Infected_A + 2] & [SusceptiblePopulation_A - 1] & [Infected_A - 1]
24 : MassAction(k_infect_a)
25 ;
26 Infect_1
27 :
28 : [Infected_B + 2] & [SusceptiblePopulation_B - 1] & [Infected_B - 1]
29 : MassAction(k_infect_b)
30 ;
31 Recover_0
32 :
33 : [Recovered_A + 1] & [Infected_A - 1]
34 : MassAction(k_recover_a)
35 ;
36 Recover_1
37 :
38 : [Recovered_B + 1] & [Infected_B - 1]
39 : MassAction(k_recover_b)
40 ;
41 }
```

B. SOURCE CODE: HEURISTIC METHOD OF PARAMETER OPTIMIZATION

B.2 SPC Configuration Template: SIR-CPN-spc.tmp

```
1 import: {
2   from: "./model/SIR-SPN.and1";
3 }
4
5 configuration: {
6
7   model: [[%s]];
8
9   simulation: {
10    name: "SIR";
11    type: continuous: {
12      solver:
13      BDF: {
14        semantic: "adapt";
15        iniStep: 0.1;
16        linSolver: "CVDense";
17        relTol: 1e-5;
18        absTol: 1.0e-10;
19        autoStepSize: false;
20        reductResultingODE: true;
21        checkNegativeVal: false;
22        outputNoiseVal: false;
23      }
24    }
25
26    interval: 0:100:1000;
27
28    export: {
29      places: [];
30      csv: {
31        sep: ";";
32        file: "simresults/" << name << "_"
33          << configuration.model
34          << ".csv";
35      }
36    }
37 }
38 }
```

B.3 Experiment Set-up in Python: optimization.py

```

1  '''
2
3  Copyright 2021 Jacek Chodak
4
5  Permission is hereby granted, free of charge, to any person obtaining a copy of
6  this software and associated documentation files (the "Software"), to deal in
7  the Software without restriction, including without limitation the rights to use,
8  copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the
9  Software, and to permit persons to whom the Software is furnished to do so,
10 subject to the following conditions:
11
12 The above copyright notice and this permission notice shall be included in all
13 copies or substantial portions of the Software.
14
15 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
17 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
18 THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
19 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
20 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
21 SOFTWARE.
22
23 '''
24
25 #####
26 #####
27 #####
28
29 import numpy as np
30 from geneticalgorithmjch import geneticalgorithm as ga
31 import subprocess
32 import csv
33 import time
34 import matplotlib.pyplot as plt
35
36 #####
37 #
38 # CSV
39 #
40 #####
41 def readCSV(filename):
42     tsB = []
43     tsR = []
44
45     try:
46         with open(filename) as csvDataFile:
47             csvReader = csv.DictReader(csvDataFile, delimiter = ";")
48             for row in csvReader:
49                 tsB.append(row["Infected_B"])
50                 tsR.append(row["Recovered_B"])
51     except FileNotFoundError:
52         print("open::FileNotFoundError: %s" % (filename))
53
54     tdB = np.array(tsB).astype(np.float)
55     tdR = np.array(tsR).astype(np.float)
56
57     return tdB, tdR

```

B. SOURCE CODE: HEURISTIC METHOD OF PARAMETER OPTIMIZATION

```
58 #####
59 #
60 # Fitness function
61 #
62 #####
63 def f(X):
64
65     strTemplatePopulationInhabitant = "inhab%d: {\n"\
66         "constants: { all: {k_infect_b: %f; k_recover_b: %f;}}\n"\
67         "    }"
68
69
70     iIdx = 0
71     strPopulationInhabitant = ""
72
73     strPopulationInhabitant += strTemplatePopulationInhabitant\
74         % (iIdx,
75           X[0],
76           X[1]
77          )
78
79     # read conf template
80     with open("SIR-CPN-spc.tmp", "r") as templateFile:
81         strTemplateConf = templateFile.read().replace('\n', '')
82
83     strConf = strTemplateConf % (strPopulationInhabitant)
84     strSpikeCmd = "conf -s='%s' exe -p=4 -process=1"
85     ## remove whitespece
86     ## (space, tab, newline, and so on) -> sentence = ''.join(sentence.split())
87     strSpikeCmd = strSpikeCmd % (''.join(strConf.split()))
88
89     returnCode = subprocess.call(["./spike-release", strSpikeCmd],
90                                 stdout=None, stderr=None)
91     print("SPIKE:", returnCode)
92
93     ## Compare
94     strFile = "./simresults/SIR_inhab%d.csv" % (iIdx)
95     print(strFile)
96     tdInfec, tdRecov = readCSV(strFile)
97     dFit = 0.0;
98     dFitMax = 0;
99
100     ## Percentage Difference
101     for iIdx in range(len(m_tdInfec)):
102         if m_tdInfec[iIdx] + tdInfec[iIdx] > 0:
103             dFit = dFit + abs(m_tdInfec[iIdx] - tdInfec[iIdx]) / \
104                 (m_tdInfec[iIdx] + tdInfec[iIdx]) / 2 * 100
105
106     dFit /= len(m_tdInfec);
107
108     print("dFit: ", dFit)
109     print("k_infect: ", X[0])
110     print("k_recover: ", X[1])
111
112     return dFit
```


B.3 Experiment Set-up in Python: optimization.py

```
113 #####
114 #
115 # Report progress callback function
116 #
117 #####
118 def onProgress(genalg):
119     print('\r The best solution found:\n %s' % (genalg.best_variable))
120     print('\n\n Objective function:\n %s\n' % (genalg.best_function))
121
122     re=np.array(genalg.report)
123     rePop = np.array(genalg.reportPop)
124     plt.cla()
125     plt.plot(re)
126     plt.xlabel("Iteration: %s" % genalg.counter)
127     plt.ylabel("Objective function: %s" % genalg.best_function)
128     plt.title("Genetic Algorithm: %s" % genalg.best_variable)
129     #plt.show()
130     plt.draw()
131     plt.pause(0.0001)
132     with open("progress.csv", "w", newline = "\n") as file:
133         #with open("progress-%s.csv" % time.time(), "w", newline = "\n") as file:
134             csvwriter = csv.writer(file, delimiter = ";")
135             csvwriter.writerow(["fit", "k_infect", "k_recover"])
136             iIdx = 0
137             for x in re :
138                 csvwriter.writerow([x, rePop[iIdx][0], rePop[iIdx][1]])
139                 iIdx += 1
140
141 #####
142 #
143 # Experiment: Case A
144 #
145 #####
146 def caseA():
147     varbound=np.array([[1.0e-6, 1.0e-5], [1.0e-3, 1.0e-1]])
148     vartype=np.array([["real"], ["real"]])
149
150
151     algorithm_param = {'max_num_iteration': 20,\
152                       'population_size': 50,\
153                       'mutation_probability': 0.4,\
154                       'elit_ratio': 0.01,\
155                       'crossover_probability': 0.5,\
156                       'parents_portion': 0.3,\
157                       'crossover_type': 'uniform',\
158                       'max_iteration_without_improv': None}
159
160     model = ga(function=f,\
161               onProgress=onProgress,\
162               dimension=2,\
163               #variable_type='real',\
164               variable_type_mixed=vartype,\
165               variable_boundaries=varbound,\
166               function_timeout=30,\
167               algorithm_parameters=algorithm_param,\
168               progress_bar=True)
169
170     return model
```

B. SOURCE CODE: HEURISTIC METHOD OF PARAMETER OPTIMIZATION

