

UNIVERSITY OF CAMERINO
PHD COURSE IN COMPUTER SCIENCE AND MATHEMATICS

SIBILLA:
A COMPREHENSIVE FRAMEWORK FOR STOCHASTIC SYSTEMS ANALYSIS AND PARAMETER SYNTHESIS

DOCTORAL THESIS

Author
Lorenzo Matteucci

Tutor(s)

Prof. **Michele Loreti**

Reviewer(s)

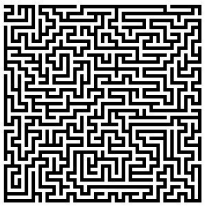
Prof. **Ezio Bartocci**
Prof. **Andrea Marin**

Camerino, May 2025

XXXVI

This thesis is dedicated to my journey,
the names I carry,
the ones who stayed,
and all the experiences that have brought me here.

*"Ringraziare voglio il divino **labirinto** degli effetti e delle cause*
*per la **ragione**, che non cesserà di sognare un qualche disegno del labirinto,*
*per la diversità delle creature che compongono questo singolare **universo**,*
*per l'**amore**, che ci fa vedere gli altri come li vede la divinità,*
*per la notte, le sue tenebre e la sua astronomia, per il **coraggio** e la felicità degli altri,*
*per gli intimi **doni** che non elenco,*
*per la musica, misteriosa forma del **tempo**"*



J.L.Borges

1964

estratto da "**Otro poema de los dones**", in '**L'altro, lo stesso**'

Acknowledgements

THIS research was supported by the Computer Science department at the University of Camerino. I would like to express my deepest gratitude to Professor Michele Loreti, not only for his supervision and guidance throughout this project but also for offering me this unique opportunity to pursue my research. I am particularly thankful for the trust he placed in me by providing this chance, which I recognize is a privilege not granted to everyone. I would like to extend my appreciation to the School of Advanced Studies (SAS) of the University of Camerino for their constant administrative support, particularly in helping me navigate the complexities of academic bureaucracy.

I am grateful to my colleagues in the department for their valuable feedback and discussions that helped shape this research. The facilities and resources provided by the University of Camerino were instrumental in completing this work. I would also like to acknowledge the contributions of the open-source community whose tools and libraries were essential for this research.

Lorenzo Matteucci
Camerino, 2024

Ringraziamenti

In primis, desidero ringraziare mia madre, **Paola Pelagagge**: È grazie al suo esempio e ai suoi insegnamenti che sono diventato la persona che sono. Un ringraziamento speciale va anche alla mia famiglia materna e il supporto che mi è stato dato.

Ringrazio il mio relatore, professor **Michele Loreti**, per avermi dato l'opportunità di affrontare un percorso formativo e personale tanto impegnativo quanto arricchente.

Un posto nei ringraziamenti lo meritano anche i colleghi di dottorato, con cui ho condiviso lavoro, discussioni e esperienze: a **Mattia Paccamiccio**, collega senza peli sulla lingua, schietto, diretto sempre pronto all'aiuto, a **Nicola Del Giudice**, collega con un percorso molto vicino al mio, con cui ho lavorato fianco a fianco fin dall'inizio, ad **Alessandro Marcelletti**, integerrimo contacalorie e instancabile esploratore di posti dove mangiare — spesso sperduti, costosi o entrambi ma quasi sempre ottimi. Un pensiero va anche a **Sara Pettinari**, che dà sempre delle buone dritte, ma che è senz'altro meglio non far arrabbiare. A **Morena Barboni**, complice di malefatte e coautrice di improbabili disegni comparsi qua e là sulle lavagne del polo, a **Matteo Belenchia**, collega conosciuto più avanti nel percorso, con cui ho condiviso numerose conversazioni interessanti — spesso a scapito della pazienza dei colleghi d'ufficio. Un ringraziamento speciale a **Daniele Marchei**, amico e compagno di viaggio sin dal primo giorno di triennale. Abbiamo condiviso ogni tappa di questo percorso, ed è anche grazie a lui se sono arrivato fin qui. Similmente, un ringraziamento va a **Lorenzo Dezi**, che ha proseguito il suo percorso universitario altrove per poi darsi al (vil) lavoro in azienda — presto probabilmente ci uniremo a lui.

Un ringraziamento speciale va anche alle persone che hanno fatto parte della mia vita personale durante questo periodo: **Silvia Spaventa**, che per tutto il periodo del dottorato ha assistito in prima linea agli alti e bassi del percorso, **Arianna Marongiu** ed **Elvira Muminovic**, amiche di una vita, che con la loro presenza costante e il loro affetto mi hanno fatto sentire come se non fossi mai stato davvero figlio unico, a **Lisa Guagnini**, la cui presenza affettuosa, incondizionata e costante ha saputo sempre portare serenità, a **Desy Michelini**, che mi ha spinto a nuovi viaggi e esperienze, ad

Ilaria Torello, per il suo costante supporto e il suo spronarmi a fare pace con me stesso e ad **Anna Maria Panza**, che mi ha supportato concretamente nel conciliare l'attività didattica nelle scuole con gli impegni del dottorato.

Infine un ringraziamento sentito va anche a tutti gli studenti con cui ho avuto modo di interagire in questi anni. Il confronto con loro è stato una delle parti più gratificanti del mio percorso, sia dal punto di vista umano che professionale: una menzione particolare va a **Filippo Biondi**, studente diligente e preciso, che ha contribuito al mio lavoro attraverso la sua tesi triennale e a **Marta Musso**, studentessa capace e determinata.

Un pensiero particolare va a mio nonno, **Giovanni Pelagagge**; se oggi la mia famiglia vive un certo benessere, è anche grazie a quello che ha costruito con fatica e intelligenza. Ma l'eredità più grande non è quella materiale: è qualcosa che mi porto dentro e che ogni tanto riaffiora, come quando mi raccontava con calma e precisione come funzionava un trattore. Più passa il tempo, più mi accorgo che, in modi diversi, gli somiglio. Un po' come *F. Battiato* in *Mesopotamia* che, in figure e civiltà del passato, riconosceva qualcosa di sé — non un'origine concreta, ma un'affinità più profonda, che attraversa il tempo, anche io, sento un legame, una risonanza, una storia che, in qualche modo, voglio continuare.

Sicuramente ho dimenticato qualcuno: me ne scuso in anticipo. Ma se siete stati parte di questo percorso, anche solo per un tratto, sappiate che vi sono grato. Anche se non vi ho nominati, il vostro contributo c'è stato.

Lorenzo Matteucci
Cingoli, 2025

Summary

MODERN technological systems are becoming increasingly complex, with numerous components that interact and adapt their behaviour based on their environment and the actions of other elements. These systems can be found in various domains, from smart cities and autonomous vehicle networks to biological and social networks. The challenge of understanding, analysing, and optimising such systems is significant, as their behaviour emerges from countless local interactions that can be difficult to predict and control.

This dissertation addresses these challenges by developing and enhancing an analysis framework that combines statistical methods, surrogate-based optimisation, and formal verification techniques. The research focuses on three fundamental aspects: finding optimal system parameters, verifying system properties, and comparing different system variants.

The proposed approach introduces several methods for system analysis, particularly in the realm of parameter optimisation. Integrating machine learning techniques with statistical verification methods provides efficient ways to explore large parameter spaces and understand their impact on system behaviour.

The research demonstrates the framework's broad applicability through several case studies. These include the analysis of epidemiological models, where the framework helps identify optimal intervention strategies; the study of synthetic biological circuits, where it aids in understanding complex cellular behaviours; and the investigation of consensus protocols, where it helps evaluate different approaches to achieving agreement in distributed systems.

A significant contribution of this work is the development of methods for systematically comparing different system versions to observe how modifications affect its behaviour, supporting the exploration of alternative designs.

This work represents a step forward in understanding and improving complex adaptive systems. While significant challenges remain, the foundations laid here provide a robust platform for future developments in both theoretical understanding and practical applications. The continuing evolution of complex systems in critical applications underscores the importance of further research in this direction.

Keywords: Collective Adaptive Systems, Parameter Synthesis, Statistical Model Checking, Surrogate Optimization, Formal Verification, Stochastic Systems, Monte Carlo Methods

Sommario

I sistemi tecnologici moderni stanno diventando sempre più complessi, con numerosi componenti che interagiscono e adattano il loro comportamento in base all'ambiente e alle azioni di altri componenti. Questi sistemi si trovano in vari domini, dalle città intelligenti e reti di veicoli autonomi ai sistemi biologici e alle reti sociali. La sfida di comprendere, analizzare e ottimizzare tali sistemi è significativa, poiché il loro comportamento emerge da innumerevoli interazioni locali che possono essere difficili da prevedere e controllare. Questa dissertazione affronta queste sfide attraverso lo sviluppo e il miglioramento di un framework di analisi che combina metodi statistici, ottimizzazione basata su surrogati e tecniche di verifica formale. La ricerca si concentra su tre aspetti fondamentali: trovare parametri ottimali del sistema, verificare le proprietà del sistema e confrontare diverse varianti del sistema. Il framework introduce diversi metodi per l'analisi dei sistemi, in particolare nel campo dell'ottimizzazione dei parametri. Integrando tecniche di apprendimento automatico con metodi di verifica statistica, fornisce modi efficienti per esplorare ampi spazi parametrici e comprendere il loro impatto sul comportamento del sistema. Attraverso diversi casi di studio, la ricerca dimostra l'ampia applicabilità del framework. Questi includono l'analisi di modelli epidemiologici, dove il framework aiuta a identificare strategie di intervento ottimali; lo studio di circuiti biologici sintetici, dove aiuta a comprendere comportamenti cellulari complessi; e l'investigazione di protocolli di consenso, dove aiuta a valutare diversi approcci per raggiungere l'accordo in sistemi distribuiti. Un contributo significativo di questo lavoro è lo sviluppo di metodi per confrontare sistematicamente diverse versioni di un sistema per osservare come le modifiche a un sistema ne influenzano il comportamento, supportando l'esplorazione di design alternativi. L'implementazione enfatizza l'usabilità pratica insieme all'estensibilità. Il framework supporta molteplici approcci alla modellazione dei sistemi e fornisce interfacce flessibili per incorporare nuove tecniche di analisi. Attraverso l'integrazione con linguaggi di programmazione popolari e piattaforme di cloud computing, rende gli strumenti di analisi avanzati accessibili a una più ampia comunità di ricerca. Questo lavoro rappresenta un passo avanti nella nostra capacità di comprendere e migliorare i sistemi adattivi complessi. Mentre rimangono sfide importanti, le fondamenta poste qui forniscono una solida piattaforma per sviluppi futuri sia nella comprensione teorica che nelle applicazioni pratiche. La continua evolu-

zione dei sistemi complessi in applicazioni critiche sottolinea l'importanza di ulteriori ricerche in questa direzione.

Parole chiave: Sistemi Adattativi Collettivi, Sintesi dei Parametri, Verifica Statistica del Modello, Ottimizzazione tramite Surrogati

List of publications

International Journals

1. Matteucci, L., Loreti, M., and Quadrini, M. (2024). Sibilla: A Tool for Reasoning about Collective Systems. *Science of Computer Programming*. (Vol. 235, pp. 103095). Elsevier. https://doi.org/10.1016/j.scico.2024.103095

International Conferences/Workshops with Peer Review

1. Matteucci, L., Loreti, M., and Quadrini, M. (2022) Sibilla: A Tool for Reasoning about Collective Systems *Coordination Models and Languages*. (pp. 92–98). Springer. https://doi.org/10.1007/978-3-031-08143-9_6
2. Matteucci, L., Loreti, M., and Quadrini, M. (2024). Synthesis Tool Suite for Large Stochastic Models. Accepted for presentation at *18th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2024)*. https://valuetools.eai-conferences.org/2024/

List of Abbreviations

A

ANN	Artificial Neural Networks. 63
ANTLR	ANother Tool for Language Recognition. 143, 144, 157
API	Application Programming Interface. 95, 97, 99, 139, 141, 145, 155

B

BDD	Binary Decision Diagram. 47

C

CAS	Collective Adaptive System. 1–4, 56–59, 81–84, 86–90, 92, 94, 100, 101, 112, 118–120, 122, 125, 138, 193, 194, 196, 198
CPS	Cyber-Physical Systems. 57
CS	Coordinate Search. 72, 73
CSV	Comma-Separated Values. 100
CTL	Computation Tree Logic. 30
CTMC	Continuous Time Markov Chain. 18, 21, 45–50, 101, 108, 113, 118, 129

D

DTMC	Discrete Time Markov Chain. 18–21, 45–50, 102, 103, 105

E

EI	Expected Improvement. 79

F

FPT	First Passage Time. 120–122, 228

G

GPS	Generalized Pattern Search. 72, 73
GTB	Gradient Tree Boosting. 69, 179

I

IDE	Interactive Development Environment. 97

L

LCB	Lower Confidence Bound. 79
LIO	Language of Interactive Objects. 95, 102, 112, 115, 116, 135, 213, 215, 216
LPM	Language of Population Models. 95, 101, 109, 112–114, 128, 135, 170, 209, 215, 216
LTL	Linear-time temporal logic. 30
LtMads	Lower Triangular MADS. 74, 76

M

MADS	Mesh Adaptive Direct Search. 72–74, 76, 146, 147, 152, 179
MAE	Maximum Absolute Error. 63
MAM	Very Large Scale Integration. 108
MAS	Multi-Agent Systems. 103, 105, 218
MITL	Metric Interval Temporal Logic. 30
MSE	Mean Squared Error. 63, 68, 189
MTBDD	Multi-Terminal Binary Decision Diagram. 47
MVC	Model-View-Controller. 95, 100

O

OrthoMADS	Orthogonal MADS. 76, 77

P

PLCF	Piecewise Linear and Continuous Function. 37, 39
PSO	Particle Swarm Optimization. 71, 72, 152, 179, 183, 189

R

RBF	Radial Basis Function. 63, 70, 71, 179
RCPF	Right-Closed Constant Piecewise Function. 13, 31, 39
RF	Random Forest. 68, 179
RMSE	Root Mean Square Error. 63, 68, 189

S

SA	Simulated Annealing. 76–78, 179
SDE	Stochastic Differential Equation. 23–25, 107
SHA	Stochastic Hybrid Automata. 17, 18, 23–25, 27, 45, 49, 50, 105, 107, 108
SIR	Susceptible-Infected-Recovered. 171, 178–181, 183, 184, 195, 235
SLAM	Simple Language for Agent Modelling. 95, 105–108, 112, 114, 115, 135, 159, 170, 221
SMC	Statistical Model Checking. 2, 4, 6, 7, 51, 53, 94, 100, 101, 109, 111, 112, 119
SOLID	Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion. 95, 99
SSA	Stochastic Simulation Algorithm. 51, 52, 111
STL	Signal time temporal logic. 30–34, 36, 37, 40, 118, 122–126, 135, 178, 183, 184, 189, 235, 238
SVR	Support Vector Regression. 63

T

TL	Temporal Logic. 30
TSP	Three-State Protocol. 174, 176, 185

V

VLSI	Very Large Scale Integration. 78

Y

YAML	YAML Ain't Markup Language. 148, 155, 197, 230, 235–237, 239
YODA	Yet anOther agent Description lAnguage. 95, 103–105, 112, 116, 135, 218–220

Notation

General Mathematical Notation

$[a, b]$	Closed interval from a to b
$[a, b)$	Half-open interval from a to b
$(a, b]$	Half-open interval from a to b
$\mathcal{F}$	σ-algebra
$\mathcal{B}(\mathbb{R})$	Borel σ-algebra on $\mathbb{R}$
$\mathbf{P}$	Probability measure
$\mathbb{E}[\cdot]$	Expected value operator
$\mathbb{1}_{\{\cdot\}}$	Indicator function
$\sup$	Supremum (least upper bound)
$\inf$	Infimum (greatest lower bound)
$\min$	Minimum function
$\max$	Maximum function
$\arg\min$	Argument that minimizes a function
$\arg\max$	Argument that maximizes a function
$\lfloor\cdot\rfloor$	Floor function
$\lceil\cdot\rceil$	Ceiling function
$\bigcup_{i\in\mathbb{N}} F_i$	Countable union of sets F_i
$\Omega\backslash F$	Set complement of F in Ω
$\sum_{i\in\mathbb{N}} \mathbf{P}(F_i)$	Countable sum of probabilities
$\prod_{j=1}^{k}$	Product notation
$\sum_{j=1}^{k}$	Sum notation

Standard Sets and Domains

$\mathbb{R}$	Set of real numbers
$\mathbb{R}_{\geqslant 0}$	Set of non-negative real numbers
$\mathbb{R}^*$	Extended real numbers: $\mathbb{R} \cup \{+\infty, -\infty\}$
$\mathbb{R}^p$	p-dimensional real vector space
$\mathbb{N}$	Set of natural numbers
$\mathbb{N}^+$	Set of positive integers
$\mathbb{Z}$	Set of integers
$\mathbb{B}$	Boolean set: $\{0, 1\}$
$\mathbb{T}$	Time domain

Logic and Relations

$\forall$	Universal quantifier (for all)
$\exists$	Existential quantifier (there exists)
$\nexists$	Negated existential quantifier (there does not exist)
$\varnothing$	Empty set
$\in$	Element membership
$\subset, \subseteq$	Subset relations
$\cap$	Set intersection
$\cup$	Set union
$\sim$	Distributed as (e.g., $X \sim \mathcal{N}(0, 1)$)
$\approx$	Approximately equal
$\equiv$	Equivalent to
$\models$	Satisfies (semantic entailment)
$\not\models$	Does not satisfy
$\vDash$	Semantic satisfaction
$\leftrightarrow$	If and only if / bidirectional mapping
$\Leftrightarrow$	Logical equivalence
$\rightarrow$	Implies / function mapping
$\Rightarrow$	Logical implication
$\mapsto$	Maps to

Probability Theory and Stochastic Processes

$(\Omega, \mathcal{F}, \mathbf{P})$	Probability space with sample space Ω, σ-algebra $\mathcal{F}$ and probability measure $\mathbf{P}$
ω	Elementary outcome in sample space Ω
$\omega \in \Omega$	Element ω belongs to sample space Ω
$F \in \mathcal{F}$	Event F in σ-algebra $\mathcal{F}$
$X(\omega)$	Random variable X evaluated at outcome ω
$X^{-1}(F)$	Preimage of set F under random variable X
$\{X(t) : t \in T\}$	Stochastic process indexed by set T
X_t	Stochastic process at time t (alternative notation: $X(t)$)
$X(t, \omega)$	Stochastic process at time t and outcome ω
T	Index set (typically time domain)
$\mathcal{S}$	State space (alternative notation S)
$\mathit{Dist}(S)$	Probability distributions over state space S
$P(\cdot \mid \cdot)$	Conditional probability
$\Pr[\cdot]$	Probability notation

Trajectories and Paths

Path^{X_t}	Trajectory of stochastic process X_t
$\mathrm{Path}^{\mathcal{M}}$	Trajectory/path of model $\mathcal{M}$
$\lvert \mathrm{Path}^{X} \rvert$	Length of trajectory (number of state transitions)
w	Execution trace or trajectory
$(t_i, X(t_i))$	State-time pair in trajectory
$(X_0, t_0), (X_1, t_1), \ldots$	Sequence representation of trajectory
X_{init}	Initial state
x_0	Initial state (alternative notation)
σ_i	i-th simulation trajectory

Signal Temporal Logic (STL)

φ, ψ	STL formulae
$\varphi(\Theta)$	Parametrised STL formula with parameter set Θ
Φ	Set or vector of STL formulae
Φ_Θ	Vector of parametrised STL formulae
$\mathtt{tt}$	Boolean constant true
μ	Atomic predicate
$y(t) \bowtie c$	Atomic predicate with projection function y, relation $\bowtie$ and threshold c
$\bowtie \in \{<, \leqslant, =, \geqslant, >\}$	Comparison operators
$\neg, \wedge, \vee, \rightarrow$	Logical operators: negation, conjunction, disjunction and implication
$\mathbf{U}_{[a,b]}$	Bounded until operator on interval $[a, b]$
$\Diamond_{[a,b]}$	Eventually (finally) operator on interval $[a, b]$
$\Box_{[a,b]}$	Globally (always) operator on interval $[a, b]$
$(w, t) \vDash \varphi$	Signal w satisfies formula φ at time t (Boolean semantics)
$w \vDash \varphi$	Signal w satisfies formula φ (equivalent to $(w, 0) \vDash \varphi$)
$\rho(\varphi, w, t)$	Robustness degree of formula φ on signal w at time t (quantitative semantics)
$\rho(\varphi, w)$	Robustness degree (equivalent to $\rho(\varphi, w, 0)$)
$\chi(\psi, t)$	Boolean satisfaction function
$hrz(\varphi)$	Time horizon of formula φ

Signals and Traces

$s : T \to S$	Signal function from time domain T to state space S
s_i^w	i-th signal component of trace w
w	Execution trace
$\{s_1^w, \ldots, s_k^w\}$	Collection of k signals in trace w
$y : \mathbb{R}^p \to \mathbb{R}$	Projection function mapping p-dimensional signal to scalar
$(t_i, x(t_i), dx(t_i))$	Signal representation with time, value, and derivative

Simulation and Measures

$\mathfrak{m}$	Measure function
$\mathfrak{M}$	Set of measures $\{\mathfrak{m}_1, \mathfrak{m}_2, \ldots, \mathfrak{m}_k\}$
$\mathfrak{m} : X_t \to \mathbb{R}$	Measure mapping state to real number
$\mathrm{R} \in \mathbb{N}^+$	Number of replicas in simulation
$\mathrm{T} \in \mathbb{R}^+$	Simulation deadline
$\Delta t \in \mathbb{R}^+$	Sampling time step
`Result`	Collection of statistical summaries
$\mathrm{M}_k \in \mathbb{R}^{m \times n}$	Result matrix for measure k
$m = \lceil T/\Delta_t \rceil$	Number of time points
$n = 4$	Number of statistics (time, mean, std dev, confidence interval)
`Traj`	Individual trajectory data structure
`Replicas`	Collection of trajectory replicas
N	Number of simulations/samples

Probabilistic Model Checking and Statistical Analysis

$P(\varphi \mid \mathcal{M})$	Probability that formula φ holds in model $\mathcal{M}$
$R(\varphi \mid \mathcal{M})$	Expected robustness of formula φ in model $\mathcal{M}$
$\vec{R}(\Phi \mid \mathcal{M})$	Expected robustness vector for formula set Φ
$\mathbb{E}_{w \sim \mathcal{M}}[\rho(\varphi, w, 0)]$	Expected robustness over traces sampled from model $\mathcal{M}$
$w \sim \mathcal{M}$	Trace w sampled from model $\mathcal{M}$
$\{w \in \mathrm{Path}^{\mathcal{M}} \mid (w, 0) \models \varphi\}$	Set of paths satisfying formula φ
$\Delta_{\Phi}^{\simeq}(\mathcal{M}, \widetilde{\mathcal{M}}_{\Theta})$	Preservation distance (similarity objective)
$\Delta_{\Phi}^{+}(\mathcal{M}, \widetilde{\mathcal{M}}_{\Theta})$	Improvement distance (better performance objective)
$\Delta_{\Phi}^{-}(\mathcal{M}, \widetilde{\mathcal{M}}_{\Theta})$	Intentional divergence distance (worse performance objective)
δ	Confidence level
ϵ	Approximation parameter / error tolerance
λ	Threshold value

Optimization and Parameter Synthesis

Θ	Parameter set $\{\theta_1, \theta_2, \ldots, \theta_n\}$
θ_j	Individual parameter
θ^*	Optimal parameter value
$\mathcal{H}$	Hyperrectangle / design space / feasible parameter space
$\mathcal{H} = \prod_{j=1}^{k}[l_j, u_j]$	Hyperrectangle as Cartesian product of intervals
l_j, u_j	Lower and upper bounds for parameter θ_j
$\mathcal{X}$	Design space
$x \in \mathbb{R}^n$	Vector of design variables
$f(x)$	Objective function
$\hat{f}(x)$	Surrogate model approximation
$g_i(x)$	Inequality constraint functions
$h_j(x)$	Equality constraint functions
$\mathcal{D}$	Dataset / set of sampled points
$\mathcal{D} = \{(x_i, f(x_i))\}_{i=1}^{N}$	Dataset with function evaluations
k	Dimensionality / number of parameters
$d(\vec{x}, \vec{y})$	Distance function (e.g., Minkowski distance)
$d^*(\vec{x}, \vec{y})$	Modified distance function for maximization
p	Minkowski distance parameter
ω	Inertia weight (PSO)
c_1, c_2	Confidence coefficients (PSO)
$\delta_{mesh}, \delta_{poll}$	Mesh and poll size parameters (MADS)
τ	Mesh refinement rate (MADS)

Error Metrics and Model Evaluation

R^2	Coefficient of Determination
y_i	Actual/observed value
$\hat{y}_i$	Predicted value
$\bar{y}$	Mean of observed values
$d_{KL}(P, Q)$	Kullback-Leibler divergence
$d_{L2}(P, Q)$	L2 (Euclidean) distance
$d_{TV}(P, Q)$	Total variation distance
n	Sample size / number of data points

Sampling and Design of Experiments

P_j	Permutation for dimension j
$x_{i,j}$	Coordinate value for sample i in dimension j
$r \sim \text{Uniform}(0, 1)$	Random number from uniform distribution
$\text{Uniform}(l_j, u_j)$	Uniform distribution on interval $[l_j, u_j]$

Population Modelling Framework

$\mathcal{M}$	Stochastic model
$\mathcal{M}(\Theta)$	Parametric model
$\widetilde{\mathcal{M}}_\Theta$	Variant model
$\mathcal{M} = (\mathbf{S}, \mathbf{X}, \mathcal{D}, x_0, \mathcal{T})$	Population model with species, variables, domain, initial state, transitions
$\mathbf{S}$	Set of species in population model
$\mathbf{X}$	Vector of population variables
$\mathcal{D}$	State space (domain)
$\mathcal{T}$	Set of transition rules
τ	Individual transition rule
$\tau = (\text{name}, \mathbf{u}, \mathbf{g}, \lambda)$	Transition with consumption vector $\mathbf{u}$, generation vector $\mathbf{g}$, rate function λ
$\lambda(\mathbf{X})$	Rate function depending on state $\mathbf{X}$

Indexing and Subscripts

t_i	Time point at index i
X_i	State at index i or time i
s_i	i-th element or signal
φ_1, φ_2	Different formulae or formula components
ω_1, ω_2	Different outcomes in sample space
λ_1, λ_2	Different rate functions
$F_0, F_1, \ldots$	Sequence of events
$[a, b], [a, +\infty)$	Time intervals
init	Initial (subscript)
star	Optimal value indicator (e.g., θ^*)

Contents

Acknowledgements I

Ringraziamenti II

Summary IV

List of publications VIII

List of Abbreviations IX

Notation XII

1 Introduction 1
- 1.1 Context: Background and Motivation . . . 1
- 1.2 Key Concepts and Novel Approach . . . 2
- 1.3 Motivation and Research Goals . . . 3
- 1.4 Thesis Organisation . . . 4
  - 1.4.1 Overview of Chapters . . . 4
  - 1.4.2 Supporting Materials . . . 5

2 Quantitative Analysis 6
- 2.1 Preliminary concepts and notation . . . 7
  - 2.1.1 Law of Large Numbers and Monte Carlo Methods . . . 16
- 2.2 Modelling, system specification . . . 17
  - 2.2.1 Low-level Models . . . 17
  - 2.2.2 High-level Models . . . 28
- 2.3 Temporal Logic, formalising requirements . . . 30
  - 2.3.1 Signal Temporal Logic . . . 30
  - 2.3.2 Monitoring . . . 34
  - 2.3.3 Quantitative monitoring . . . 36
- 2.4 Analytical Analysis . . . 45
  - 2.4.1 Limitations of Analytical Methods . . . 50

2.5 Simulation-Based Statistical Verification . . . 51
2.5.1 Monte Carlo Methods in SMC . . . 51
2.5.2 Simulation Algorithms . . . 51
2.5.3 Verification Techniques . . . 53
2.6 Synthesis . . . 53
2.6.1 Definition of the Synthesis Problem . . . 54
2.6.2 Challenges in Parameter Synthesis . . . 55
2.6.3 Methods for Parameter Synthesis . . . 56
2.6.4 Applications of Parameter Synthesis . . . 56
2.7 Collective Adaptive Systems . . . 56

3 Surrogate Optimisation **60**
3.1 Preliminary Concepts and Notation . . . 61
3.1.1 Optimisation Problem Formulation . . . 61
3.1.2 Design Space and Hyperrectangles . . . 62
3.1.3 Surrogate Models . . . 62
3.1.4 Error Metrics . . . 63
3.2 Sampling Techniques . . . 63
3.2.1 Random Sampling . . . 64
3.2.2 Full Factorial Sampling . . . 64
3.2.3 Latin Hypercube Sampling . . . 65
3.3 Surrogates . . . 67
3.3.1 Dataset, Training Set, and Test Set . . . 67
3.3.2 Regression in Surrogate Modelling . . . 67
3.3.3 Metrics for Surrogate Model Evaluation . . . 68
3.3.4 Random Forest Regression . . . 68
3.3.5 Gradient Tree Boost . . . 69
3.3.6 Radial Basis Function Network . . . 70
3.4 Optimisation algorithm . . . 71
3.4.1 Particle swarm optimisation . . . 71
3.4.2 Mesh adaptive direct search . . . 72
3.4.3 Simulated Annealing . . . 76
3.5 Infill Strategies . . . 78
3.5.1 Common Infill Strategies . . . 79
3.5.2 Shrinking Trust Region with Increased Sampling . . . 79

4 State of the Art **81**
4.1 Modeling Formalisms and Languages . . . 82
4.2 Verification and Analysis Techniques . . . 84
4.3 Parameter Synthesis and Optimisation . . . 86
4.4 Software Tools and Frameworks . . . 88
4.4.1 Novel Contributions of This Work . . . 92

5 Sibilla **94**
5.1 Sibilla Architecture . . . 95
5.1.1 Sibilla front-end . . . 95
5.1.2 Sibilla back-end . . . 99

5.1.3 Sibilla Runtime . . . 100
5.2 Models Definition . . . 100
5.2.1 Language of Population Model (LPM) . . . 101
5.2.2 Language of Interactive Object (LIO) . . . 102
5.2.3 Yet anOther agent Description lAnguage (YODA) . . . 103
5.2.4 Stochastic Agent-based Modelling Language (SLAM) . . . 105
5.3 Simulating Models . . . 109
5.3.1 Language of Population Models (LPM) . . . 113
5.3.2 Simple Language for Agent Modelling (SLAM) . . . 114
5.3.3 Language of Interactive Objects (LIO) . . . 115
5.3.4 Yet anOther agent Description lAnguage (YODA) . . . 116
5.4 Analysis . . . 118
5.4.1 Reachability Analysis . . . 119
5.4.2 First Passage Time Analysis in Stochastic Models . . . 120
5.4.3 STL Formula Monitoring . . . 122
5.4.4 Validation Against Analytical Solutions . . . 125
5.5 Synthesis . . . 135
5.5.1 Theoretical Overview of Synthesis Approaches . . . 136
5.6 Framework Implementation and Design Patterns . . . 138
5.6.1 Simulation Module Implementation . . . 138
5.6.2 Analysis Module . . . 141
5.6.3 Model Module . . . 143
5.6.4 Synthesis Module . . . 145
5.6.5 Synthesis Specification . . . 148
5.6.6 Synthesis Output Format . . . 151
5.6.7 Default Synthesis Configuration . . . 152
5.6.8 Libraries Used . . . 155
5.7 Sibilla extendability . . . 156
5.7.1 Add a new model . . . 156
5.7.2 Add a new Analysis . . . 159
5.7.3 Add a new Synthesis Task . . . 163
5.7.4 Add a new Simulation Component . . . 166

6 Case Studies **170**
6.1 Parameterised Population Models . . . 170
6.2 Models Descriptions . . . 171
6.2.1 The SIR Epidemiological Model . . . 171
6.2.2 The Repressilator: A Synthetic Genetic Oscillator . . . 173
6.2.3 Three-State Protocol with Malicious Agents . . . 174
6.2.4 Spatial Three-State Protocol with Ring Topology . . . 176
6.3 Optimal Feasibility . . . 178
6.3.1 SIR Model Case Study . . . 178
6.4 Considerations on Training Set Size impact on performance . . . 180
6.4.1 Repressillator case of study . . . 183
6.5 Variant model adjustment cases . . . 184
6.5.1 Three-State Protocol (TSP) consensus variants . . . 184

7 Conclusion **193**
7.1 Research Summary and Contributions 194
7.2 Critical Evaluation 194
7.2.1 Strengths and Achievements 194
7.2.2 Limitations and Challenges 195
7.3 Future Research Directions 196
7.3.1 New Synthesis Tasks 196
7.3.2 Expanding Case Studies 197
7.3.3 Web-Based Interface Development 197
7.4 Concluding Remarks 198

Bibliography **199**

A Specification Languages for Model Definition **209**
A.1 Common syntax 209
A.2 LIO Model Specification Language 213
A.3 LPM Model Specification Language 216
A.4 YODA Model Specification Language 218
A.5 SLAM Model Specification Language 221

B Sibilla Shell Commands **223**
B.1 Commands 223

C Numerical Validation Results **231**

D Case Study Specifications **235**
D.1 SIR Model Specifications 235
D.2 Repressilator Model Specification 236
D.3 Three-State Protocol Specification 238

E Web-Based Interface Prototypes **240**
E.1 Dynamic Measurement Dashboard 240
E.1.1 Key Features 240
E.1.2 Technical Implementation 240
E.2 Synthesis Optimisation Dashboard 241
E.2.1 Dashboard Components 241
E.2.2 Visualisation Capabilities 241
E.2.3 Future Development Directions 242

CHAPTER 1

Introduction

"You can take the road that takes you to the stars now, I can take the road that'll see me through."