```
171 #####
172 #
173 # Experiment: Case B
174 #
175 #####
176 def caseB():
177     varbound=np.array([[1.0e-7, 1.0], [1.0e-7, 1.0]])
178     vartype=np.array(["real", "real"])
179
180     algorithm_param = {'max_num_iteration': 20,\
181                       'population_size': 50,\
182                       'mutation_probability': 0.6,\
183                       'elit_ratio': 0.01,\
184                       'crossover_probability': 0.5,\
185                       'parents_portion': 0.3,\
186                       'crossover_type': 'uniform',\
187                       'max_iteration_without_improv': None}
188
189     model = ga(function=f,\
190               onProgress=onProgress,\
191               dimension=2,\
192               #variable_type='real',\
193               variable_type_mixed=vartype,\
194               variable_boundaries=varbound,\
195               function_timeout=30,\
196               algorithm_parameters=algorithm_param,\
197               progress_bar=True)
198
199     return model
200
201 #####
202 #
203 # Main
204 #
205 #####
206
207 ## Refernce data trace
208 m_tdInfec, m_tdRecov = readCSV("./sim/data/SIR-SPN_continuous_BDF.csv")
209
210 ## Select example case
211 model = caseA()
212 #model = caseB()
213
214 model.run()
215
216 solution = model.best_variable
217
218 print(model.output_dict)
```

B.4 Genetic Algorithm Library for Python: `geneticalgorithmjch.py`

```

1  '''
2
3  Copyright 2020 Ryan (Mohammad) Solgi
4
5  Permission is hereby granted, free of charge, to any person obtaining a copy of
6  this software and associated documentation files (the "Software"), to deal in
7  the Software without restriction, including without limitation the rights to use,
8  copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the
9  Software, and to permit persons to whom the Software is furnished to do so,
10 subject to the following conditions:
11
12 The above copyright notice and this permission notice shall be included in all
13 copies or substantial portions of the Software.
14
15 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
17 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
18 THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
19 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
20 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
21 SOFTWARE.
22
23 '''
24
25 #####
26 #####
27 #####
28
29 import numpy as np
30 import sys
31 import time
32 from func_timeout import func_timeout, FunctionTimedOut
33 import matplotlib.pyplot as plt
34
35 #####
36 #####
37 #####
38
39 class geneticalgorithm():
40
41     ''' Genetic Algorithm (Elitist version) for Python
42
43     An implementation of elitist genetic algorithm for solving problems with
44     continuous, integers, or mixed variables.
45     Implementation and output:
46     methods:
47         run(): implements the genetic algorithm
48     outputs:
49         output_dict: a dictionary including the best set of variables
50         found and the value of the given function associated to it.
51         {'variable': , 'function': }
52         report: a list including the record of the progress of the
53         algorithm over iterations
54
55     '''

```

B. SOURCE CODE: HEURISTIC METHOD OF PARAMETER OPTIMIZATION

```
56 #####
57 def __init__(self, function, onProgress, dimension, variable_type='bool', \
58             variable_boundaries=None, \
59             variable_type_mixed=None, \
60             function_timeout=10, \
61             algorithm_parameters={'max_num_iteration': None, \
62                                 'population_size':100, \
63                                 'mutation_probability':0.1, \
64                                 'elit_ratio': 0.01, \
65                                 'crossover_probability': 0.5, \
66                                 'parents_portion': 0.3, \
67                                 'crossover_type':'uniform', \
68                                 'max_iteration_without_improv':None}, \
69             convergence_curve=True, \
70             progress_bar=True):
71     '''
72     @param function <Callable> - the given objective function to be minimized
73     NOTE: This implementation minimizes the given objective function.
74     (For maximization multiply function by a negative sign: the absolute
75     value of the output would be the actual objective function)
76
77     @param dimension <integer> - the number of decision variables
78
79     @param variable_type <string> - 'bool' if all variables are Boolean;
80     'int' if all variables are integer; and 'real' if all variables are
81     real value or continuous (for mixed type see @param variable_type_mixed)
82
83     @param variable_boundaries <numpy array/None> - Default None; leave it
84     None if variable_type is 'bool'; otherwise provide an array of tuples
85     of length two as boundaries for each variable;
86     the length of the array must be equal dimension. For example,
87     np.array([0,100],[0,200]) determines lower boundary 0 and upper boundary
88     100 for first and upper boundary 200 for second variable where dimension is 2.
89
90     @param variable_type_mixed <numpy array/None> - Default None; leave it
91     None if all variables have the same type; otherwise this can be used to
92     specify the type of each variable separately. For example if the first
93     variable is integer but the second one is real the input is:
94     np.array(['int'],['real']). NOTE: it does not accept 'bool'. If variable
95     type is Boolean use 'int' and provide a boundary as [0,1]
96     in variable_boundaries. Also if variable_type_mixed is applied,
97     variable_boundaries has to be defined.
98
99     @param function_timeout <float> - if the given function does not provide
100     output before function_timeout (unit is seconds) the algorithm raise error.
101     For example, when there is an infinite loop in the given function.
102
103     @param algorithm_parameters:
104         @ max_num_iteration <int> - stopping criteria of the genetic algorithm (GA)
105         @ population_size <int>
106         @ mutation_probability <float in [0,1]>
107         @ elit_ratio <float in [0,1]>
108         @ crossover_probability <float in [0,1]>
109         @ parents_portion <float in [0,1]>
110         @ crossover_type <string> - Default is 'uniform'; 'one_point' or
111         'two_point' are other options
112         @ max_iteration_without_improv <int> - maximum number of
113         successive iterations without improvement. If None it is ineffective
114
115     @param convergence_curve <True/False> - Plot the convergence curve or not
116     Default is True.
117     @progress_bar <True/False> - Show progress bar or not. Default is True.
118
119     for more details and examples of implementation please visit:
120     https://github.com/rmsolgi/geneticalgorithm
121     '''
122     '''
```

B.4 Genetic Algorithm Library for Python: `geneticalgorithmjch.py`

```
123     self.__name__=geneticalgorithm
124     #####
125     # input function
126     assert (callable(function)), "function must be callable"
127
128     self.f=function
129     #####
130     # input function onProgress
131     assert (callable(onProgress)), "function onProgress must be callable"
132
133     self.onProgressDo=onProgress
134     #####
135     #dimension
136
137     self.dim=int(dimension)
138
139     #####
140     # input variable type
141
142     assert(variable_type=='bool' or variable_type=='int' or\
143            variable_type=='real'), \
144            "\n variable_type must be 'bool', 'int', or 'real'"
145     #####
146     # input variables' type (MIXED)
147
148     if variable_type_mixed is None:
149
150         if variable_type=='real':
151             self.var_type=np.array([[ 'real' ]]*self.dim)
152         else:
153             self.var_type=np.array([[ 'int' ]]*self.dim)
154
155     else:
156         assert (type(variable_type_mixed).__module__=='numpy'),\
157                "\n variable_type must be numpy array"
158         assert (len(variable_type_mixed) == self.dim), \
159                "\n variable_type must have a length equal dimension."
160
161         for i in variable_type_mixed:
162             assert (i=='real' or i=='int'),\
163                    "\n variable_type_mixed is either 'int' or 'real' "+\
164                    "ex: ['int', 'real', 'real']"+\
165                    "\n for 'boolean' use 'int' and specify boundary as [0,1]"
166
167         self.var_type=variable_type_mixed
168
169     #####
170     # input variables' boundaries
171
172     if variable_type!='bool' or type(variable_type_mixed).__module__=='numpy':
173
174         assert (type(variable_boundaries).__module__=='numpy'),\
175                "\n variable_boundaries must be numpy array"
176
177         assert (len(variable_boundaries)==self.dim),\
178                "\n variable_boundaries must have a length equal dimension"
179
180
181         for i in variable_boundaries:
182             assert (len(i) == 2), \
183                    "\n boundary for each variable must be a tuple of length two."
184             assert (i[0]<=i[1]),\
185                    "\n lower_boundaries must be smaller than upper_boundaries "+\
186                    "[lower,upper]"
187         self.var_bound=variable_boundaries
188     else:
189         self.var_bound=np.array([[0,1]]*self.dim)
```

B. SOURCE CODE: HEURISTIC METHOD OF PARAMETER OPTIMIZATION