Nick Drake, "Road" (1972)

THIS chapter introduces the foundations and motivation for our research on the analysis and optimisation of complex computational systems that operate on a large scale. We begin by examining the challenges posed by modern systems whose behaviours emerge from countless local interactions across distributed components. When applied to such large-scale stochastic systems, the limitations of traditional analysis methods motivate our development of new approaches implemented within the Sibilla framework. This work addresses challenges in parameter synthesis, system verification, and comparative analysis. The remainder of this chapter establishes the context and background necessary to understand both the challenges we face and the solutions we propose.

1.1 Context: Background and Motivation

Contemporary technological advancements have introduced Collective Adaptive Systems (CAS) - complex systems characterised by large-scale interactions, dynamic adaptability, and emergent behaviours from numerous autonomous components. These systems are prevalent in domains ranging from smart cities to biological networks.

A more detailed definition of what qualifies as a Collective Adaptive System, along with its theoretical foundations, is provided in Section 2.7 of Chapter 5, while Section 5.2, also in Chapter 5, illustrates how our tool approaches the modelling and analysis of such systems.

The significance of CAS extends far beyond theoretical interest, manifesting in var-

ious real-world applications. In urban environments, smart city infrastructures leverage networks of sensors, actuators, and computational nodes to optimise everything from traffic flow to energy distribution. In biological systems, cellular networks demonstrate sophisticated collective behaviours. Even in social systems, the emergence of collective intelligence through digital platforms exemplifies the principles of CAS.

However, CAS present significant challenges: massive scale overwhelms traditional analysis methods, inherent stochasticity requires probabilistic frameworks, and the open nature of operating environments introduces uncertainties that traditional verification approaches cannot handle effectively. This necessitates novel analytical frameworks combining statistical approaches with computational efficiency.

The motivation for advancing our understanding and analysis of CAS stems from their increasing deployment in critical applications. As these systems become more prevalent in autonomous transportation, smart infrastructure, and healthcare domains, ensuring their correctness and reliability becomes paramount. The potential consequences of system failures or unexpected behaviours in these contexts underscore the importance of developing robust system analysis and verification methods.

This thesis addresses these challenges by developing novel approaches to analysing and optimising CAS. Through integrating surrogate-based optimisation techniques, statistical model checking (SMC), and comparative analysis frameworks, we aim to provide practical tools to understand and improve these complex systems. Our work bridges the gap between theoretical advances in system analysis and the practical needs of system designers and engineers, contributing to the broader goal of making CAS more reliable, efficient, and trustworthy.

1.2 Key Concepts and Novel Approach

Before outlining our research goals, we briefly introduce the key methodological concepts that underpin our approach.

Statistical Model Checking (SMC) provides a simulation-based alternative to exhaustive verification, using Monte Carlo methods to estimate the probability that a system satisfies given properties. Unlike traditional model checking, SMC scales to large systems by trading exact results for statistical guarantees with quantifiable confidence bounds.

Surrogate modeling addresses the computational bottleneck in parameter exploration by constructing fast-to-evaluate approximations of expensive simulation-based objective functions. Machine learning techniques (e.g., random forests, Gaussian processes) learn from a limited set of simulation runs to predict system behavior across the parameter space, enabling efficient optimization.

Our Novel Synthesis Integration: While existing tools typically provide either simulation capabilities or optimization algorithms in isolation, our work introduces the first *architecturally integrated synthesis module* within a statistical model checking framework specifically designed for stochastic collective adaptive systems. This integration enables:

- **Seamless workflow**: Direct coupling between simulation, verification, and synthesis without external tool coordination

- **Statistical validation**: Automatic verification of synthesized parameters through SMC with confidence bounds
- **Extensible architecture**: Modular design supporting multiple surrogate models and optimization strategies within a unified framework

This architectural innovation distinguishes our approach from existing parameter synthesis tools, which typically operate as standalone optimization packages requiring manual integration with simulation and verification workflows.

1.3 Motivation and Research Goals

The rapid proliferation of CAS across critical domains requires the development of robust methodologies for their analysis and optimisation. As these systems become increasingly integrated into our technological infrastructure, from smart cities to autonomous vehicle networks, the consequences of system failures or suboptimal performance become more severe. This reality drives our research towards developing more sophisticated system verification and optimisation approaches.

At the heart of our motivation lies the recognition that traditional analytical methods, while powerful for conventional systems, prove inadequate when confronting the unique challenges of CAS. The combination of massive scale, inherent stochasticity, and adaptive behaviour demands new paradigms in system analysis.

Parameter synthesis emerges as a critical challenge in this landscape. CAS typically involves numerous parameters that influence individual component behaviour and system-wide properties. These parameters often interact non-intuitively, making manual tuning impractical and potentially unreliable. The development of automated methods for parameter synthesis becomes essential for optimising system performance while ensuring that the desired properties are maintained.

Moreover, the verification of CAS properties presents unique challenges that motivate new approaches to system analysis. Based on exhaustive state space exploration, traditional verification methods become computationally intractable when applied to systems with millions of interacting components. Statistical approaches offer promise but must be carefully designed to provide meaningful guarantees while remaining computationally feasible.

Our research aims to address these challenges through several interconnected goals.

- First, we seek to develop scalable verification techniques that can handle the massive state spaces characteristic of CAS without sacrificing analytical rigor. This involves creating an approach that combines statistical sampling with intelligent exploration strategies, enabling efficient analysis of large-scale systems.
- Second, we focus on creating efficient methods for parameter synthesis, recognising that optimal system performance often depends on careful parameter tuning. Our approach leverages surrogate modelling techniques to create computationally tractable approximations of system behaviour, enabling rapid exploration of parameter spaces while maintaining acceptable accuracy.
- Third, we aim to establish methodologies for the comparative analysis of system variants. As CAS often admit multiple possible designs or implementations,

understanding the relative merits of different approaches becomes crucial. We aim to develop methodologies that enable systematic comparison of alternatives, considering both functional requirements and non-functional properties such as resilience and efficiency.

Underpinning these goals aims to advance our understanding of CAS behaviour. By developing more sophisticated analytical tools, we strive to improve system design and verification and deepen our comprehension of how complex, adaptive behaviours emerge from local interactions. This understanding has implications beyond immediate engineering applications, potentially informing fields ranging from biology to social systems.

Through our research, we strive to contribute to the foundation of a more comprehensive approach to CAS analysis and optimisation. Our goals reflect the need for theoretical advancement and practical utility, recognising that progress in understanding and managing these complex systems requires advances on multiple fronts. The methods and frameworks we develop aim to enable more reliable, efficient, and well-understood collective adaptive systems, ultimately contributing to their successful deployment in critical applications.

1.4 Thesis Organisation

This thesis is structured to present a coherent progression from theoretical foundations through practical implementation and experimental validation. The organisation reflects both the logical development of our research contributions and the interdependencies between different aspects of our work.

1.4.1 Overview of Chapters

Chapter 2: Quantitative Analysis presents the theoretical foundations for understanding stochastic model checking and parameter synthesis. The chapter begins with fundamental concepts in probability theory and stochastic processes, progressing to more advanced topics in temporal logic and formal verification. Key concepts, including Signal Temporal Logic, Continuous-Time Markov Chains, and SMC are thoroughly developed. This theoretical framework underpins the methodologies developed in subsequent chapters.

Chapter 3: Sibilla Framework introduces our analysis framework's core architecture and implementation. The chapter details the modular design of Sibilla, explaining how different components interact to enable scalable analysis of CAS. Particular attention is given to:

- The modelling layer, supporting multiple formalisms
- The simulation engine for efficient trajectory sampling
- The analysis components for property verification
- The synthesis module for parameter optimisation

Chapter 4: Study Cases demonstrates the practical application of our methods through several comprehensive case studies. Each case study is chosen to highlight different aspects of our framework:

- A biological system (Repressilator) showcasing parameter synthesis
- An epidemiological model illustrating optimal feasibility analysis
- A consensus protocol examining variant model adjustment

The chapter provides detailed analysis of results and performance metrics for each case study.

Chapter 5: Conclusion synthesises our research contributions and discusses their implications for the field. The chapter includes:

- A summary of key theoretical and practical contributions
- Critical evaluation of the developed methodologies
- Discussion of current limitations and challenges
- Directions for future research and potential extensions

1.4.2 Supporting Materials

The thesis includes several appendices that provide additional technical details and supplementary materials.

Appendix A: Specification Languages provides comprehensive documentation of the domain-specific languages used in our framework, including syntax specifications and usage examples.

Appendix B: Sibilla Commands details the complete command set available in the Sibilla shell, serving as a reference manual for framework users.

Appendix C: Case Study Specifications contains the complete specifications for all case studies, enabling reproduction of our experimental results.

CHAPTER 2

Quantitative Analysis

"I have approximate answers and possible beliefs in different degrees of certainty about different things, but I'm not absolutely sure of anything."

R. Feynman

In the realm of computer science and systems engineering, ensuring the correctness and reliability of the systems is paramount and for that reason quantitative analysis has become increasingly crucial. Real-world processes are influenced by numerous unpredictable factors, from machine breakdowns to human decision variability, introducing an element of randomness that must be accounted for in system design and verification. Traditional techniques often fail to capture this stochastic nature, resulting in overly simplified models that do not accurately represent the system's actual behaviour.

To address this challenge, a suite of sophisticated methods has emerged, collectively known as stochastic model checking. These techniques aim to provide rigorous, quantitative analysis of systems with inherent randomness, calculate the likelihood of specific events occurring during system execution, and assess overall system performance and reliability.

Stochastic model checking [152] (SMC) involves verifying systems with inherent randomness. Probabilistic model checking [19] uses exhaustive state-space exploration to verify properties of mathematical models like Markov chains, providing exact results but facing the state explosion problem.

SMC addresses this limitation by using simulation methods to estimate the system's behaviour. It runs multiple simulations and applies statistical inference to determine property satisfaction, offering better scalability for complex systems, then, based on the statistical analysis, it estimates with a given confidence level whether the property holds, providing either a probabilistic guarantee or identifying potential violations.

Although both fall under stochastic model checking, they differ in methodology. Probabilistic model checking provides exact results but struggles with large state spaces, while SMC offers probabilistic guarantees with improved efficiency for complex systems.

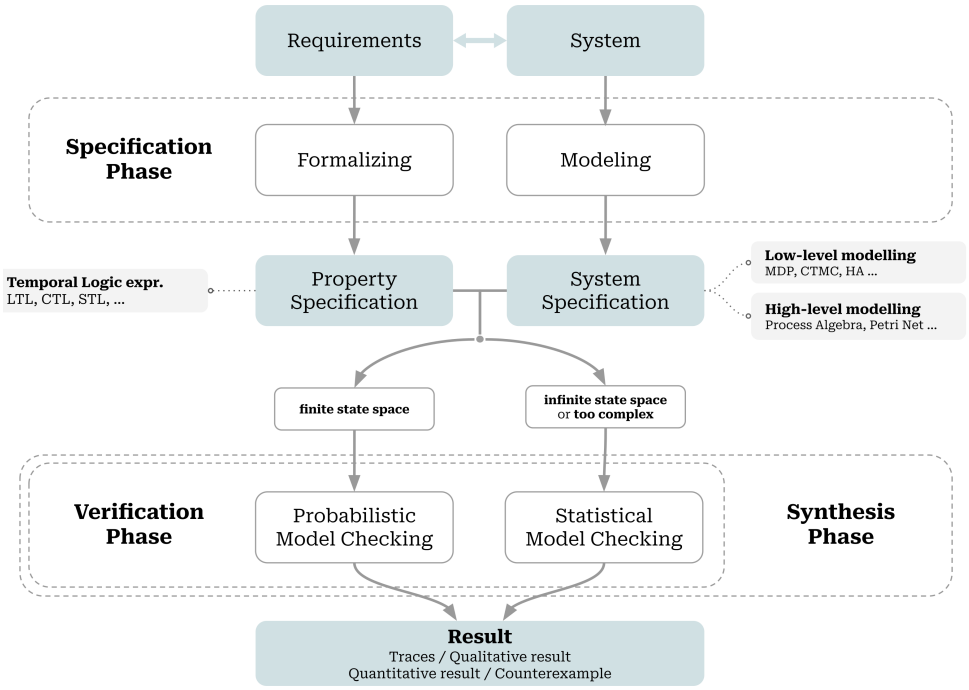


Figure 2.1: *SMC workflow*

Figure 2.1 presents a schema of SMC the workflow, illustrating the systematic approach to verifying systems with inherent randomness. This involves constructing a model of the system, defining properties that the system should adhere to, and then employing verification techniques to check these properties. The process is divided into several key phases: specification, verification, and synthesis.

This chapter will explore these phases in detail, providing a comprehensive overview of model checking. We will examine the theoretical foundations, practical applications, and tools and techniques used in the field. By the end of this chapter, the reader will have a deep understanding of model checking, its importance in ensuring system reliability, and how it is applied in practice to achieve robust system verification.

2.1 Preliminary concepts and notation

Before dissecting the components of the quantitative analysis, this section recalls the basic concepts of probability theory and establishes the notation used throughout the thesis. This section's following is based on [12, 107, 181]. Probability theory focusses on the outcome of phenomena that appear random because of a human inability to collect enough information to "calculate" the result. A σ-algebra collects all events relevant to an experiment, where an experiment is any process with a random outcome.

Definition 1 (**σ-algebra**)
A σ-algebra $\mathcal{F}$ on a set Ω is a nonempty collection of subsets of Ω (i.e., a subset of the power set $\mathcal{F} \subseteq 2^{\Omega}$) that satisfies the following properties:

- $\Omega \in \mathcal{F}$. *This means the entire set Ω is an element of the σ-algebra $\mathcal{F}$. (The requirements imply that $\varnothing \in \mathcal{F}$ since $\Omega \in \mathcal{F}$ and $\Omega \backslash \Omega = \varnothing$.)*
- *If $F_0, F_1, \ldots \in \mathcal{F}$, then $(\bigcup_{i \in \mathbb{N}} F_i) \in \mathcal{F}$ (i.e., $\mathcal{F}$ is closed under countable union). This property ensures that if we can describe individual events (subsets of Ω) using the σ-algebra, we can also describe any event formed by taking a countable union of these individual events.*
- *If $F \in \mathcal{F}$, then $(\Omega \backslash F) \in \mathcal{F}$ (i.e., $\mathcal{F}$ is closed under complement).*

There is also a need for a measure defining the probability of each event of an experiment and its structure. A probability measure for the σ-algebra, $\mathcal{F}$, assigns a value from 0 to 1 to each event $F \in \mathcal{F}$, where 0 indicates that F is unlikely to occur and one indicates that F is almost certain to occur.