```
190 #####
191 #Timeout
192 self.funtimeout=float(function_timeout)
193 #####
194 #convergence_curve
195 if convergence_curve==True:
196     self.convergence_curve=True
197 else:
198     self.convergence_curve=False
199 #####
200 #progress_bar
201 if progress_bar==True:
202     self.progress_bar=True
203 else:
204     self.progress_bar=False
205 #####
206 #####
207 # input algorithm's parameters
208
209 self.param=algorithm_parameters
210 self.pop_s=int(self.param['population_size'])
211
212 assert (self.param['parents_portion']<=1\
213         and self.param['parents_portion']>=0), \
214 "parents_portion must be in range [0,1]"
215
216 self.par_s=int(self.param['parents_portion']*self.pop_s)
217 trl=self.pop_s-self.par_s
218 if trl % 2 != 0:
219     self.par_s+=1
220
221 self.prob_mut=self.param['mutation_probability']
222
223 assert (self.prob_mut<=1 and self.prob_mut>=0), \
224 "mutation_probability must be in range [0,1]"
225
226 self.prob_cross=self.param['crossover_probability']
227 assert (self.prob_cross<=1 and self.prob_cross>=0), \
228 "mutation_probability must be in range [0,1]"
229
230 assert (self.param['elit_ratio']<=1 and self.param['elit_ratio']>=0), \
231 "elit_ratio must be in range [0,1]"
232
233 trl=self.pop_s*self.param['elit_ratio']
234 if trl<1 and self.param['elit_ratio']>0:
235     self.num_elit=1
236 else:
237     self.num_elit=int(trl)
238
239 assert(self.par_s>=self.num_elit), \
240 "\n number of parents must be greater than number of elits"
241
242 if self.param['max_num_iteration']==None:
243     self.iterate=0
244     for i in range (0,self.dim):
245         if self.var_type[i]=='int':
246             self.iterate+=(self.var_bound[i][1]-self.var_bound[i][0])* \
247                 self.dim*(100/self.pop_s)
248         else:
249             self.iterate+=(self.var_bound[i][1]-self.var_bound[i][0])* \
250                 50*(100/self.pop_s)
251     self.iterate=int(self.iterate)
252     if (self.iterate*self.pop_s)>10000000:
253         self.iterate=10000000/self.pop_s
254 else:
255     self.iterate=int(self.param['max_num_iteration'])
```

B.4 Genetic Algorithm Library for Python: geneticalgorithmjch.py

```
256 self.c_type=self.param['crossover_type']
257 assert (self.c_type=='uniform' or self.c_type=='one_point' or\
258         self.c_type=='two_point'),\
259        "\n crossover_type must 'uniform', 'one_point', or 'two_point' Enter string"
260
261
262 self.stop_mniwi=False
263 if self.param['max_iteration_without_improv']==None:
264     self.mniwi=self.iterate+1
265 else:
266     self.mniwi=int(self.param['max_iteration_without_improv'])
267
268
269 #####
270 def run(self):
271
272
273 #####
274 # Initial Population
275
276 self.integers=np.where(self.var_type=='int')
277 self.reals=np.where(self.var_type=='real')
278
279
280
281 pop=np.array([np.zeros(self.dim+1)]*self.pop_s)
282 solo=np.zeros(self.dim+1)
283 var=np.zeros(self.dim)
284
285 for p in range(0,self.pop_s):
286
287     for i in self.integers[0]:
288         var[i]=np.random.randint(self.var_bound[i][0],\
289                                self.var_bound[i][1]+1)
290     solo[i]=var[i].copy()
291     for i in self.reals[0]:
292         var[i]=self.var_bound[i][0]+np.random.random()*\
293              (self.var_bound[i][1]-self.var_bound[i][0])
294     solo[i]=var[i].copy()
295
296
297     obj=self.sim(var)
298     solo[self.dim]=obj
299     pop[p]=solo.copy()
300
301 #####
302
303 #####
304 # Report
305 self.report=[]
306 self.reportPop=[]
307 self.test_obj=obj
308 self.best_variable=var.copy()
309 self.best_function=obj
310 #####
311
312 t=1
313 counter=0
314 self.counter = counter
315 while t<=self.iterate:
316
317     if self.progress_bar==True:
318         self.progress(t,self.iterate,status="GA is running...")
319     #####
320     #Sort
321     pop = pop[pop[:,self.dim].argsort()]
```

B. SOURCE CODE: HEURISTIC METHOD OF PARAMETER OPTIMIZATION

```
322     if pop[0,self.dim]<self.best_function:
323         counter=0
324         self.best_function=pop[0,self.dim].copy()
325         self.best_variable=pop[0,: self.dim].copy()
326     else:
327         counter+=1
328     self.counter = counter
329     #####
330     # Report
331
332     self.report.append(pop[0,self.dim])
333     self.reportPop.append(pop[0,: self.dim])
334
335
336     #####
337     # Normalizing objective function
338
339     normobj=np.zeros(self.pop_s)
340
341     minobj=pop[0,self.dim]
342     if minobj<0:
343         normobj=pop[:,self.dim]+abs(minobj)
344
345     else:
346         normobj=pop[:,self.dim].copy()
347
348     maxnorm=np.amax(normobj)
349     normobj=maxnorm-normobj+1
350
351     #####
352     # Calculate probability
353
354     sum_normobj=np.sum(normobj)
355     prob=np.zeros(self.pop_s)
356     prob=normobj/sum_normobj
357     cumprob=np.cumsum(prob)
358
359     #####
360     # Select parents
361     par=np.array([np.zeros(self.dim+1)]*self.par_s)
362
363     for k in range(0,self.num_elit):
364         par[k]=pop[k].copy()
365     for k in range(self.num_elit,self.par_s):
366         index=np.searchsorted(cumprob,np.random.random())
367         par[k]=pop[index].copy()
368
369     ef_par_list=np.array([False]*self.par_s)
370     par_count=0
371     while par_count==0:
372         for k in range(0,self.par_s):
373             if np.random.random()<=self.prob_cross:
374                 ef_par_list[k]=True
375                 par_count+=1
376
377     ef_par=par[ef_par_list].copy()
378
379     #####
380     #New generation
381     pop=np.array([np.zeros(self.dim+1)]*self.pop_s)
```


B.4 Genetic Algorithm Library for Python: geneticalgorithmjch.py

```
382         for k in range(0,self.par_s):
383             pop[k]=par[k].copy()
384
385         for k in range(self.par_s, self.pop_s, 2):
386             r1=np.random.randint(0,par_count)
387             r2=np.random.randint(0,par_count)
388             pvar1=ef_par[r1,: self.dim].copy()
389             pvar2=ef_par[r2,: self.dim].copy()
390
391             ch=self.cross(pvar1,pvar2,self.c_type)
392             ch1=ch[0].copy()
393             ch2=ch[1].copy()
394
395             ch1=self.mut(ch1)
396             ch2=self.mutmiddle(ch2,pvar1,pvar2)
397             solo[: self.dim]=ch1.copy()
398             obj=self.sim(ch1)
399             solo[self.dim]=obj
400             pop[k]=solo.copy()
401             solo[: self.dim]=ch2.copy()
402             obj=self.sim(ch2)
403             solo[self.dim]=obj
404             pop[k+1]=solo.copy()
405         #####
406         t+=1
407         if counter > self.mniwi:
408             pop = pop[pop[: ,self.dim].argsort()]
409             if pop[0,self.dim]>=self.best_function:
410                 t=self.iterate
411                 if self.progress_bar==True:
412                     self.progress(t,self.iterate,status="GA is running...")
413                 time.sleep(2)
414                 t+=1
415                 self.stop_mniwi=True
416
417         #####
418         #Sort
419         pop = pop[pop[: ,self.dim].argsort()]
420
421         if pop[0,self.dim]<self.best_function:
422
423             self.best_function=pop[0,self.dim].copy()
424             self.best_variable=pop[0,: self.dim].copy()
425
426         #####
427         # Report
428
429         self.report.append(pop[0,self.dim])
430
431         #self.reportPop.append(pop)
432         self.reportPop.append(pop[0,: self.dim])
433
434         self.output_dict={'variable': self.best_variable, 'function':\
435                             self.best_function}
436         if self.progress_bar==True:
437             show=' '*100
438             sys.stdout.write('\r%s' % (show))
439             ## JCH-S
440             self.onProgress()
441             ## JCH-E
```

B. SOURCE CODE: HEURISTIC METHOD OF PARAMETER OPTIMIZATION