Definition 2 (**Measurable Space**)
A measurable space is a pair $(\Omega, \mathcal{F})$ where Ω is a set and $\mathcal{F}$ is a σ-algebra over Ω

A probability space is built upon a measurable space but includes a probability measure that quantifies the likelihood of events.

Definition 3 (**Probability Space**)
A probability space is a triple $(\Omega, \mathcal{F}, \mathbf{P})$ where $(\Omega, \mathcal{F})$ is a measurable space and $\mathbf{P}$ is a probability function, which assigns, to each event in the event space, a probability $\mathbf{P} : \mathcal{F} \to [0, 1]$ with the conditions

- $\mathbf{P}(\Omega) = 1$
- *if events $F_0, F_1 \ldots$ are pair-wise disjoint sets in $\mathcal{F}$ then $\mathbf{P}\left(\bigcup_{i \in \mathbb{N}} F_i\right) = \sum_{i \in \mathbb{N}} \mathbf{P}\left(F_i\right)$, that ensure that $\mathbf{P}$ is additive.*

In Example 2.1.1, a detailed explanation of the probability space for a dice roll with specific probability values and subsets is shown.

Example 2.1.1 (Probability space of a dice roll)
Considering the throw of a standard dice, the probability space can be characterised as follows:

- Ω : The sample space Ω is typically the set $\{⚀, ⚁, ⚂, ⚃, ⚄, ⚅\}$, where each element in the set is a label which represents the outcome of the die landing on that label.
- $\mathcal{F}$: The event space $\mathcal{F}$ could be the set of all subsets of the sample space, which would then contain simple events F such as $\{⚄\}$ ("the die lands on ⚄ "), as well as complex events such as $\{⚁, ⚃, ⚅\}$ ("the die lands on an even number").
- $\mathbf{P}$: The probability function $\mathbf{P}$ that map each event $F \in \mathcal{F}$ to probability value calculated as the ratio of the number of outcomes in the event F to the total numbero of possibile outcomes in the sample space Ω so for example, $\{⚄\}$ would be mapped to $1/6$, and $\{⚁, ⚃, ⚅\}$ would be mapped to $3/6 = 1/2$.

In probability and statistics, the focus is often not on the specific outcome of a single experiment but instead on a summary or aggregate measure that can be derived from the outcomes. This is where random variables come into play.

The study of random variables is motivated by the fact that in many scenarios, one might not be interested in the precise elementary outcome of a random experiment but rather in some numerical function of the outcome.

> *"Random variables are much like Guinea pigs, neither a pig, nor from Guinea. Random variables are functions, which are deterministic by definition."*

The term random variable is a misnomer because a random variable is neither random nor a variable. A random variable X is a function from the sample space Ω to $\mathbb{R}$. The term 'random' signifies the underlying randomness in picking an element ω from the sample space Ω. Once the elementary outcome ω is fixed, the random variable takes a fixed real value, $X(\omega)$. It is important to remember that the probability measure is associated with subsets (events), whereas a random variable is associated with each elementary outcome ω.

Definition 4 (**Random Variable**)
Let $(\Omega_1, \mathcal{F}_1, \mathbf{P})$ be a probability space and $(\Omega_2, \mathcal{F}_2)$ be a measurable space then a $(\Omega_2, \mathcal{F}_2)$-valued random variable is a function

$$X : \Omega_1 \to \Omega_2$$

such that for any $F \in \mathcal{F}_2$,

$$X^{-1}(F) = \{f \in \Omega_1 \mid X(f) \in F\} \in \mathcal{F}_1.$$

Definition 4 states that a random variable is a function that maps from a probability space to a measurable space and that given a random variable one can always define a measure $\mathbf{P}_2$ on $(\Omega_2, \mathcal{F}_2)$ as $\mathbf{P}_2(F) = \mathbf{P}_1\left(X^{-1}(F)\right)$ for all $F \in \mathcal{F}_2$.

Definition 4 is general in the co-domain of the random variable. This means that the random variable can map outcomes from the sample space to various sets and structures, as long as these mappings preserve measurability (see Example 2.1.1). However, we focus on real-valued random variables for many practical purposes and the most common applications in probability and statistics. These are random variables where the codomain is the set of real numbers $\mathbb{R}$ i.e. those with codomain given by the measurable space $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$ where $\mathcal{B}(\mathbb{R})$ denotes the Borel σ-algebra on $\mathbb{R}$.

Although random variables provide a snapshot of randomness at a single point, many real-world phenomena involve randomness that evolves or space. To model such dynamic random behaviour, we introduce the concept of stochastic processes. A stochastic process can be considered a collection of random variables indexed by time or another parameter, allowing us to describe how randomness unfolds across a continuum.

"What is a stochastic process? Oh, it's just one darn thing after another." ***R.A. Fisher***

Stochastic processes are useful mathematical constructs to describe the random evolution of a system in time.

While a stochastic process describes a system's overall behaviour, it is often useful to consider individual realisations of this process. These realisations are called trajectories and provide a concrete way to visualise and analyse the evolution of a stochastic process.

Definition 5 (Stochastic process)
Given a probability space $(\Omega, \mathscr{F}, \mathbf{P})$, a stochastic process is any collection of random variables defined on this probability space. In other words, a stochastic process is a collection of random valued variables :

$$\{X(t) : t \in T\}$$

*where **T** is the index set. A stochastic process can also be written as*

$$\{X(t, \omega) : t \in T, \omega \in \Omega\}$$

To reflect that it is a function of two variables, $t \in T$ and $\omega \in \Omega$
The value of the stochastic process at time t could be denoted X_t, $X(t)$, $X_t(\omega)$ or $X(t, \omega)$.

The index set $\mathbf{T}$ determines the type of stochastic process. For instance, $\mathbf{T} = 0, 1, 2, \ldots$ corresponds to discrete-time stochastic processes, commonly represented as $X_n \ : \ n \in \mathbb{N}$. When $\mathbf{T} = [0, \infty)$, it describes continuous-time stochastic processes, referred to as $X_t \ : \ t \geqslant 0$.

Depending on the choice of the index set T, we distinguish between the following types of stochastic processes:

- **Single-element Stochastic Process**: If T consists of just one element (called, say, 1), then a stochastic process reduces to just one random variable $X_1 : \Omega \to \mathbb{R}$. So, the concept of a stochastic process includes the idea of a random variable as a special case.
- **Random Vector**: If $T = \{1, \ldots, n\}$ is a finite set with n elements, then a stochastic process reduces to a collection of n random variables $X_1, \ldots, X_n$ defined on the same probability space. This collection is called a *random vector*. So, the concept of a stochastic process includes the idea of a random vector as a special case.
- **Discrete-time Stochastic Process**: Stochastic processes with index sets $T = \mathbb{N}, T = \mathbb{Z}, T = \mathbb{N}^d, T = \mathbb{Z}^d$ (or any other countable set) are called *stochastic processes with discrete time*.
- **Continuous-time Stochastic Process**: Stochastic processes with index sets $T = \mathbb{R}, T = \mathbb{R}^d, T = [a, b]$ (or other similar uncountable sets) are called *stochastic processes with continuous time*.
- **Random Fields**: Stochastic processes with index sets $T = \mathbb{R}^d, T = \mathbb{N}^d$ or $T = \mathbb{Z}^d$, where $d \geqslant 2$, are sometimes called *random fields*.

Definition 6 (**Trajectory**)
Given a stochastic process $X(t, \omega) : t \in T$ defined on a probability space $(\Omega, \mathscr{F}, \mathbf{P})$, a trajectory is a function:

$$X(\cdot, \omega) : T \to S$$

for any fixed $\omega \in \Omega$, where: T is the index set (often interpreted as time) and S is the state space.
In other words, a trajectory is obtained by fixing ω and considering the mapping from the index set T to the state space S. Mathematically, for a fixed $\omega \in \Omega$, trajectory maps:

$$\text{Path}^{X_t} : t \mapsto X(t, \omega), \quad \textit{for all } t \in T$$

It represents a single possible outcome of the stochastic process.

A trajectory represents the evolution of a stochastic process over its index set. It can be understood as a sequence of observations of the random variables X_t, where each X_t represents the state of the stochastic process at a specific point t in the index set.

The parameter t in X_t can have various interpretations depending on the context. Most commonly, t represents time (i.e., X_t could denote the price of a stock at time t). In some applications, t represents spatial coordinates. For instance, X_t might represent the air temperature at location $t = (u, v)$ in a two-dimensional space. In more complex scenarios, t can represent both space and time. For example, X_t could denote the air temperature at spatial coordinates (u, v) and time s, where $t = (u, v, s)$ is a point in three-dimensional space-time.

We define $|\,\mathrm{Path}^X\,|$ as the trajectory's length: the number of recorded state transitions.

Example 2.1.2 illustrates how ω represents a specific outcome sequence in the sample space and how it determines the values of X_t for all t, thus defining a particular trajectory of the stochastic process.

Example 2.1.2 (Coin Toss Process)
Let's consider a simple stochastic process of repeated coin tosses:

- Let X_t represent the outcome of a coin toss at time t, where $t \in \mathbb{N}$ (i.e., we toss the coin at discrete time points 1, 2, 3,...).
- For each toss, $X_t = 1$ if the outcome is heads, and $X_t = 0$ if it's tails.
- The sample space Ω consists of all possible infinite sequences of heads and tails.

Now, let's consider some specific ω's:

- $\omega_1 = (H, T, H, H, T, H, \ldots)$ could represent one possible sequence of outcomes.
- $\omega_2 = (T, T, T, H, H, T, \ldots)$ could represent another.

For ω_1, we would have:

$$\begin{aligned} X_1(\omega_1) &= 1 \text{ (Heads)} \\ X_2(\omega_1) &= 0 \text{ (Tails)} \\ X_3(\omega_1) &= 1 \text{ (Heads)} \\ X_4(\omega_1) &= 1 \text{ (Heads)} \\ &\vdots \end{aligned}$$

The trajectory $\mathrm{Path}^{X_t}(\omega_1)$ would be the sequence $(1, 0, 1, 1, 0, 1, ...)$.
For ω_2, we would have a different trajectory
$\mathrm{Path}^{X_t}(\omega_2) = (0, 0, 0, 1, 1, 0, ...)$.
Each ω in Ω corresponds to one possible "universe" or "timeline" of coin toss outcomes. The stochastic process X_t encompasses all these possibilities, and when we observe a specific realisation, we're essentially picking one specific ω from Ω.

We often work with several stochastic models $\mathcal{M}$ that represent different stochastic processes in practical applications. In these cases, the index set $\mathbf{T}$ corresponds to the time domain, and thus, the parameter t represents a specific time coordinate. We assume that the model $\mathcal{M}$ manages the outcome ω and provides it as needed. This allows us to simplify the definition of the trajectory function to:

$$\mathrm{Path}^{\mathcal{M}} : t \mapsto X(t)$$

A trajectory of the model $\mathcal{M}$ is represented by a sequence of time-state pairs:

$$\text{Path}^{\mathcal{M}} = \left((t_0, X(t_0)), (t_1, X(t_1)), \ldots, (t_n, X(t_n)), \ldots\right)$$

where $X(t_0)$ is the initial state. This representation provides a practical way to work with and analyse trajectories in various applications of stochastic processes.

In our work with trajectories, we will treat them as Right-Closed Constant Piecewise Functions. This is a slight modification to the standard definition of a constant piecewise function, where the last interval is closed rather than open.

Definition 7 (Right-Closed Constant Piecewise Function (RCPF))
A function $f : [a_0, a_n] \to \mathbb{R}$ is called a right-closed constant piecewise function if there exists a partition of $[a_0, a_n]$ into a finite number of intervals, where all but the last are half-open, such that:

1. *The intervals are of the form $[a_i, a_{i+1})$ for $i = 0, 1, ..., n-2$, and $[a_{n-1}, a_n]$ for the last interval.*
2. *The intervals are disjoint and cover $[a_0, a_n]$, i.e., $\bigcup_{i=0}^{n-2}[a_i, a_{i+1}) \cup [a_{n-1}, a_n] = [a_0, a_n]$.*
3. *For each interval, there exists a constant $c_i \in \mathbb{R}$ such that $f(x) = c_i$ for all x in that interval.*
4. *The intervals are pairwise disjoint, i.e., $[a_i, a_{i+1}) \cap [a_j, a_{j+1}) = \varnothing$ for $i \neq j$.*
5. *Each interval has a positive length, i.e., $a_{i+1} - a_i > 0$ for all $i = 0, 1, ..., n-1$.*

Formally, we can write:

$$f(x) = \sum_{i=0}^{n-2} c_i \cdot \mathbb{1}_{[a_i, a_{i+1})}(x) + c_{n-1} \cdot \mathbb{1}_{[a_{n-1}, a_n]}(x)$$

where $\mathbb{1}_{[a_i, a_{i+1})}(x)$ and $\mathbb{1}_{[a_{n-1}, a_n]}(x)$ are the indicator functions of the respective intervals.

An indicator function can be defined as follows:

Definition 8 (Indicator Function)
Let A be a set. The indicator function of A, denoted by $\mathbb{1}_A$, is a function defined on a universal set X (typically understood from context) as:

$$\mathbb{1}_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

Let's finally introduce the notion of distribution function. Random variables are

often used to calculate the probabilities of events. For example, if we are interested in a sum of at least 8, we could experiment with throwing two dice. Usually, when dealing with a random variable X, for constants a and $b(b < a)$, computation of one or several of the probabilities $P(X = a), P(X < a), P(X \leqslant a), P(X > b)$, $P(X \geqslant b), P(b \leqslant X \leqslant a), P(b < X \leqslant a), P(b \leqslant X < a)$, and $P(b < X < a)$ is our ultimate goal. For this reason, we calculate $P(X \leqslant t)$ for all $t \in (-\infty, +\infty)$. As we will show shortly, if $P(X \leqslant t)$ is known for all $t \in \mathbf{R}$, then for any a and b, all the probabilities mentioned above can be calculated. In fact, since the real-valued function $P(X \leqslant t)$ characterises X, it tells us almost everything about X. This function is called the distribution function of X.

Definition 9 (Cumulative Distribution Function (CDF))
If X is a random variable, then the function F defined on $(-\infty, +\infty)$ by $F(t) = P(X \leqslant t)$ is called the distribution function of X.
Since F "accumulates" all the probabilities of the values of X up to and including t, sometimes it is called the cumulative distribution function F_X of X.

$$F_X(x) = \mathrm{P}(X \leqslant x)$$

To sample from a CDF, inverse transform sampling is a basic method for pseudorandom number sampling. Inverse transform sampling takes uniform samples of a number u between 0 and 1 (interpreted as a probability). Then it returns the smallest number $x \in \mathbb{R}$ such that $F(x) \geqslant u$ for the cumulative distribution function F of a random variable.

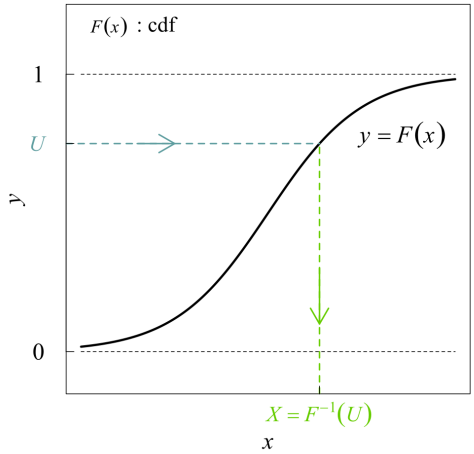
Definition 10 (Inverse Transform Sampling)
For any random variable $X \in \mathbb{R}$, the random variable $F_X^{-1}(U)$ has the same distribution as X, where F_X^{-1} is the generalised inverse of the cumulative distribution function F_X of X and U is uniform on $(0, 1)$.

For continuous random variables, the inverse probability integral transform is indeed the inverse of the probability integral transform, which states that for a continuous random variable X with cumulative distribution function F_X, the random variable $U = F_X(X)$ is uniform on $(0, 1)$.

Example 2.1.3 (Sampling from the Exponential Distribution)
Consider the exponential distribution with cumulative distribution function (CDF) given by:

$$F_X(x) = \begin{cases} 1 - e^{-\lambda x} & \text{for } x \geqslant 0, \\ 0 & \text{otherwise.} \end{cases}$$

This distribution is characterised by the parameter $\lambda > 0$.