```
442 sys.stdout.write('\r The best solution found:\n %s' % (self.best_variable))
443 sys.stdout.write('\n\n Objective function:\n %s\n' % (self.best_function))
444 sys.stdout.flush()
445 re=np.array(self.report)
446 if self.convergence_curve==True:
447     plt.plot(re)
448     plt.xlabel('Iteration')
449     plt.ylabel('Objective function')
450     plt.title('Genetic Algorithm')
451     plt.show()
452
453 if self.stop_mniwi==True:
454     sys.stdout.write('\nWarning: GA is terminated due to the'+\
455                     ' maximum number of iterations without'+\
456                     ' improvement was met!')
457
458 #####
459 #####
460 def cross(self,x,y,c_type):
461
462     ofs1=x.copy()
463     ofs2=y.copy()
464
465     if c_type=='one_point':
466         ran=np.random.randint(0,self.dim)
467         for i in range(0,ran):
468             ofs1[i]=y[i].copy()
469             ofs2[i]=x[i].copy()
470
471     if c_type=='two_point':
472
473         ran1=np.random.randint(0,self.dim)
474         ran2=np.random.randint(ran1,self.dim)
475
476         for i in range(ran1,ran2):
477             ofs1[i]=y[i].copy()
478             ofs2[i]=x[i].copy()
479
480     if c_type=='uniform':
481
482         for i in range(0, self.dim):
483             ran=np.random.random()
484             if ran < 0.5:
485                 ofs1[i]=y[i].copy()
486                 ofs2[i]=x[i].copy()
487
488     return np.array([ofs1,ofs2])
489 #####
490 def mut(self,x):
491
492     for i in self.integers[0]:
493         ran=np.random.random()
494         if ran < self.prob_mut:
495
496             x[i]=np.random.randint(self.var_bound[i][0],\
497                                   self.var_bound[i][1]+1)
498
499     for i in self.reals[0]:
500         ran=np.random.random()
501         if ran < self.prob_mut:
502
503             x[i]=self.var_bound[i][0]+np.random.random()*\
504                 (self.var_bound[i][1]-self.var_bound[i][0])
505
506     return x
```

B.4 Genetic Algorithm Library for Python: geneticalgorithmjch.py

```
507 #####
508 def mutmidle(self, x, p1, p2):
509     for i in self.integers[0]:
510         ran=np.random.random()
511         if ran < self.probab_mut:
512             if p1[i]<p2[i]:
513                 x[i]=np.random.randint(p1[i],p2[i])
514             elif p1[i]>p2[i]:
515                 x[i]=np.random.randint(p2[i],p1[i])
516             else:
517                 x[i]=np.random.randint(self.var_bound[i][0],\
518                                     self.var_bound[i][1]+1)
519
520     for i in self.reals[0]:
521         ran=np.random.random()
522         if ran < self.probab_mut:
523             if p1[i]<p2[i]:
524                 x[i]=p1[i]+np.random.random()*(p2[i]-p1[i])
525             elif p1[i]>p2[i]:
526                 x[i]=p2[i]+np.random.random()*(p1[i]-p2[i])
527             else:
528                 x[i]=self.var_bound[i][0]+np.random.random()*\
529                     (self.var_bound[i][1]-self.var_bound[i][0])
530     return x
531 #####
532 def evaluate(self):
533     return self.f(self.temp)
534 #####
535 def sim(self,X):
536     self.temp=X.copy()
537     obj=None
538     try:
539         obj=func_timeout(self.funtimeout,self.evaluate)
540     except FunctionTimedOut:
541         print("given function is not applicable")
542     assert (obj!=None), "After "+str(self.funtimeout)+" seconds delay "+\
543             "func_timeout: the given function does not provide any output"
544     return obj
545
546 #####
547 def progress(self, count, total, status=''):
548     bar_len = 50
549     filled_len = int(round(bar_len * count / float(total)))
550
551     percents = round(100.0 * count / float(total), 1)
552     bar = '|' * filled_len + '_' * (bar_len - filled_len)
553
554     sys.stdout.write('\r%s %s%% %s' % (bar, percents, '%', status))
555     sys.stdout.flush()
556
557     ## JCH-S
558     self.onProgress()
559     ## JCH-E
560
561 #####
562 def onProgress(self):
563     return self.onProgressDo(self)
564
565 #####
566 #####
```


Appendix C

A Quick Guide to SPC

A quick guide to SPC

Cheat Sheet: v1.0.0rc2
Spike: v1.6.x
Jacek Chodak, Monika Hejner

Basics

SPC (Spike Configuration) is a structured data-oriented configuration script. The SPC syntax is concise and human readable, requiring as little effort as possible from the user.

Keywords

For purpose of configuration Spike reserves some names which can be used as variable identifiers: **configuration, else, export, if, import, interval, log, model, range, simulation, solver.**

White Space

Multiple spaces are ignored. To make a configuration script more readable, spaces can be used for formatting. The following lines are equivalent:

```
1 x:1;
2 x: 1;
```

Comments

Comments are not evaluated. They can be used to make notes or to prevent part of the configuration from being evaluated when testing an alternative configuration. SPC allows for two types of comments:

- single line comments - any text between **double slash //** and end of the line will be ignored,
- multi-line comments - any text between **/*** and ***/** will be ignored.

```
1 /* Note: examples of comments
2 */
3
4 // x: 1; // this line is commented out
5
6
7
8
9
10
11
12
13
14
15
16
17
```

```
1 /* Note: following lines are commented out
2 * - will be not evaluated
3 */
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

Statements

SPC is list of statements that define a configuration. SPC statements are composed of:

- values,
- operators,
- expressions,

- keywords,
- comments.

Semicolon ;

Each SPC statement must end with a semicolon:

```
1 x: 1;
2 y: "a";
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

This allows for several statements in one line: **x: 1; y: "a";**

Semicolon can be omitted after closing curly bracket **{...}**.

```
1 x: {
2   y:1;
3 } // <--- Semicolon can be omitted
```

Line Length and Line Breaks

For readability, it is better to avoid long lines - longer than 80 characters. If a statement does not fit on a line, it is best to split it after/before an operator:

```
1 x: "a" << "b" <<
2   "c" << "d";
```

Syntax

SCP syntax comprise a set of rules how to construct a configuration script:

```
1 // Declare variables and assign values
2 x: 1; y: 2;
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

Values

Syntax of SCP defines two types of values:

- literals - constant values,
- variables - variable values.

Literals

Syntax rules for constant values:

- Numbers - may contain decimals and may be written in scientific notation:
1 1002 // A number without decimals
2 10.02 // A number with decimals
3 12.0e3
4 12.0e-3 // 0.012
- Strings - a text written within double quotes:
1 "example text"

- Booleans - can have only two values: **true** or **false**.

Variables

Variables are used to store data values. They must be identified with unique names (identifiers). The type of variable is derived from its initial value. A colon sign is used to assign an initial value.

```
1 /* Declare variables
2 *//
3 x: 1; // of type number
4
5 b: true; // of type boolean
6
7 s: "text"; // of type string
```

Objects

Objects are variables that associate many variables which are surrounded by curly brackets **{...}**. Any variable declared inside an object becomes its member. An object, groups statements depending on their functionality and its definition can span multiple lines. Each object creates its own scope with separated set of variables. Variables can share the same identifier if they are not members of the same scope (object).

```
1 /* Declare the variable "obj" of type object
2 * that associates several variables
3 * of different types
4
5 obj: {
6   xx: 20;
7   yy: "2";
8   zz: "text";
9
10  * A child object with its own set of
11  * variables
12  childObj: {
13    // The Identifier "xx" is shared by
14    * two variables, but each of
15    * the variables belong to different
16    * object(scope)
17    xx: "I belong here";
18    ...
19  }
20
21
22
23
24
```

Accessing Object Members

Inside an object its members can be accessed directly by identifiers. An object's members can be accessed also by using dot **.** operator. Just like other variables, object members can be used to construct an expression:

```
1 obj: {
2   x: 1;
3   y: 2;
4   a: x; // Access to the local member
5 }
6
7 * Access to the object member outside of
8 * its local scope
9
10 obj.x + obj.y;
11
12 c: obj.x;
```

Array of Values

An array is a special type of variable, that can hold more than one value of the same type at a time. Values are separated by comma **[,]** surrounded by square brackets **(array operator) [...]**:

```
1 [1, 2, 3] // Array of numbers
2 ["a", "b", "c"] // Array of strings
3
4 * Array of objects
5
6 obj1: {x: 1}; obj2: {x: 2};
```

Declaring an Array

An array is declared by use of the array operator and can span multiple lines:

```
1 /* An array declaration can span multiple
2 * lines
3
4 a: [
5   "text1",
6   "text2",
7   "text3",
8 ];
```

Accessing Array Elements

NOTE:

In current version of SPC, elements of an array CAN NOT be accessed directly, as they are used only to set values of some configuration options.

Range

A range is a special type of variable, that defines array values of type number. It consists of three values (operands) of type number separated by colon **[:]**, where:

- first - defines start of a range,
- second - step size which is used to obtain next array element, what can be described by a simple algorithm:

```
1 value = start
2 while value <= end do
3   add_value_to_array(value)
4   value = value + step
5 end do
```

- third - end of a range.

```
1 start:step:end
```

Declaring an Array Using a Range

```
1 /* Equivalent to array declaration
2 * x: [1:0, 1:5, 2: 2.5, 3];
3
4 */
5 x: [1:0:5:3];
```

Branching

Branching is an operation on a configuration. For each value in an array a new configuration branch is created (if the size of an array is > 1). To distinguish branches from a regular array a list of values are surrounded by double square brackets (**branching operator** `[[...]]`):

```

1 /*
2 * Branching operation will
3 * create 2 configuration branches
4 * where x will have values defined by range
5 * x: [[1:1:2]];
6 */
7
8 /*
9 * After evaluation, two configurations
10 * will be generate:
11 */
12
13 // Configuration branch 1:
14 x: 1;
15
16 // Configuration branch 2:
17 x: 2;
18
19
20 /*
21 * Branching by an object declaration
22 */
23 x: [[
24   x_branch_1: {
25     a: "branch1";
26   },
27   x_branch_2: {
28     a: "branch2";
29   }
30 ]];
31
32 /*
33 * After evaluation, two configurations
34 * will be generate:
35 */

```

```

36 /* Configuration branch 1:
37 */
38 ... // Script of configuration
39 x: {
40   a: "branch1";
41 }
42 ... // Script of configuration
43
44 /* Configuration branch 2:
45 */
46 ... // Script of configuration
47 x: {
48   a: "branch2";
49 }
50 ... // Script of configuration
51

```

The set of configuration branches can be executed sequentially or in parallel.

Operators

The following operators are allowed to construct expressions:

- arithmetic operators `+ - * /` can be used to

compute values:

```
1 (2 + 3) * 5
```

- comparison operators `== != >= <= > <` can be used to test for `true` or `false`:

```
1 2 == 2 // true
```

- the string concatenation operator `<<` can be used to create a string from many values:

```
1 "text_" << 1 << "_text"
```

- the assignment operator `=` can be used to assign a new value to the variable after it has been declared:

```
1 x: 1; // Variable declaration
2 x = 2; // Assign a new value
```

NOTE: The assignment operator is only evaluated in the case of step by step evaluation (see: **Stepwise Simulation**)

- boolean operators can be used to create boolean expressions that can be used in a conditional block:

```

&& - AND - denoted x&&y;
|| - OR - denoted x||y;
! - NOT - denoted !x;

```

the denoted expression values can be expressed by the truth table:

x	y	x&&y	x y	!x
false	false	false	false	true
true	false	false	true	false
false	true	false	true	true
true	true	true	true	false

Conditional Block

NOTE: The conditional block is only evaluated in the case of step by step evaluation (see: **Stepwise Simulation**)

A conditional block consists of conditional statements that allow to perform different actions based on different conditions:

- if - execute a block of code if a given condition is true;

```

1 if(condition) {
2   /* Block of code to be executed
3   * if the condition is true
4   */
5   ...
6 }

```

- else if - test a new condition if the previous is false and execute an alternative block of code if the current condition is true;

```

1 if(condition1) {
2   /* Block of code to be executed
3   * if the condition1 is true
4   */
5   ...
6 } else if(condition2) {
7   /* Block of code to be executed
8   * if the condition1 is false
9   * and condition2 is true
10  */
11  ...
12 }
13
14

```

- else - execute block of code if all previous conditions are false;

```

1 if(condition1) {
2   /* Block of code to be executed
3   * if the condition1 is true
4   */
5   ...
6 } else {
7   /* Block of code to be executed
8   * if all previous conditions
9   * are false
10  */
11  ...
12 }
13
14

```

Expressions

An expression computes a value. It is a combination of variables, values and operators. The expression computation is called an evaluation:

```

1 2 == 2 // Evaluate to true
2 (2 + 3) * 5 // Evaluate to 25

```

An expression may contain variables: `x + 1` and depending on the operator used, the values can be of different types, such as numbers and strings:

```

1 "text_" << 1 // Evaluate to text_1

```

An expression can be used to assign an initial value to a variable. The evaluation results determine type of variable.

```

1 /*
2 * Evaluate to 3 which represents
3 * Integer number type.
4 */
5 b: 1 + 2;
6
7 /*
8 * Evaluate to 0.5 which represents floating
9 * point number type.
10 */
11 x: 1/2;
12
13 /*
14 * Evaluate text_1 which represents
15 * string type.
16 */
17 s: "text_" << 1;

```

A boolean expression can be used as a condition in a conditional statement:

```

1 x: 1;
2 y: 2;
3
4 if((y - 1) >= x && x != 0) {
5   /* Block of code to be executed
6   * if the condition is true
7   */
8   ...
9 }
10

```

Identifiers

Identifiers are used to name variables. An identifier is a sequence of characters that starts with a letter or an underscore `_`. The following characters can be letters, digits and underscores:

```

1 // example identifiers
2 var // Valid
3 var_ // Valid
4 _var1 // Valid
5 var__a1 // Valid
6
7 2var // Invalid !!

```

Case Sensitive

All identifiers are case sensitive.

```

1 /*
2 * The variables: var and Var,
3 * are two different variables.
4 */
5 var: 1;
6 Var: "testk";
7

```

Data Types

SPC has dynamic types - what means that the same variable can be used to hold different data types:

```

1 // x is initialised as a number type
2 x: 1;
3 x = "tekse"; // Now x is of type string

```

Primitive Data

A primitive data is a value without additional properties:

- number,
- string,
- boolean.

Composite Data

A composite data is data type that can comprise primitive data:

- array,
- range,
- object.