To sample from this distribution using the inverse transform method, we solve the equation $y = F_X(x)$ for x. The inverse function $x = F_X^{-1}(y)$ is derived as follows:

$$x = F_X^{-1}(y) = -\frac{1}{\lambda}\ln(1-y), \tag{2.1}$$

This implies that if we draw a random variable y from a uniform distribution $U \sim \text{Unif}(0,1)$ and compute the inverse function in Equation 2.1, then x will follow an exponential distribution with parameter λ. The derivation of the inverse transform is shown step-by-step below:

$$\begin{aligned} U &= 1 - e^{-\lambda \cdot x} & x &= -\frac{1}{\lambda} \cdot \ln(1-U) \\ U - 1 &= -e^{-\lambda \cdot x} & x &= \frac{1}{\lambda} \cdot \ln\left(\frac{1}{1-U}\right) \\ 1 - U &= e^{-\lambda \cdot x} & x &= \frac{1}{\lambda} \cdot \ln\left(\frac{1}{U}\right) \\ \ln(1-U) &= -\lambda \cdot x & x &= -\frac{1}{\lambda} \cdot \ln(U) \end{aligned}$$

Here, u is a random variable uniformly distributed on the interval (0,1). Each step confirms the transformation needed to obtain a sample from the exponential distribution using inverse transform sampling.

$$F_X(x) = P(X \leqslant x) = 1 - e^{-\lambda x}$$

2.1.1 Law of Large Numbers and Monte Carlo Methods

After introducing methods for sampling from distributions, it is essential to understand the foundational theorems that justify the use of sampled data in estimating statistical properties. The Law of Large Numbers provides the theoretical underpinning for such estimations, ensuring that averages computed from large samples tend to be close to the expected value.

Definition 11 (**Law of Large Numbers**)
Let $X_1, X_2, \ldots, X_n$ be a sequence of independently and identically distributed (i.i.d.) random variables with finite expected value $\mu = \mathbb{E}[X_i]$. Then, the sample average

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

converges in probability to μ as $n \to \infty$. Formally,

$$\lim_{n\to\infty} \mathbf{P}\left(\left|\bar{X}_n - \mu\right| > \epsilon\right) = 0, \quad \textit{for all } \epsilon > 0.$$

The Law of Large Numbers ensures that as the number of trials increases, the sample average $\bar{X}_n$ becomes a more accurate estimator of the expected value μ. This theorem forms the foundation for many statistical estimation techniques.

Monte Carlo methods use random sampling to compute numerical estimates of quantities that may be difficult or impossible to calculate analytically. By simulating random samples and applying the Law of Large Numbers, these methods provide practical solutions to complex problems.

Definition 12 (**Monte Carlo Method**)
Monte Carlo methods are computational algorithms that use repeated random sampling to obtain numerical results. For estimating the expected value $\mu = \mathbb{E}[X]$ of a random variable X, the Monte Carlo estimator is defined as:

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} X_i,$$

where $X_1, X_2, \ldots, X_n$ are independent samples drawn from the distribution of X.
By the Law of Large Numbers (Definition 11), the estimator $\hat{\mu}_n$ converges in probability to μ as $n \to \infty$, providing an unbiased estimate of μ.

Monte Carlo methods are instrumental when dealing with high-dimensional integrals or complex probability distributions where analytical solutions are infeasible.

2.2 Modelling, system specification

> *"All models are approximations. Essentially, all models are wrong, but some are useful. However, the approximate nature of the model must always be borne in mind."*
>
> George E.P. Box

Modelling systems requires careful consideration of several key factors, including the treatment of time (whether as continuous or discrete), the approach to representing system states (either by analysing each agent or by counting the number of agents in each state), the choice between stochastic, deterministic, or hybrid dynamics, and the treatment of physical space (whether continuous or discrete). These decisions shape the modelling approach and determine the framework used to represent and analyse the system. In modelling, we distinguish between two levels of abstraction: low-level formalisms, such as automata-based approaches, and high-level formalisms. Most high-level modelling languages map onto extensions of low-level formalisms (like automata), where the concepts of states (potentially with internal structure) and transitions (which connect states and may include various annotations) are central.

2.2.1 Low-level Models

Modelling is a field that offers a variety of approaches to analysing and representing systems. At one end of the spectrum are deterministic models, which assume that the system's behaviour is entirely predictable based on its initial conditions and the governing equations. These models are typically represented by differential or algebraic equations and are well-suited for systems where randomness and uncertainty are negligible or irrelevant.

As we move along the spectrum, stochastic models introduce randomness into the modelling process, acknowledging that many real-world systems are subject to inherent variability that cannot be captured by deterministic approaches alone. These models incorporate random variables or probabilistic events to account for aspects of the system that are not entirely predictable. Examples include Markov chains and stochastic differential equations.

Further along this continuum are probabilistic models, a subset of stochastic models. These models go beyond acknowledging randomness by explicitly quantifying uncertainty through probability distributions. In probabilistic models, the randomness inherent in the system is precisely described using discrete probability distributions, which handle countable outcomes, such as the number of successes in a series of trials, or continuous probability distributions, which relate to uncountable outcomes, often represented by real numbers, such as the time between events.

In the context of automata-based formalisms, we see a reflection of this spectrum from deterministic to probabilistic approaches. Labelled Transition Systems (LTS) represent the most basic, low-level formalism, allowing for nondeterministic choices without probabilistic considerations. As we progress to more complex formalisms such as stochastic hybrid automata (SHA), we incorporate continuous dynamics and stochastic elements, offering a richer and more flexible framework for modelling systems that

exhibit a combination of discrete events, continuous evolution, and randomness.

In Figure 2.2 we list some of the most well known automata-based formalisms, and graphically show their relationships (with a higher-up formalism being an extension of the lower-level formalisms it is connected to).

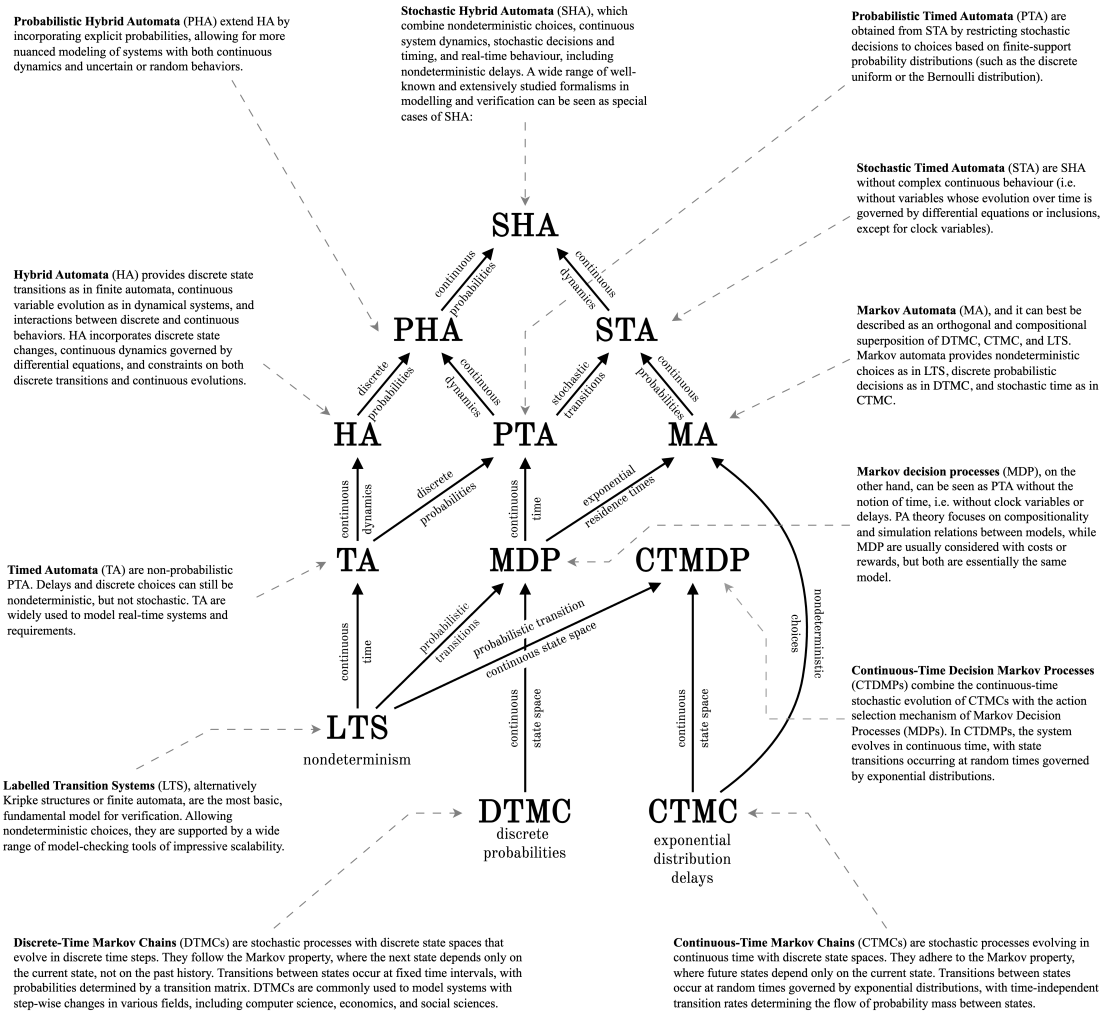


Figure 2.2: *A comprehensive overview of various automata and process models, showing their relationships and key characteristics.*

We will delve deeper into Discrete-Time Markov Chains (DTMCs), Continuous-Time Markov Chains (CTMCs), and Stochastic Hybrid Automata (SHA). These models serve as the foundational building blocks for the high-level models discussed in this thesis, which will be further explored in subsequent chapters.

Discrete Time Markov Chain

A Discrete-Time Markov Chain (DTMC) chain [19] consists of a set of states and a set of transitions between states. Each state is associated with a probability distribution over the transitions, i.e., the successor state is selected probabilistically. The main property of a DTMC is that the probabilities in the current state are independent of what happened in the past. While DTMCs can be either time-homogeneous (where transition

probabilities are constant over time) or time-inhomogeneous (where they may vary), the following definition specifically describes the time-homogeneous case.

Definition 13 (**Discrete-Time Markov Chain**)
A DTMC is a tuple $\mathcal{M} = (\mathcal{S}, \mathbf{P}, AP, L, s_0)$*, where:*

- $\mathcal{S}$ *is a finite, non-empty set of states.*
- $s_0 \in \mathcal{S}$ *is the initial state.*
- $\mathbf{P} : \mathcal{S} \times \mathcal{S} \to [0, 1]$ *is the transition probability function, where* $\mathbf{P}(s, s')$ *gives the probability of transitioning from state* s *to state* s'*. For each* $s \in \mathcal{S}$*, it holds that*
$$\sum_{s' \in \mathcal{S}} \mathbf{P}(s, s') = 1.$$
- AP *is a finite set of atomic propositions.*
- $L : \mathcal{S} \to 2^{AP}$ *is a labelling function that assigns to each state* $s \in \mathcal{S}$ *a set* $L(s) \subseteq AP$ *of atomic propositions that hold in* s*.*

The essential characteristic of DTMC is the Markov property, which encapsulates the memoryless nature of the process. This property states that the future state of the process depends only on the present state and not on the sequence of events that preceded it. In other words, given the present state, the past states do not provide additional information about the future states.

To formally define this, we consider a stochastic process $\{X_t\}_{t \geqslant 0}$, where X_t represents the state of the process at time t. The Markov property is then described as follows:

Definition 14 (**Markov Property for DTMC**)
A stochastic process $\{X_t\}_{t \geqslant 0}$*, taking values in the finite state space* $\mathcal{S}$*, is called a DTMC if for all times* $t \geqslant 0$ *and all states* $s_0, s_1, \ldots, s_t, s_{t+1} \in \mathcal{S}$*:*

$$\begin{aligned} \mathbf{P}\left(X_{t+1} = s_{t+1} \,|\, X_t = s_t, X_{t-1} = s_{t-1}, \ldots, X_0 = s_0\right) \\ = \mathbf{P}\left(X_{t+1} = s_{t+1} \,|\, X_t = s_t\right) \\ = \mathbf{P}(s_t, s_{t+1}). \end{aligned}$$

$\mathbf{P}(s_t, s_{t+1})$ denotes the probability that the chain, whenever in state s_t, moves next (one unit of time later) into state s_{t+1}, and is referred to as a one-step transition probability. The square matrix $\mathbf{P} = [\mathbf{P}(s, s')],\ s, s' \in \mathcal{S}$, is called the transition probability matrix. Since when leaving state s the chain must move to one of the states $s' \in \mathcal{S}$, each row sums to one (forms a probability distribution): For each $s \in \mathcal{S}$,

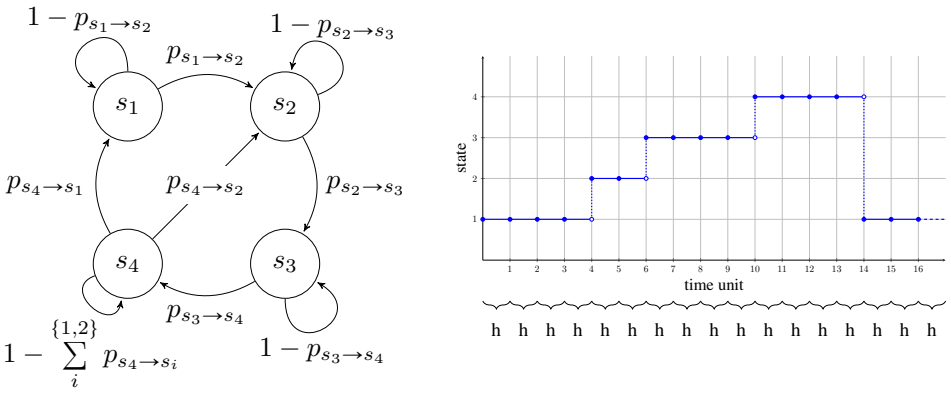
$$\sum_{s' \in \mathcal{S}} \mathbf{P}(s, s') = 1.$$

The defining Markov property (Definition 14) can be described as the future be-

ing independent of the past given the present state. Letting t be the present time, the future after time t is $\{X_{t+1}, X_{t+2}, \ldots\}$, the present state is X_t, and the past is $\{X_0, X_1, \ldots, X_{t-1}\}$. If the value $X_t = s_t$ is known, then the future evolution of the chain depends only on s_t and is stochastically independent of the past values $X_{t-1}, \ldots, X_0$.

Figure 2.3 illustrates the behavior of a DTMC, showing both the state transition diagram and a sample execution trace.

DTMC example



Transition probability matrix

$$\mathbf{P} = \begin{bmatrix} 1 - p_{s_1 \to s_2} & p_{s_1 \to s_2} & 0 & 0 \\ 0 & 1 - p_{s_2 \to s_3} & p_{s_2 \to s_3} & 0 \\ 0 & 0 & 1 - p_{s_3 \to s_4} & p_{s_3 \to s_4} \\ p_{s_4 \to s_1} & p_{s_4 \to s_2} & 0 & 1 - (p_{s_4 \to s_1} + p_{s_4 \to s_2}) \end{bmatrix}$$

State transition

Holding time	Next state
$\mathrm{h} = t_{unit}$	$P\left(X_{t+\mathrm{h}_i} = s_j \middle| X_t = s_i\right) = P_{ij}$

Figure 2.3: *Discrete-Time Markov Chain (DTMC): Diagram illustrating state transitions with corresponding probabilities, a transition probability matrix, and a visualisation of state changes over discrete time intervals.*

Continuous Time Markov Chain

A Continuous Time Markov Chain [182] extends the DTMC model by incorporating real-time information into the states. In particular, each state is associated with an exponential distribution that gives the system time to stay in that state.