Data Type Checking

Checking of values data types is done during evaluation of expressions:

- arithmetic expressions involve arithmetic operators that operate only on the number data type;
- `"a" + 2.5` // `Valid`
- `"a" + 2` // `Invalid`
- `true + 3` // `Invalid`
- string concatenation operator operate on the primitive data and implicitly convert each operand to the string type;
- `"a" << 1` // `Valid`
- boolean expressions involve boolean operators that operate only on the boolean data type;

```
1 true || false // Valid
2
3 0 || false // Invalid
4
5 "a" || true // Invalid
```

Configuration

Structure

NOTE: Not obligatory options are marked as [OPTIONAL].
NOTE: Unless explicitly stated, the order of configuration options is not important.
SPC consists of three main blocks/objects:

```
1 /* What to import
2 */
3 import: {...}
4
5
6 /* Configuration of model and simulation
7 */
8 configuration: {...}
9
10
11 /* Logging user-defined variables
12 */
13 log: {...} // [OPTIONAL]
```

Logging

Additional logging data can be defined in the `log` block where values of declared variables can be traced:

```
1 /**
2 * Extra log
3 */
4 log: {
5 simulation: configuration.simulation.name;
6 valueset: configuration.model.constants.valueset;
7
8 all_in_one_line: "simulation: "
9 <<< configuration.simulation.name
10 <<< " valueset: "
11 <<< configuration.model.constants.valueset;
12
13 }
```

Import

The import block allows specifying source of a model:

```
1 /* What to import
2 */
3 import: {
4 from: "path/to/model";
5 }
6
```

Supported model formats:

- ANDL,
- CANDL,
- SBML,
- ERODE.

If a SBML model is imported, additional configuration is required:

```
1 /* What to import
2 */
3 import: {
4 from: "path/to/model";
5
6 /* Additional configuration
7 * if the imported model is SBML
8 */
9 sbml: {
10
11
12
13
14
15
16
17
18
19 }
```

```
/* Import model as CPN (continuous PN)
* or SPN (stochastic PN)
*/
net: "CPN";
boundary: true;
reversible: false;
```

Configuration

The configuration block allows setting up configurations of model and simulation:

```
1 configuration: {
2 model: {...} // [OPTIONAL]
3 simulation: {...}
4 }
```

The basic configuration is not evaluated step by step but on demand, that means when Spike lookups for given configuration parameter, its value is evaluated.

Model

Configuration of a model allows specifying its parameters (constants and initial markings) and defining observers:

```
1 model: {
2 constants: {...} // [OPTIONAL]
3 places: {...} // [OPTIONAL]
4 observers: {...} // [OPTIONAL]
5 }
```

Configuration of model constants allows overriding their values. In case the constant does not exist in the model definition, it will be created and added to it with the associated group.

Simulation

Basic configuration of a simulation allows specifying:

- a simulation type and its solver,
- a simulation time given in the form of an interval or a range,
- export of simulation results.

```
1 simulation: {
2 name: "example"; // Name of a simulation
3 type: nameOfType; {
4 solver: nameOfSolver; {
5 // Solver options
6 ...
7 }
8 }
9
10 /* Interval
11 * start: splitting: end
12 interval: 0:100:100;
13 //range: start:step:end;
14
15 /* Define the stepwise simulation
16 */
17 onStep: { // [OPTIONAL]
18 ...
19 }
20
21 /* Export results to a file
22 */
23 export: { // [OPTIONAL]
24 ...
25 }
```

Simulation Time

Simulation time can be in the form of an interval or a range. The interval and the range are objects with properties that can be accessed:

```
1 // Start time of a simulation recording
2 interval.start
3 // How many samples to record
4 interval.splitting
5 // End time of a simulation recording
6 interval.end
7
8 /* Mapping interval to range:
9 * start = interval.start;
10 * end = interval.end;
11 * step = (interval.end - interval.start)
12 // interval.splitting;
13 */
14
15 // Start time of a simulation recording
16 range.start
17 // Step size up to the next point in time
18 range.step
19 // End time of a simulation recording
20 range.end
```

Note: If the observer exists, in the model definition, its definition will be overridden.

Simulation Types

Supported simulation types:

- continuous,
- stochastic,
- hybrid.

NOTE:
Unless explicitly stated, the simulation options have no default values and all are mandatory.

```

1 type:continuous {
2   solver.nameOfSolver: {
3     ...
4   }
5 }
6
7 type:stochastic {
8   solver.nameOfSolver: {
9     ...
10    }
11  }
12  * Define whether includes to an export
13  * results of a single run and averaged
14  * NOTE: this option is only valid for
15  * single thread simulation
16  */
17  single: false; // Default
18  avg: true; // Default
19 }
20
21 type:hybrid {
22   solver.nameOfSolver: {
23     ...
24   }
25 }

```

Continuous Simulation Solvers

```

1 BDF: {
2   semantic: "adapt"; // "bio", "adapt"
3   iniStep: 0.1;
4   /*
5    * "CVDense", "CVSpgrm", "CVDiag",
6    * "CVSpbcg", "CVSptfqr",
7    */
8   linSolver: "CVDense";
9   relTol: 1e-5;
10  autoStepSize: false;
11  reduceResultingODE: true;
12  checkNegativeVal: false;
13  outputNoiseVal: false;
14 }
15
16 ADAMS: {
17   semantic: "adapt"; // "bio", "adapt"
18   iniStep: 0.1;
19   /*
20    * "CVDense", "CVSpgrm", "CVDiag",
21    * "CVSpbcg", "CVSptfqr",
22    */
23   linSolver: "CVDense";
24   relTol: 1e-5;
25   autoStepSize: false;
26   reduceResultingODE: true;
27   checkNegativeVal: false;
28   outputNoiseVal: false;
29 }

```

Stochastic Simulation Solvers

```

1 direct: {
2   // all
3   threads: 1;
4   runs: 10;
5   /*
6    * If not present then random seed is set
7    */
8   //seed: 2413805201; // Default random
9 }
10
11 tauLeaping: {
12   // all
13   threads: 1;
14   runs: 100;
15   /*
16    * If not present then random seed is set
17    */
18   //seed: 2413805201; // Default random
19 }

```

```

1 deltaLeaping: {
2   // all
3   threads: 1;
4   runs: 100;
5   delta: 1.0;
6   /*
7    * If not present then random seed is set
8    */
9   //seed: 2413805201; // Default random
10 }

```

```

1 fau: {
2   threads: 1;
3   runs: 1;
4   maxLambda: 0;
5   tolerance: 1.0e-7;
6   delta: 1.0e-11;
7   epsilon: 1.0e-7;
8   /*
9    * If not present then random seed is set
10   */
11   //seed: 2413805201; // Default random
12 }

```

Hybrid Simulation Solvers

```

1 static: {
2   threads: 1;
3   runs: 1;
4   odeSolver: "ADAMS"; // "ARK", "BDF", "ADAMS"
5   iniStep: 0.1;
6   /*
7    * "CVDense", "CVSpgrm", "CVDiag",
8    * "CVSpbcg", "CVSptfqr",
9    */
10  linSolver: "CVDense";
11  relTol: 1e-5;
12  autoStepSize: true;
13  reduceResultingODE: true;
14  checkNegativeVal: false;
15  outputNoiseVal: false;
16 }

```

```

1 staticAcc: {
2   threads: 1;
3   runs: 1;
4   odeSolver: "ADAMS"; // "ARK", "BDF", "ADAMS"
5   iniStep: 0.1;
6   /*
7    * "CVDense", "CVSpgrm", "CVDiag",
8    * "CVSpbcg", "CVSptfqr",
9    */
10  linSolver: "CVDense";
11  relTol: 1e-5;
12  autoStepSize: true;
13  reduceResultingODE: true;
14  checkNegativeVal: false;
15  outputNoiseVal: false;
16 }

```

```

1 HRSSA: {
2   threads: 1;
3   runs: 1;
4   odeSolver: "ADAMS"; // "ARK", "BDF", "ADAMS"
5   iniStep: 0.1;
6   /*
7    * "CVDense", "CVSpgrm", "CVDiag",
8    * "CVSpbcg", "CVSptfqr",
9    */
10  linSolver: "CVDense";
11  relTol: 1e-5;
12  abstol: 1.0e-10;
13  autoStepSize: true;
14  reduceResultingODE: true;
15  checkNegativeVal: false;
16  outputNoiseVal: false;
17  /*
18   * fluctatio: 0.2; // Fluct ratio
19   * applyMonPlaces: true;
20   */
21 }

```

```

1 HRSSAacc: {
2   threads: 1;
3   runs: 1;
4   odeSolver: "ADAMS"; // "ARK", "BDF", "ADAMS"
5   iniStep: 0.1;
6   /*
7    * "CVDense", "CVSpgrm", "CVDiag",
8    * "CVSpbcg", "CVSptfqr",
9    */
10  linSolver: "CVDense";
11  relTol: 1e-5;
12  autoStepSize: true;
13  reduceResultingODE: true;
14  checkNegativeVal: false;
15  outputNoiseVal: false;
16  /*
17   * setRatio: 0.2; // Fluct ratio
18   * applyMonPlaces: true;
19   * applyInterfacePlaces: true;
20   */
21 }

```

Stepwise Simulation

NOTE:

The order of configuration options is **important**. The `onStep` block is evaluated step by step (instruction after instruction) and consists of two parts:

- declaration - allows for declaring and initializing of variables, and it is evaluated once at the beginning of a simulation; a declared variable can be observed, (added to a set of observers) what allows to include them in a simulation result;

- 1 `varName:observe: 1;`
- `do -` is evaluated after each step of a simulation and should consist main logging that adjust a model after each simulation step.

The read access is also possible to a current time and step of a simulation:

```
1 export: {
2   places: [];
3   transitions: [];
4   observers: []};
5
6 csv: {
7   sep: ";", // Data separator
8   file: "path/to/file.csv";
9 }
10
11 }
```

Supported Regular Expressions

```
1 x - Match any character x;
2 . - Match any character;
3 [] - Match a range of characters;
4 [xyz] - Matches x, y or z;
5 [ab]-oz] - Match character range in it; matches a, b any letter from j through o or Z;
6 [~] - Match any character except those in square brackets;
7 [-A-Z] - In this case, any character except an uppercase letter;
8 r* - Zero or more r's, where r is any regular expression;
9 r+ - One or more r's;
10 r? - Zero or one r's;
11 r{n} - Exactly n r's;
```

```
1 /* Export specified place that pass
2 * to a given expression.
3 * NOTE: If the array is empty ALL places
4 * will be exported
5 */
6 places: ["PlaceName01", "Pla.*Name0[1-3]"];
7
8 /* Export only uncoloured places
9 */
10 places:u: [];
11
12 /* Export only coloured places
13 */
14 places:c: [];
15
16 /* Export all transitions
17 */
18 transitions: [];
19
20 /* Export only coloured transitions
21 */
22 transitions:c: [];
23
24 /* Export only uncoloured transitions
25 */
26 transitions:u: [];
27
28 /* Export all observers
29 */
30 observers: [];
```

One configuration file may comprise multiple exports of simulation results:

```
1 export: {
2   ...
3 }
4
5 export: {
6   ...
7 }
8
9 ...
```

By use of arrays of strings that represent regular expressions (see: **Supported Regular Expressions**) over the nodes of which the simulation traces are to be recorded. If a given array is empty then ALL nodes from a given set will be exported:

```
1 /* Export specified place that pass
2 * to a given expression.
3 * NOTE: If the array is empty ALL places
4 * will be exported
5 */
6 places: ["PlaceName01", "Pla.*Name0[1-3]"];
7
8 /* Export only uncoloured places
9 */
10 places:u: [];
11
12 /* Export only coloured places
13 */
14 places:c: [];
15
16 /* Export all transitions
17 */
18 transitions: [];
19
20 /* Export only coloured transitions
21 */
22 transitions:c: [];
23
24 /* Export only uncoloured transitions
25 */
26 transitions:u: [];
27
28 /* Export all observers
29 */
30 observers: [];
```

In the scope of `onStep` block is possible the read/write access to places and constants of a model:

```
1 place_nameOfPlace
2 constant_nameOfConstant
```

```
1 dynamic: {
2   threads: 1;
3   runs: 1;
4   onSolver: "ADAMS"; // "ARK", "BDJ", "ADAMS"
5   onStep: 0;
6   * "CVdense", "CVSparse", "CV0lag",
7   * "CVSpbcg", "CVSptEqm";
8 }
9
10 LinSolver: "CVdense";
11 rellol: 1e-5;
12 ans: 1e-6;
13 onSteps: 10;
14 reduceResultingODE: false;
15 checkNegativeVal: false;
16 outputNoiseVal: false;
17 //
18 fluctRatio: 0.2; // Fluct ratio
19 // Apply interface places
20 // Apply interface faces: true;
21 rateThr: 10.0; // Rate threshold
22 markingThr: 3.0; // Marking threshold
23 // Max partitioning messages
24 maxPartMsg: 100;
25 // Show partitioning information
26 showPartInfo: true;
27
28 }
```

`onStep: {`
 * Declaration part - evaluates only once at the beginning of a simulation
 *
 a: 0;
 * A variable can be added to observers, * what allows recording how variable * change over a simulation time
 b: observe: 0;
 do: {
 * Main loop part - evaluates after * each simulation step
 *
 * Log value of the:
 * - simulation time and step,
 * - constant C.
 LOG = "time: " << simulation.time;
 LOG = "step: " << simulation.step;
 LOG = "value_C: " << constant.C;
 if(a < 10 && place.p > 5) {
 constant.C = constant.C - 1;
 } else {
 constant.C = constant.C + 1;
 }
 a = a + 1;
 // Log value of the variable a;
 LOG = "value_a: " << a;
 * Increment observable var
 *
 b = b + 1;
 }
}

In the scope of `onStep` block is possible the read/write access to places and constants of a model:

```
1 place_nameOfPlace
2 constant_nameOfConstant
```

It is possible to combine the results of places, transitions and observers, coloured and uncoloured, in one

Full Examples

The model. As a working example a simple compartmental model SIR is used to model the epidemic process. The model consists of three compartments (SusceptiblePopulation_A - susceptible population, Infected_A - infected population and Recovered_A - recovered population) and two events (Infect and Recover) which are represented by places and transitions, respectively. The model is represented on Figure 1 and as ANDL source code.



Figure 1: SIR model as SPN.

```

1 spn [SIR-SPN]
2 constants:
3 all:
4 double k_infect_a = 5.0e-5;
5 double k_recover_a = 1.0e-1;
6
7 places:
8 Infect:
9 : [Infected_A + 2] & [SusceptiblePopulation_A - 1] &
10 : [Infected_A - 1]
11 Recover:
12 : [Recovered_A + 1] & [Infected_A - 1]
13
14 stochastic:
15 Infect:
16 :
17 :
18 :
19 :
20 :
21 :
22 :
23 :
24 :
25 :
26 :
27 }

```

Example 1: Simple simulation configuration

```

1 /**
2  * Example configuration
3  */
4 // - Line comment
5 // - Block comment
6
7 /** Import - exactly one model
8  * From: "./model/SIR-SPN.andl";
9  */
10 configuration: {
11   SusceptiblePopulation_A: 20000;
12 }
13
14 // Name of a simulation
15 name: "SIR";
16
17 /* Set up a simulation
18 */
19 type: continuous: {
20   solver: BDF;
21   semantic: "adapt"; // "bio", "adapt"
22   inStep: 0.1; // "CVSpmr", "CVDIag", "CVSpbg", "CVSptqmr"
23   inSolver: "CVDense";
24   relTol: 1e-5;
25   autoStepSize: false;
26   checkResultingODE: true;
27   outputNoiseVal: false;
28 }
29 interval: 0:200:100;
30
31 export: {
32   // Array of places to save (if empty export all)
33   places: []; // all places
34   // places: []; // all coloured places
35   // transitions: []; // all transitions
36   // transitions:c: []; // all coloured transitions
37   // transitions:u: []; // all uncoloured transitions
38   observers: [];
39   csv: { // Separator
40     sep: ";";
41     file: "/data/";
42     // << name << " "
43     // << import name << " "
44     // << configuration.simulation.type << " "
45     // << configuration.simulation.type.solver
46     // << _FILESE-SEP.csv; // File name
47   }
48 }

```

Results

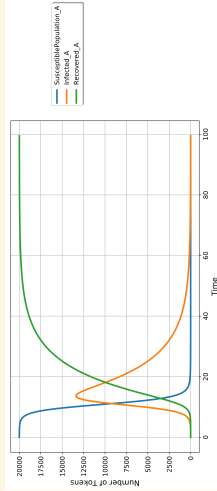


Figure 2: Simulation results.