Definition 15 (Continuous-Time Markov Chain)
A continuous-time Markov chain () is a tuple $\mathcal{M} = (\mathcal{S}, \mathbf{Q}, AP, L, s_0)$, *where:*

- $\mathcal{S}$ *is a finite, non-empty set of states.*
- $s_0 \in \mathcal{S}$ *is the initial state.*
- $\mathbf{Q} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ *is the infinitesimal generator matrix (or generator matrix), where:*
  - *For* $s \neq s'$, $\mathbf{Q}(s, s') \geqslant 0$ *represents the rate of transitioning from state* s *to state* s'.
  - *The diagonal elements are defined as*

$$\mathbf{Q}(s, s) = -\sum_{\substack{s' \in \mathcal{S} \\ s' \neq s}} \mathbf{Q}(s, s'),$$

  ensuring that the rows sum to zero:

$$\sum_{s' \in \mathcal{S}} \mathbf{Q}(s, s') = 0, \quad \forall s \in \mathcal{S}.$$

- AP *is a finite set of atomic propositions.*
- $L : \mathcal{S} \to 2^{AP}$ *is a labelling function that assigns to each state* $s \in \mathcal{S}$ *a set* $L(s) \subseteq AP$ *of atomic propositions that hold in* s.

Similarly to DTMCs,CTMCs possess the essential memoryless property. The Markov property for CTMC is described as follows:

Definition 16 (Markov Property for CTMC)
A stochastic process $\{X(t) : t \geqslant 0\}$, *taking values in the finite state space* $\mathcal{S}$, *is called a continuous-time Markov chain if for all times* $t, h \geqslant 0$ *and all states* $s, s' \in \mathcal{S}$:

$$\begin{aligned} \mathbf{P}\left(X(t+h) = s' \,|\, X(t) = s, \{X(u) : 0 \leqslant u < t\}\right) \\ = \mathbf{P}\left(X(t+h) = s' \,|\, X(t) = s\right) \end{aligned}$$

Transitions between states are determined by "competing" clocks with timers that go off at random, exponentially distributed times. For each state s, there is a clock associated with each state s' the process could move to in a single transition from state s. Let $C(s)$ be the set of states in which CTMC can move to the state s in a single transition. Equivalently, $C(s)$ is the set of active clocks in state s.

Each time CTMC moves to state s, clocks with timers are set or reset, if necessary, to go off at random times $\mathrm{h}_{s \to s'}$ for each $s' \in C(s)$. Each clock has an exponential timer; that is, random time $\mathrm{h}_{s \to s'}$ is given an exponential distribution with (positive finite) rate

$\mathbf{Q}(s, s')$ and thus mean $1/\mathbf{Q}(s, s')$ (depending on s and s').

Due to the properties of the exponential distribution, with probability 1, no two timers ever go off simultaneously. (Since the exponential distribution is continuous, the likelihood of any two timers coinciding is zero.)

Let h_s (called sojourn time or holding time) be the time until the first timer goes off in state s, and let N_s be the state corresponding to this timer:

$$\begin{aligned} \mathrm{h}_s &\equiv \min_{s' \in C(s)} \{\mathrm{h}_{s \to s'}\}, \\ N_s &\equiv s' \quad \text{such that} \quad \mathrm{h}_{s \to s'} = \mathrm{h}_s. \end{aligned}$$

The process then moves from state s to state N_s after an elapsed time of h_s, and the process repeats from the new state N_s.

To understand the implications of these exponential clocks, recall the basic properties of the exponential distribution. The minimum of several independent exponential random variables is again an exponential random variable with a rate equal to the sum of the rates. Hence, h_s has an exponential distribution:

$$\mathbf{P}(\mathrm{h}_s \leqslant t) = 1 - e^{-\nu_s t}, \quad t \geqslant 0,$$

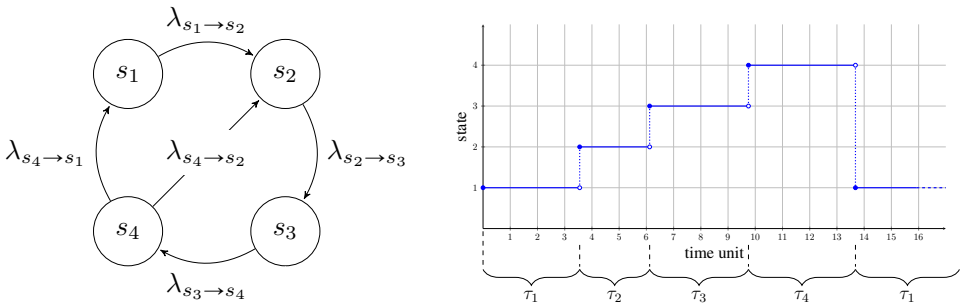
where the exiting rate ν_s is defined as

$$\nu_s \equiv -\mathbf{Q}(s, s) = \sum_{\substack{s' \in \mathcal{S} \\ s' \neq s}} \mathbf{Q}(s, s').$$

Next, recall that when considering several independent exponential random variables, each exponential random variable is the one yielding the minimum with a probability proportional to its rate. Therefore,

$$\mathbf{P}(N_s = s') = \frac{\mathbf{Q}(s, s')}{\sum_{\substack{k \in \mathcal{S} \\ k \neq s}} \mathbf{Q}(s, k)} = \frac{\mathbf{Q}(s, s')}{\nu_s}, \quad \text{for } s' \neq s.$$

automata and a possible execution



Infinitesimal generator matrix

$$\mathbf{Q} = \begin{bmatrix} -\lambda_{s_1 \to s_2} & \lambda_{s_1 \to s_2} & 0 & 0 \\ 0 & -\lambda_{s_2 \to s_3} & \lambda_{s_2 \to s_3} & 0 \\ 0 & 0 & -\lambda_{s_3 \to s_4} & \lambda_{s_3 \to s_4} \\ \lambda_{s_4 \to s_1} & \lambda_{s_4 \to s_2} & 0 & -(\lambda_{s_4 \to s_1} + \lambda_{s_4 \to s_2}) \end{bmatrix}$$

State transition

Holding time | Next State

$$\mathrm{h}_s \sim \mathrm{Exp}\left(\textstyle\sum_{s' \neq s} \mathbf{Q}(s, s')\right) \qquad \mathbf{P}\left(X_{t+\mathrm{h}_s} = s' \mid X_t = s\right) = \frac{\mathbf{Q}(s, s')}{\sum_{k \neq s} \mathbf{Q}(s, k)}$$

$$y \sim \mathrm{Exp}(x) \text{ corresponds to } y = \tfrac{1}{x} \ln\left(\tfrac{1}{u}\right) \text{ with } u \sim \mathrm{Uniform}(0, 1)$$

Figure 2.4: *Continuous-Time Markov Chain (): Diagram showing state transitions with associated rates, a infinitesimal generator matrix, and a plot demonstrating stochastic state transitions over continuous time based on exponential distributions.*

Stochastic Hybrid Automata

A stochastic hybrid automaton (SHA) [137] extends the deterministic hybrid automaton model by incorporating stochastic elements into continuous dynamics. In particular, the flow within each discrete state is governed by a Stochastic Differential Equation (SDE) rather than a deterministic one.

An SDE is a differential equation in which one or more terms are stochastic processes, leading to a solution that is a stochastic process. SDEs are used to model systems that are influenced by random noise. A general form of an SDE is:

$$dX(t) = f(X(t), t)\, dt + g(X(t), t)\, dB_t,$$

Where $X(t)$ is the state variable, f is the drift coefficient, g is the diffusion coefficient, and B_t is a Brownian motion (also known as the Wiener process). In the context of hybrid automata, an **invariant** is a condition or a set of conditions that define the valid continuous states within a particular discrete state. The invariant specifies the subset of the constant state space where the system can reside while in a given discrete state.

When the constant state reaches the boundary of the invariant set, a transition (or discrete jump) may occur to another discrete state.

Definition 17 (**Stochastic Hybrid Automaton**)
A stochastic hybrid automaton (SHA) $\mathcal{H}$ is a tuple $(\mathcal{Q}, \mathcal{X}, \mathrm{Inv}, f, g, G, R, q_0)$, where:

- *$\mathcal{Q}$ is a countable set of discrete states;*
- *$\mathcal{X} = \mathbb{R}^N$ is the continuous state space for some $N \in \mathbb{N}$;*
- *$q_0 \in \mathcal{Q}$ is the initial discrete state;*
- *$\mathrm{Inv} : \mathcal{Q} \to 2^{\mathcal{X}}$ assigns to each $q \in \mathcal{Q}$ an invariant open subset of $\mathcal{X}$;*
- *$f, g : \mathcal{Q} \times \mathcal{X} \to T\mathcal{X}$ are vector fields defining the drift and diffusion coefficients, respectively;*
- *$G : E \to 2^{\mathcal{X}}$ assigns to each $e \in E$ a guard $G(e)$, where $E = \mathcal{Q} \times \mathcal{Q}$, such that:*
  - *For each $e = (q, q') \in E$, $G(e)$ is a measurable subset of $\partial\mathrm{Inv}(q)$ (possibly empty);*
  - *For each $q \in \mathcal{Q}$, the family $\{G(e) : e = (q, q')$ for some $q' \in \mathcal{Q}\}$ is a disjoint partition of $\partial\mathrm{Inv}(q)$;*
- *$R : E \times \mathcal{X} \to \mathcal{P}(\mathcal{X})$ assigns to each $e = (q, q') \in E$ and $x \in G(e)$ a reset probability measure on $\mathcal{X}$ concentrated on $\mathrm{Inv}(q')$. Here, $\mathcal{P}(\mathcal{X})$ denotes the family of all probability measures in $\mathcal{X}$. Furthermore, for any measurable set $A \subset \mathrm{Inv}(q')$, $R(e, x)(A)$ is a measurable function in x.*

The stochastic nature of SHA is captured by the SDE that governs the flow within each discrete state. The evolution of the continuous state $X(t)$ in the discrete state q is described by the stochastic differential equation:

$$dX(t) = f(q, X(t))\, dt + g(q, X(t))\, dB_t, \tag{2.2}$$

where B_t is a standard Brownian motion.

Definition 18 (**Stochastic Execution**)
A stochastic process $(X(t), Q(t)) \in \mathcal{X} \times \mathcal{Q}$ *is called a* stochastic execution *of the SHA* $\mathcal{H}$ *if there exists a sequence of stopping times* $\tau_0 = 0 \leqslant \tau_1 \leqslant \tau_2 \leqslant \cdots$ *such that for each* $n \in \mathbb{N}$*:*

- *In each interval* $[\tau_n, \tau_{n+1})$, $Q(t) \equiv Q(\tau_n)$ *is constant, and* $X(t)$ *is a (continuous) solution to the SDE:*

$$dX(t) = f(Q(\tau_n), X(t))\, dt + g(Q(\tau_n), X(t))\, dB_t;$$

- $\tau_{n+1} = \inf\{t \geqslant \tau_n : X(t) \notin \mathrm{Inv}(Q(\tau_n))\}$*;*
- $X(\tau_{n+1}^-) \in G(Q(\tau_n), Q(\tau_{n+1}))$*, where* $X(\tau_{n+1}^-)$ *denotes* $\lim_{t \uparrow \tau_{n+1}} X(t)$*;*
- *The probability distribution of* $X(\tau_{n+1})$ *given* $X(\tau_{n+1}^-)$ *is governed by the law* $R(e_n, X(\tau_{n+1}^-))$*, where* $e_n = (Q(\tau_n), Q(\tau_{n+1})) \in E$*.*

An important concept in analysing SHAs is the embedded Markov chain:

Definition 19 (**Embedded Markov Chain**)
Let $Q_n \triangleq Q(\tau_n)$ *and* $X_n \triangleq X(\tau_n)$*. Then* $\{(Q_n, X_n), n \geqslant 0\}$ *is called* embedded Markov chain *for the stochastic execution* $(X(t), Q(t))$*.*

The transition probability of the embedded Markov chain can be characterised as follows:

$$P(Q_{n+1} = q', X_{n+1} \in dx' \mid Q_n = q, X_n = x) = \int_{y \in G(e)} R(e, y)(dx')\, P(Y_x(\eta) \in dy), \tag{2.3}$$

where $e = (q, q')$, $Y_x(t)$ is the solution to SDE

$$dY(t) = f(q, Y(t))\, dt + g(q, Y(t))\, dB_t, \quad Y(0) = x,$$

and $\eta = \inf\{t \geqslant 0 : Y_x(t) \notin \mathrm{Inv}(q)\}$ is the first exit time of $Y_x(t)$ from $\mathrm{Inv}(q)$.

The analysis of SHAs often focusses on properties such as reachability, stability, and recurrence, which can be studied through the behaviour of the embedded Markov chain.

Example: To clarify the above concepts, consider the following simple example. Two cars, labelled 1 and 2 with car 2 in the lead, are moving from left to right on a highway (see Figure 2.5). Due to various random factors such as road conditions, wind, and the presence of human operators, the movements of both cars are stochastic. If we absorb all the randomness into the motion of car 1 and ignore the possible occurrence of emergency braking, then the motion of car 2 can be modelled as having a constant speed v_2. Let Δx be the distance between the two cars. Let $d_0 > d_1 > d_2 > d_3 > 0$ be four thresholds. We propose the following hybrid control scheme for car 1 (see the diagram in Figure 2.5): It consists of three discrete states $\{1, 2, 3\}$ corresponding to chasing, keeping, and braking, respectively.

1. **Chasing**: In this stage, $\Delta x \geqslant d_2$, and car one will try to catch car two at speed $v_1 > v_2$. So the perturbed motion of car 1 is governed by $\dot{x}_1 = v_1 - v_2 + d\,B_t$, where B_t is a standard one-dimensional Brownian motion;
2. **Keeping**: In this stage $d_3 \leqslant \Delta x \leqslant d_1$, and car 1 will try to move at speed v_2 under the perturbation $d\,B_t$;
3. **Braking**: If $\Delta x \leqslant d_3$, car one will brake according to some prescribed procedure until $\Delta x = d_0$. For simplicity, we ignore the presence of noise during braking.

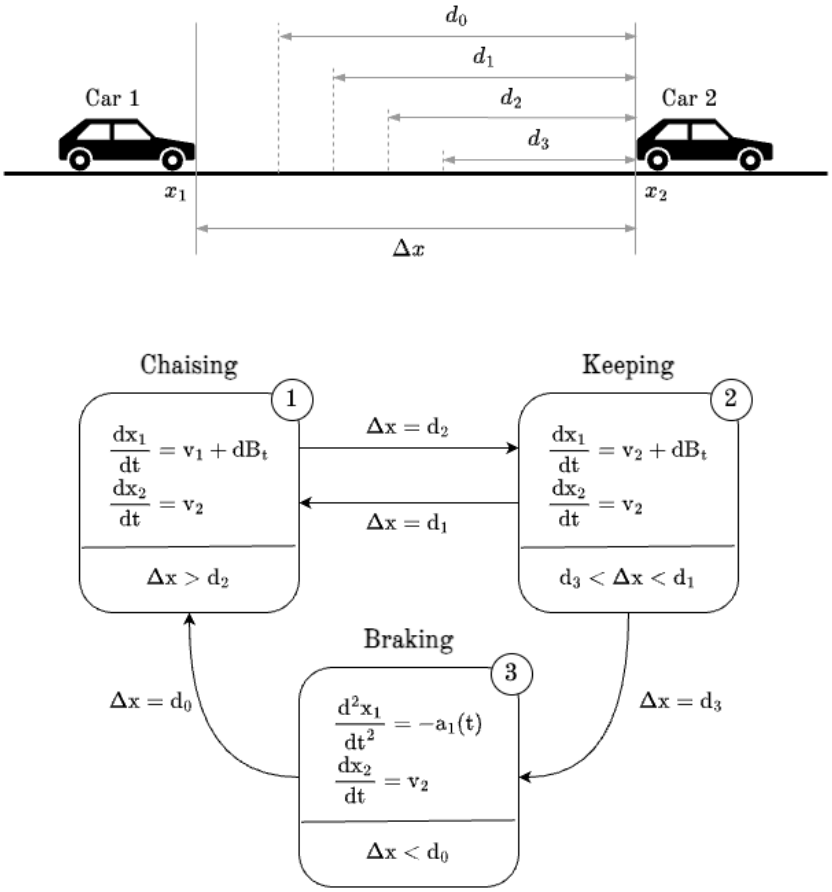


Figure 2.5: *Scheme of chasing cars example modeled as a Stochastic Hybrid Automaton*

The invariant sets and guards for each discrete state are also shown in Figure 2.5. The reset kernels are trivial; more precisely, $R(e, x)$ is concentrated at x for any $e = (q, q') \in E$ and any $x \in G(e)$. Hence, we can define the two-car system as a tuple $(\mathcal{Q}, \mathcal{X}, \mathrm{Inv}, f, g, G, R)$, where:

- $\mathcal{Q} = \{1, 2, 3\}$ is the set of discrete states:
  - 1: Chasing state;
  - 2: Keeping state;
  - 3: Braking state.

- $\mathcal{X} = \mathbb{R}$ is the continuous state space, representing the distance Δx between the two cars.
- $\mathrm{Inv} : \mathcal{Q} \to 2^{\mathcal{X}}$ is the invariant set for each discrete state:

$$\begin{aligned} \mathrm{Inv}(1) &= [d_2, \infty), && \text{(Chasing when } \Delta x \geqslant d_2\text{)} \\ \mathrm{Inv}(2) &= [d_3, d_1], && \text{(Keeping when } d_3 \leqslant \Delta x \leqslant d_1\text{)} \\ \mathrm{Inv}(3) &= (0, d_3], && \text{(Braking when } \Delta x \leqslant d_3\text{)} \end{aligned}$$

- $f : \mathcal{Q} \times \mathcal{X} \to \mathbb{R}$ is the deterministic drift term:

$$\begin{aligned} f(1, x) &= v_1 - v_2, && \text{(Chasing: car 1 moves faster than car 2)} \\ f(2, x) &= 0, && \text{(Keeping: car 1 matches speed with car 2)} \\ f(3, x) &= -a, && \text{(Braking: car 1 decelerates)} \end{aligned}$$

- $g : \mathcal{Q} \times \mathcal{X} \to \mathbb{R}$ is the diffusion coefficient:

$$\begin{aligned} g(1, x) &= d, && \text{(Noise in chasing state)} \\ g(2, x) &= d, && \text{(Noise in keeping state)} \\ g(3, x) &= 0, && \text{(No noise in braking state)} \end{aligned}$$

- $G : E \to 2^{\mathcal{X}}$ is the set of guard conditions, where $E = \mathcal{Q} \times \mathcal{Q}$:

$$\begin{aligned} G((1, 2)) &= \{d_2\}, && \text{(Transition from chasing to keeping)} \\ G((2, 1)) &= \{d_1\}, && \text{(Transition from keeping to chasing)} \\ G((2, 3)) &= \{d_3\}, && \text{(Transition from keeping to braking)} \\ G((3, 2)) &= \{d_0\}, && \text{(Transition from braking to keeping)} \end{aligned}$$

- $R : E \times \mathcal{X} \to \mathcal{P}(\mathcal{X})$ is the reset map:

$$R(e, x) = \delta_x \quad \text{for all } e \in E \text{ and } x \in G(e),$$

  Where δ_x denotes the Dirac delta measure concentrated at x.

This SHA model captures the stochastic nature of the two-car system, where the behaviour of car 1 switches between chasing, keeping distance, and braking based on the distance to car 2, with added noise in the chasing and keeping states to represent random factors in the car's motion.

The successive visits to the discrete states $\{Q_n\}$ form a Markov chain. The probability transition matrix P can be derived based on the dynamics and thresholds:

$$P = \begin{pmatrix} 0 & 1 & 0 \\ p & 0 & 1-p \\ 1 & 0 & 0 \end{pmatrix},$$

where

$$p = \frac{d_2 - d_3}{d_1 - d_3}.$$

In this example, p represents the probability that the system transitions from the keeping state back to the chasing state, based on the stochastic dynamics within the keeping state's invariant set.

2.2.2 High-level Models

High-level modelling approaches offer a more expressive and abstract framework than low-level models. While low-level models, such as differential equations or finite automata, focus on detailed, component-level interactions, high-level models capture the overall system behaviour and complex interactions more holistically. These advanced formalisms represent intricate system behaviours, concurrency, and adaptability, which would be cumbersome or infeasible to model using low-level approaches alone. High-level models often build upon the foundations laid by low-level models, extending and specialising them to address specific modelling needs and facilitate analysis at different abstraction levels.

The following paragraphs introduce several key high-level modelling approaches, each offering unique strengths in representing different aspects or types of systems.

Process Algebras [29, 171, 188] Process algebras provide a compositional approach for specifying concurrent systems, offering a formal syntax and semantics to describe complex systems in terms of simpler components and their interactions. They incorporate operators for parallel composition, choice, and sequencing of processes, along with communication primitives for synchronisation between processes. Process algebras support hierarchical system descriptions and enable formal reasoning about system properties. When modelling systems with complex dynamics, process algebras can be extended with attribute-based communication to model context-aware interactions, explicit representation of locations or spaces, stochastic aspects to capture quantitative behaviour, and dynamic process creation and destruction to model open systems.

Rule-based Systems [55, 103] Rule-based systems use sets of rules to specify system behaviour and transitions, making them particularly suited for modelling systems with many similar interacting entities. These systems consist of rules with conditions and actions that use pattern matching to determine when the rules apply. They have the potential to generate emergent behaviour from simple local rules and naturally represent adaptive behaviour through rule modifications. In complex systems, rule-based approaches offer advantages in modelling individual entity behaviours and interactions, capturing system-wide adaptation through dynamic rule sets, and representing environmental influences on entity behaviour.

Petri Nets [126] Petri nets are graphical modelling tools well suited to represent concurrent and distributed systems. They consist of places, transitions, and tokens, providing an intuitive visual representation of system states and events. Petri nets offer clear representation of system state and state changes, enable modelling of concurrent and asynchronous events, support hierarchical structuring for complex systems, and provide a mathematical foundation for formal analysis. For systems with complex dynamics, the extensions include coloured Petri nets to represent different types of entities or resources, stochastic Petri nets to model quantitative aspects and time, and adaptive Petri nets where the structure of the net can change dynamically.

Boolean/Qualitative Networks [148, 201] Boolean networks and their generalisations provide a discrete, qualitative approach to modelling biological regulatory networks and

other complex systems. They represent system components as variables with discrete states and use update rules to define state transitions based on the states of different components. This approach can capture complex regulatory relationships and is suitable for analysing the stability and attractors of the system. Extensions and variants of Boolean networks include multivalued networks allowing more than two states per variable, probabilistic Boolean networks incorporating uncertainty in state transitions, and temporal extensions to capture timing aspects of regulatory processes.

Statecharts [121] Statecharts provide a visual formalism for modelling reactive systems, including biological processes. They feature a hierarchical organisation of states, concurrent states to represent parallel processes, events and conditions governing transitions between states, and actions associated with state changes. For modelling complex systems, statecharts offer an intuitive representation of multiscale processes, the ability to model different levels of abstraction within a single diagram, and support for modelling adaptive behaviour through dynamic state configurations.

Hybrid Systems [5] Hybrid systems combine discrete state transitions with continuous dynamics, making them suitable for modelling systems that exhibit both discrete and continuous behaviours. They integrate discrete events and continuous processes, allowing modelling of switching between different continuous dynamics. Hybrid systems support modelling of both qualitative and quantitative aspects of a system and are suitable for systems with multiple time scales. In complex system modelling, hybrid systems can be used to represent decision-making processes alongside continuous dynamics, model the interaction between discrete events and continuous population-level dynamics, and capture the interplay between digital control systems and physical processes.

Spatio-temporal Models [90,167] Spatio-temporal Models are a notable category of models closely related to those examined in this study. These models distinguish themselves by simulating the collective behaviour emerging from local and non-local interactions within continuous state deterministic systems. Within this category, there are several specialised types of models:

- Compartment-based models, which specialise in organising systems into distinct compartments.
- Agent-based models, which involve autonomous entities that exhibit dynamic behaviours.
- Lattice-based models employ a repeated graph structure to effectively capture interconnected processes occurring at various levels of system organisation.

These models are particularly useful for systems where spatial distribution and temporal evolution are crucial factors. Compartment-based models are adequate for systems with well-defined spatial divisions, such as cellular organelles or ecosystem habitats. Agent-based models represent systems with many individual entities making autonomous decisions, such as crowd dynamics or animal flocking behaviours. Lattice-based models are powerful tools for studying phenomena like crystal growth or cellular

automata, where spatial relationships are regular and critical to system behaviour. All these spatio-temporal approaches provide valuable insights into how spatial arrangements and temporal dynamics interact to produce complex system behaviours, making them essential tools in studying many systems with intricate dynamics.

2.3 Temporal Logic, formalising requirements

Temporal logics [15, 66, 120, 186] (TL) are very concise languages to specify the occurrence of specific temporal behaviours rigorously.

One of the most popular TL is Linear Temporal Logic (LTL), introduced by Pnueli in 1977 [186] to reason about the order of events occurring during the execution of a programme. LTL operates on a single path of model execution, and a temporal property can be formulated only for one possible trajectory of the system. Other temporal logics such as computational tree logic (CTL) [66] and CTL* [102] have in their syntax special quantifiers that enable the specification of properties over all possible trajectories or branches in time.

All the aforementioned temporal logics consider only the temporal order of the events and not the actual time at which they occur. For example, it is impossible to specify that a formula should hold after two units of time and before three and a half units of time. Even if we decide to discretise the time, recording all the events at each time step, the syntax of these logics is not equipped to deal directly with the specification of real-time intervals.

Real-time temporal logics [7–9,168] overcome these limits by using continuous time semantics and embedding a time interval in the until temporal operator.

2.3.1 Signal Temporal Logic

The Signal Temporal Logic (STL) [96,168] is an example of a real-time temporal logic suitable for many biological case studies [25, 48, 93, 94, 192]. STL extends LTL with the continuous-time semantics and with predicates over real variables

STL builds upon Metric Interval Temporal Logic (MITL) [7], adding parametric numerical predicates as atomic propositions. STL offers both boolean and quantitative semantics: boolean semantics assess whether a given trace complies with the STL specification, while quantitative semantics provides a measure of the robustness of the specification for a specific trace.

To monitor and evaluate systems' behaviour over time, we need to introduce the concepts of signals and traces. A signal captures the values of the variables in a system over a time interval, while a trace records the sequence of these values over time. The following definitions formalise these notions.

Definition 20 (Signal and Trace)
*A **signal** s is a function $s : T \to S$, where:*

- *T is the time domain, represented as a real-valued interval $[0, T] \subseteq \mathbb{R}_{\geqslant 0}$, for some $T > 0$*
- *S is the state space, which can be:*
  - *$S = \mathbb{B} = \{0, 1\}$ for Boolean signals*
  - *$S = \mathbb{R}^* = \mathbb{R} \cup \{+\infty, -\infty\}$ for real-valued or quantitative signals*

*An execution **trace** w, which corresponds to a trajectory in our earlier notation, can be defined as:*

$$w = \text{Path}^{\mathcal{M}} = ((t_0, X(t_0)), (t_1, X(t_1)), \ldots, (t_n, X(t_n)), \ldots)$$

Where:

- *$X(t_i)$ represents the state of the stochastic process at time t_i*
- *Each $X(t_i)$ can be viewed as a collection of k signals: $\{s_1^w, \ldots, s_k^w\}$ and each $s_i^w : T \to S$ is a signal as defined above*

This definition of a trace w aligns with the concept of a trajectory introduced in the preliminary concepts section while also incorporating the idea of multiple signals that can represent different aspects or dimensions of the system's state at each time point.

The trace w can be treated as a Right-Closed Constant Piecewise Function (RCPF) as defined earlier, where the function values correspond to the states $X(t_i)$ at each time point t_i.

Definition 21
(STL Syntax) The syntax of STL is defined as follows:

$$\varphi := \overbrace{\texttt{tt}}^{\text{True}} \mid \overbrace{\mu}^{\text{Atomic pred.}} \mid \underbrace{\overbrace{\neg\varphi}^{\text{Not}} \mid \overbrace{\varphi_1 \wedge \varphi_2}^{\text{And}}}_{\text{Logic op.}} \mid \underbrace{\overbrace{\varphi_1 \, \mathbf{U}}^{\text{Until}} \overbrace{_{[a,b]}}^{0 \leqslant a < b} \varphi_2}_{\text{Temporal op.}}$$

Where $\texttt{tt}$ is the Boolean constant true; μ is an atomic proposition of the form $y(t) \bowtie c$, with $y : \mathbb{R}^p \to \mathbb{R}$ projecting the p-dimensional signal onto a single variable, $\bowtie \in \{<, \leqslant, =, \geqslant, >\}$ and $c \in \mathbb{R}^+$ being a threshold; $\neg$ and $\wedge$ are the usual Boolean connectives, and finally $\mathbf{U}$ is a binary operator *Until*: φ_1 has to hold at least until φ_2 becomes true, which must hold at the current or a future position within the time interval $[a, b]$, with $a, b \in \mathbb{T}$ and $a < b$.

In addition to the basic syntax, STL allows for the definition of derived logical and temporal operators. These operators help build more expressive specifications using

the primary constructs introduced above.

Definition 22
(Derived STL Operators) The following logical and temporal operators can be derived:

Or	$\varphi_1 \vee \varphi_2$	$=$	$\neg(\neg\varphi_1 \wedge \neg\varphi_2)$
Implication	$\varphi_1 \rightarrow \varphi_2$	$=$	$\neg\varphi_1 \vee \varphi_2$
Eventually (or Finally)	$\Diamond_{[a,b]}\varphi$	$=$	$\mathtt{tt}\ \mathbf{U}_{[a,b]}\varphi$
Globally (or Always)	$\Box_{[a,b]}\varphi$	$=$	$\neg\Diamond_{a,b}\neg\varphi$

Then we define two kinds of semantics: with the *Boolean semantics*, the satisfaction assumes Boolean values (does satisfy, does not satisfy); with *the quantitative semantics*, the satisfaction assumes real-valued value.

A trace w satisfies φ, denoted by $w \vDash \varphi$, if and only if $(w, 0) \vDash \varphi$.

Definition 23
(STL Boolean Semantics) The Boolean satisfaction relation $\vDash$ for STL formula φ on a temporal trace w is defined recursively by:

$$
\begin{aligned}
&(w,t) \vDash \mathtt{tt} \\
&(w,t) \vDash \mu && \Leftrightarrow\ y(\mathtt{s}_i^w) \bowtie c \\
&(w,t) \vDash \neg\varphi && \Leftrightarrow\ (w,t) \nvDash \varphi \\
&(w,t) \vDash \varphi \wedge \psi && \Leftrightarrow\ (w,t) \vDash \varphi \wedge (w,t) \vDash \psi \\
&(w,t) \vDash \varphi \mathbf{U}_{[a,b]} \psi && \Leftrightarrow\ \exists t' \in [t+a, t+b] : (w,t') \vDash \psi \\
& && \quad \wedge\ \ \forall t'' \in [t, t'), (w,t'') \vDash \varphi
\end{aligned}
$$

A trace w satisfied φ, denoted by $w \vDash \varphi$ if and only if $(w, 0) \vDash \varphi$

As shown in the following, Boolean semantics only specify whether a trace satisfies a property (true or false). However, this binary approach may sometimes lack nuance, especially when the degree of satisfaction is essential to assess robustness or tolerance to deviations.

To address this, quantitative semantics is introduced, which assigns a real-valued score to the satisfaction of the STL formula. This score, referred to as the degree of robustness, measures how strongly the trace satisfies (or violates) the formula, providing valuable information on the degree of adherence. In the following, we formally define both Boolean and quantitative semantics.

Definition 24
(STL Quantitative Semantics) The quantitative satisfaction function $\rho : \varphi \times w \times [0, \infty) \to \mathbb{R}^*$ returns a value $\rho(\varphi, w, t) \in \mathbb{R}^*$ quantifying the robustness degree of the property φ by the signal w at time t. It is defined recursively as follows:

$$\begin{array}{lcl}
\rho(\mathtt{tt}, w, t) & = & +\infty \\
\rho(\mu, w, t) & = & \begin{cases} \bowtie \in [>, \geqslant] & : \quad c - y(\mathrm{s}_i^w) \\ \bowtie \in [<, \leqslant] & : \quad y(\mathrm{s}_i^w) - c \end{cases} \\
\rho(\neg\varphi, w, t) & = & -(\varphi, w, t) \\
\rho(\varphi \wedge \psi, w, t) & = & \min\{\rho(\varphi, w, t), \rho(\psi, w, t)\} \\
\rho(\varphi \mathbf{U}_{[a,b]} \psi, w, t) & = & \sup\limits_{t' \in [t+a, t+b]} \min \left\{ \rho\left(\psi, w, t'\right), \inf\limits_{t'' \in [t, t']} \left\{ \rho\left(\varphi, w, t''\right) \right\} \right\}
\end{array}$$

Moreover, we let $\rho(\varphi, w) := \rho(\varphi, w, 0)$

Trajectories $\mathrm{Path}^{\mathcal{M}}$ can be monitored as trace w, and a highly efficient monitoring algorithm for STL robustness introduced in [95] was implemented to evaluate ρ. Because we consider the signal as càdlàg piecewise constant function, we have adapted the algorithm presented in [95]. The temporal operators we consider are all time-bounded, implying that time-bounded trajectories are sufficient to evaluate the truth of every formula. To know the time required to evaluate the truth of a formula, we define a function time horizon $hrz(\varphi)$.