Example 2: Configuration branching

How to set up configuration branching for scanning over:

- model parameters,
- configuration options; in this case, type of simulation.

```
1 /**
2  * Example configuration of branching
3  */
4 // - line comment
5 // - block comment
6
7 // Import - exactly one model
8 import: {
9   from: {
10     ".model/SIR-SPW.andl";
11   }
12 }
13
14
15 configuration: {
16
17   model: {
18     constants: {
19       a: 1;
20       b: 1;
21     };
22     // Branching: Scanning over model parameters
23     k_infect_a: [[5.0e-5, 8.0e-5]];
24
25   };
26   places: {
27     // Branching: Scanning over model parameters
28     SusceptiblePopulation_A: [[20000, 50000]];
29   };
30   simulation:
31   {
32
33     /** This is example variable that is added
34     * to the log
35     */
36     varExa: model.places.SusceptiblePopulation_A;
37     // Name of a simulation
38     name: "SIR";
39
40     // Branching:
41     // Scanning over simulation types
42     type: {
43       // Stochastic simulation
44       stochastic: {
45         direct: {
46           threads: 1;
47           runs: 3;
48           //seed: 2413805201;
49         };
50         //single: true; // Single
51         //avg: false; // Default set of true
52       };
53       // Continuous Simulation
54       continuous: {
55         solver: {
56           // Define new "runs" variable that
57           // is used in the export file name
58           runs: "COM1";
59           semantic: "adapt"; // "bio", "adapt"
60           initStep: 0; // "CISppmr", "CVDlag", "CVSpbcg", "CVSpfpmr"
61           //Solver: "CVDense";
62           relTol: 1e-5;
63           absTol: 1.0e-10;
64           autoStepSize: false;
65           reductResultingODE: true;
66           checkNegativeVal: false;
67           outputNoiseVal: false;
68         };
69       };
70     };
71
72   };
73
74 };
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

Example 3: Stepwise simulation

Depending on the current number of infected specimens set restriction or relaxation rules by applying a change of infection kinetic rates in a given 57 time frame.

```

1  /** Example configuration of a stepwise simulation
2  */
3  // - Line comment
4  // - block comment
5  // Import - exactly one model
6  import { from: ".,model/SIR-SPW.and1";
7  }
8  configuration: {
9  model: {
10 constants: {
11   k_infect_a: 5.0e-5;
12 }
13 places: {
14   SusceptiblePopulation_A: 20000;
15 }
16 simulation:
17 {
18   /* This is example variable that is added
19   * to the log
20   */
21   varExample: model.places.SusceptiblePopulation_A;
22   name: "SIR";
23   /*
24   * Set up a simulation
25   */
26   type:stochastic; {
27     solver: {
28       threads: 1;
29       runs: 3;
30     };
31     //seed: 2413805201;
32     single: true; // Default set of true
33     //avg: false; // Logging extra informations
34     interval: 0:200:100;
35   }
36 }
37 }
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }
46 }
47 }
48 }
49 }
50 }
51 }
52 }
53 }

```

```

54  /* Stepwise simulation
55  * Description: Depending on the current number of
56  * infected specimens, set restriction or relaxation
57  * rules by applying a change of infection kinetic rates
58  * in a given time frame.
59  * In a given time frame.
60  */
61  onStep: enabled: {
62  * Kinetic parameters of
63  * the restriction and relaxation
64  */
65  rules:
66  {
67  k_infect_lo: 1.0e-9;
68  k_infect_hi: 5.0e-5;
69  }
70  k_infect:observe: constant.k_infect_a;
71  InitialSusceptiblePopulation_A: place.SusceptiblePopulation_A;
72  TimeFrame: 2; // Stretch factor
73  /* Calculate the window size that stretch over
74  * the full time frames defined by iTimeFrame
75  */
76  iWin: (interval.splitting / (interval.end - interval.start)) * iTimeFrame;
77  WinStep: 0;
78  bFirst: true;
79  bRelax: false;
80  dWinStepSize: 0;
81  LOG = "END_INIT";
82  /* Smoothed stepwise lockdown and relaxation
83  */
84  do: {
85  LOG = "step:" <<< simulation.step;
86  LOG = "Time:" <<< simulation.time;
87  /* Change infection rate if the number of
88  * infected specimens is > than 40% of susceptible
89  * population
90  */
91  if (place.Infected_A > iInitialSusceptiblePopulation_A * 0.4
92  && !bFirst) {
93  bFirst = true;
94  WinStep = 0; // step size
95  dWinStepSize = (constant.k_infect_a - k_infect_lo) / iWin;
96  bRelax = false;
97  }
98  /* Change infection rate if the number of
99  * infected specimens is < than 20% of susceptible
100 * population
101 */
102 else if (place.Infected_A < iInitialSusceptiblePopulation_A *
103 0.2 && !bRelax && bFirst) {
104 WinStep = 0;
105 dWinStepSize = step_size;
106 dWinStepSize = (constant.k_infect_a - k_infect_hi) / iWin;
107 bRelax = true;
108 }
109 // ABS - absolute value
110 if (dWinStepSize < 0) {
111 dWinStepSize = -dWinStepSize;
112 }
113 /* Adjust the kinetic parameter according
114 * to the position in the time frame
115 */
116 if (iWinStep < iWin) {
117   if (!bRelax) {
118     constant.k_infect_a = constant.k_infect_a -
119     dWinStepSize;
120   } else if (bRelax) {
121     constant.k_infect_a = constant.k_infect_a +
122     dWinStepSize;
123   }
124 WinStep = iWinStep + 1;
125 }
126 // Set the value of the observed variable
127 k_infect = constant.k_infect_a;
128 // Logging extra informations
129 LOG = "bRelax: " <<< bRelax;
130 }
131 }
132 }
133 }
134 }
135 }
136 }
137 }
138 }
139 }
140 }
141 }

```

```

142 export: {
143   // Array of places to save (if empty export all)
144   places: [ ]; // all places
145   //places:c: [ ]; // all coloured places
146   //places:u: [ ]; // uncoloured places
147   //transitions:c: [ ]; // all coloured transitions
148   //transitions:u: [ ]; // all uncoloured transitions
149   observers: [ ];
150   sep: " "; // Separator
151   file: "data";
152   <<< import name <<< " "
153   <<< configuration.simulation.type <<< " "
154   <<< configuration.simulation.type.solver <<< " "
155   <<< " " <<< configuration.simulation.type.constants.all.k_infect_a
156   <<< " " <<< configuration.model.constants.all.k_infect_a
157   <<< configuration.model.places.SusceptiblePopulation_A
158   <<< " _FILESE-step.csv"; // File name
159 }
160 }
161 }
162 }
163 }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }

```

Results

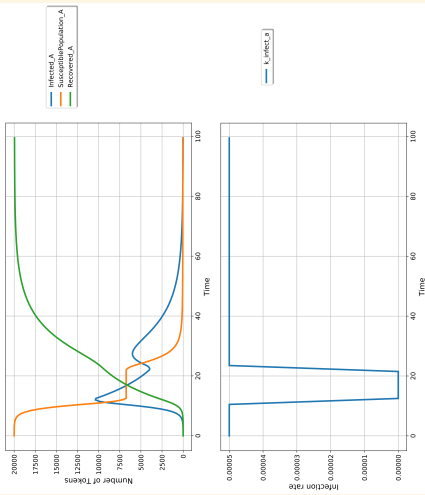


Figure 3: The simulation results.