Definition 25
(Time Horizion) We define the time horizon as a function $hrz : \varphi \to \mathbb{R}_{\geqslant 0}$ recursively as follows:

$$\begin{array}{lcl}
hrz(\mathtt{tt}) & = & 0 \\
hrz(\mu) & = & 0 \\
hrz(\neg\varphi) & = & hrz(\varphi) \\
hrz(\varphi \wedge \psi) & = & \max(hrz(\varphi), hrz(\psi)) \\
hrz(\varphi \mathbf{U}_{[a,b]} \psi) & = & b + \max(hrz(\varphi), hrz(\psi))
\end{array}$$

Using Boolean semantics, we can use probabilities to assess the likelihood of a formula φ being satisfied within the model $\mathcal{M}$. This is expressed as:

$$P\left(\varphi \mid \mathcal{M}\right) = P\left(\left\{w \in \mathrm{Path}^{\mathcal{M}} \mid (w, 0) \models \varphi\right\}\right)$$

This formulation quantifies the probability that φ is true given the model $\mathcal{M}$ under consideration.

Extending beyond Boolean semantics, we integrate quantitative semantics with path probabilities to derive a real-valued random variable. Our primary interest lies in its

expectation, denoted as:

$$R(\varphi \mid \mathcal{M}) = \mathbb{E}_{w \sim \mathcal{M}}[\rho(\varphi, w, 0)]$$

Here, $R(\varphi \mid \mathcal{M})$ represents the expected value of a quantitative measure $\rho(\varphi, x, 0)$, where w is sampled from the model $\mathcal{M}$.

We also denote a formula as $\varphi(\Theta)$ as a parametrised STL formula, where Θ represents a set of parameters that can be adjusted.

2.3.2 Monitoring

Monitoring is a critical task in analysing and verifying real-time systems, where the goal is to determine whether an execution trace satisfies a given temporal logic specification. In the context of STL, monitoring involves checking if a signal w satisfies an STL formula φ at a particular time t, denoted $(w, t) \models \varphi$.

There are two primary approaches to monitoring in STL:

- **Qualitative Monitoring**: Determines the satisfaction of the property in a Boolean sense (true or false).
- **Quantitative Monitoring**: Calculate the degree of robustness of satisfaction, providing a measure of how strongly the property is satisfied or violated.

The theoretical foundations presented in this section will be implemented in Chapter 5 within the Sibilla framework, where they enable the evaluation of system behaviours against temporal specifications through real-time simulation-based algorithms.

Qualitative Monitoring

In qualitative monitoring, the satisfaction of an STL formula by a trace is evaluated using Boolean semantics. The procedure is recursive and operates on the syntax tree of φ, constructing a Boolean satisfaction function $\chi(\psi, t)$ for each subformula ψ:

$$\chi(\psi, t) = \begin{cases} 1 & \text{if } (w, t) \models \psi, \\ 0 & \text{otherwise.} \end{cases}$$

The Boolean satisfaction relation is defined recursively on the basis of the syntax of STL:

- **Atomic Predicates**: For an atomic predicate μ, $(w, t) \models \mu$ if the predicate is satisfied at time t by the signal w.
- **Boolean Operators**:
  - Negation: $(w, t) \models \neg\psi$ if $(w, t) \not\models \psi$.
  - Conjunction: $(w, t) \models \psi_1 \wedge \psi_2$ if $(w, t) \models \psi_1$ and $(w, t) \models \psi_2$.
  - Disjunction: $(w, t) \models \psi_1 \vee \psi_2$ if $(w, t) \models \psi_1$ or $(w, t) \models \psi_2$.
- **Temporal Operators**:
  - *Until*: $(w, t) \models \psi_1 \, \mathbf{U}_{[a,b]} \, \psi_2$ if there exists $t' \in [t + a, t + b]$ such that $(w, t') \models \psi_2$ and for all $t'' \in [t, t']$, $(w, t'') \models \psi_1$.

To compute the satisfaction function $\chi(\psi, t)$ for temporal operators, particularly the Until operator, we introduce the concept of *interval shifting*.

Interval Shifting

Let $I = [m, n)$ and $[a, b]$ be intervals in $\mathbb{T}$ (the non-negative real numbers). The $[a, b]$-*back-shifting* of I is defined as:

$$I \ominus [a, b] = \{t \in \mathbb{T} \mid t + [a, b] \subseteq I\} = [m - b, n - a) \cap \mathbb{T}.$$

This operation effectively shifts the interval I backward in time by the bounds a and b, accounting for the non-negative time domain $\mathbb{T}$. Figure 2.6 illustrates the back-shifting operation for different cases.

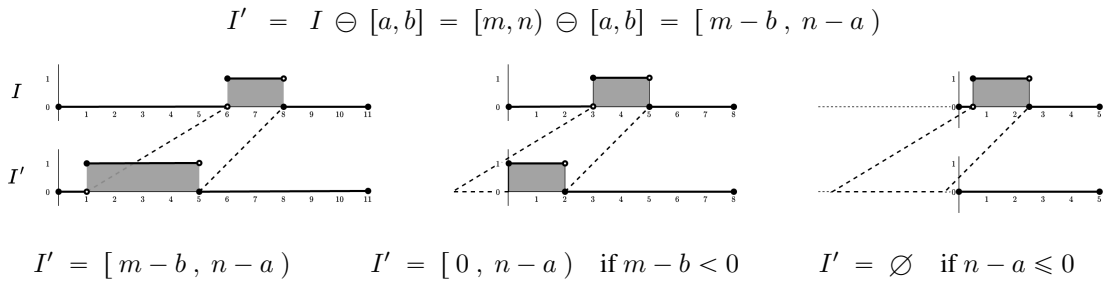


Figure 2.6: *Examples of back-shifting an interval $I = [m, n)$ by $[a, b]$: (a) $I' = [m - b, n - a)$; (b) $I' = [0, n - a)$ if $m - b < 0$; (c) $I' = \varnothing$ if $n - a \leqslant 0$.*

Computing Temporal Operators with Interval Shifting

We can efficiently compute the satisfaction functions for temporal operators using interval shifting. For instance, consider the **Eventually** operator.

Eventually Operator The satisfaction function for $\Diamond_{[a,b]}\psi$ can be computed by backward-shifting the intervals where ψ holds:

$$\chi(\Diamond_{[a,b]}\psi, t) = 1 \quad \text{if and only if} \quad t \in \bigcup_{I \in I_\psi} (I \ominus [a, b]),$$

where I_ψ is the set of intervals where $\chi(\psi, t) = 1$.

Always Operator Similarly, for the **Always** operator $\Box_{[a,b]}\psi$, the satisfaction function is computed by considering the complement:

$$\chi(\Box_{[a,b]}\psi, t) = 1 \quad \text{if and only if} \quad t \notin \bigcup_{I \in I_{\neg\psi}} (I \ominus [a, b]),$$

where $I_{\neg\psi}$ is the set of intervals where $\chi(\psi, t) = 0$.

Until Operator For the **Until** operator $\varphi = \psi_1 \, \mathbf{U}_{[a,b]} \, \psi_2$, the satisfaction function can be computed recursively using interval operations. The procedure involves:

1. Identify the intervals I_{ψ_2} where $\chi(\psi_2, t) = 1$.

2. For each interval $I \in I_{\psi_2}$, back-shift I by $[a, b]$ to obtain $I' = I \ominus [a, b]$.
3. Within I', verify that $\chi(\psi_1, t) = 1$ holds continuously from t up to the corresponding point in I.
4. The satisfaction function $\chi(\varphi, t)$ is 1 where the above conditions are met.

By integrating interval shifting into the qualitative monitoring procedure, we can handle time-bounded temporal operators effectively and accurately determine the satisfaction of temporal properties over continuous signals.

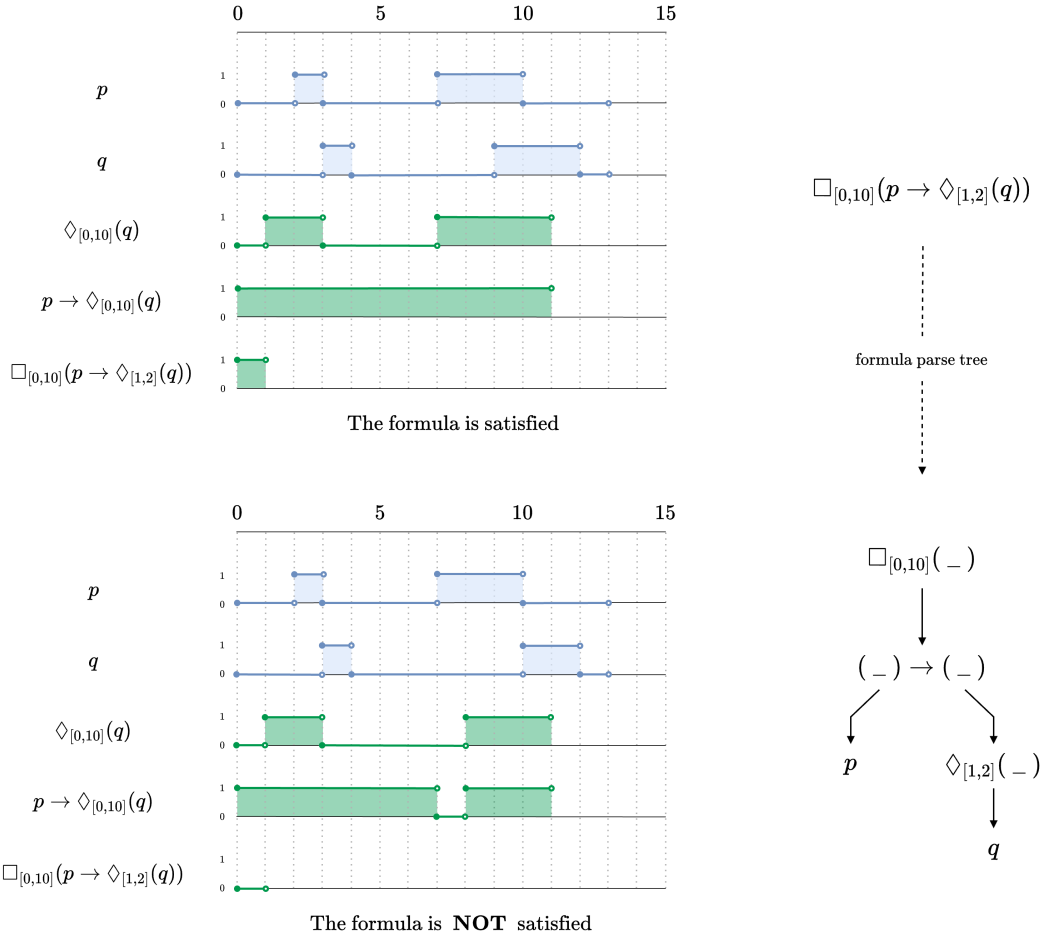


Figure 2.7: *Monitoring process for $\varphi = \Box_{[0,10]}(p \to \Diamond_{[1,2]}q)$. The signals p and q are processed to compute $\chi(\varphi, t)$ using interval shifting.*

2.3.3 Quantitative monitoring

Quantitative monitoring extends the Boolean semantics of STL by providing a *robustness estimate* $\rho(\varphi, w, t)$ that measures how robustly a signal w satisfies or violates an STL formula φ at time t. This robustness estimate is a real-valued function that not only indicates satisfaction ($\rho(\varphi, w, t) > 0$) or violation ($\rho(\varphi, w, t) < 0$) but also quantifies the degree of satisfaction or violation.

The quantitative semantics of STL have previously been defined, providing a recursive computation of $\rho(\varphi, w, t)$ based on the structure of the formula φ and the signal w. However, direct computation of the robustness estimate from its recursive definition can be computationally intensive, especially for significant signals or complex formulas. To address this, efficient algorithms have been developed, such as those described in [95], which significantly reduce the computational cost of evaluating STL robustness, particularly the *Sliding Window Algorithm* used for the temporal operators.

Before presenting the algorithm, we recall the definition of a finitely piecewise-linear and continuous (PLCF) signal, as introduced by Donzé [96], for clarity:

Definition 26 (**Continuous Piecewise-Linear Signal**)
A signal y is said to be finitely piecewise-linear and continuous *(abbreviated as PLCF) if there exists a finite sequence of time points $(t_i)_{i=0}^{n_y}$ such that:*

- *The domain of definition of y is the interval $\left[t_0, t_{n_y}\right)$.*
- *For all $i < n_y$, the signal y is continuous at t_i and affine (ie linear) at the interval $[t_i, t_{i+1})$.*

The sequence $(t_i)_{i=0}^{n_y}$ is referred to as the time sequence *of the signal y.*

Evaluating Temporal Operators Using a Sliding Window

Temporal operators can become computationally expensive to monitor, especially for long signals or complex formulas. To efficiently evaluate the *eventually* operator $\Diamond_{[a,b]}\varphi$, Donzé et al. [96] introduce an algorithm inspired by Lemire's optimal streaming algorithm [162] for computing the minimum or maximum of a numerical sequence over a sliding window. This technique significantly reduces computational overhead, ensuring the monitoring process remains feasible even for large datasets.

The core idea of the algorithm is to shift the time window across the signal and maintain a set of candidate points where the maximum value could occur, updating this set efficiently as the window moves forward. Specifically, the $\Diamond_{[a,b]}$ operator computes the supremum (maximum value) of the signal within the time interval $[t + a, t + b]$ for each time t.

Algorithm Overview

The algorithm assumes that the signal $y(t)$ is *finitely piecewise linear and continuous* (PLCF) and computes the robustness signal $z(t)$ of $\Diamond_{[a,b]}$ by processing the signal in an incremental, sliding window manner.

Input

- **Signal** $y(t) = \{(t_0, y_0), (t_1, y_1), \ldots, (t_N, y_N) \mid t_i \in \mathbb{R}^+ \wedge y_i \in \mathbb{R}\}$: A real-valued signal represented as a sequence of time-stamped samples, where $y_i = (t_i, y_i)$ is the value of the signal at time t_i, and the signal is piecewise linear between consecutive samples.

- **Interval bounds** $[a, b]$: Non-negative real numbers $a, b \in \mathbb{R}$, with $0 \leqslant a < b$, specifying the time interval $[a, b]$ within which we want to compute the supremum of the signal.

1. **Initialise:** Shifting the Signal

   Step 1: First, the signal $y(t)$ is *shifted* to create two signals $y_a(t)$ and $y_b(t)$:
   - $y_a(t) = y(t + a)$ shifts y left by a.
   - $y_b(t) = y(t + b)$ shifts y left by b.

   Shifting serves to align $[t + a, t + b]$ with the current time t.

   Step 2: The pointwise maximum of the two shifted signals y_a and y_b is then computed:

   $$y'(t) = \max\left(y_a(t), y_b(t)\right)$$

   This auxiliary signal $y'(t)$ captures the maximum between the two shifted signals, representing the intermediate candidate signal for the supremum calculation.

   It's worth noticing that computing the maximum corresponds to evaluating the robustness of the "or" operation between two subformulae. In temporal logic, the robustness of the disjunction $\varphi \vee \psi$ at time t is given by:

   $$\rho(\varphi \vee \psi, w, t) = \max\{\rho(\varphi, w, t), \rho(\psi, w, t)\}$$

   This maximum reflects that the eventuality operator $\Diamond$ focuses on the highest robustness value over the time interval.

2. **Main Loop:** Iterating through the Signal

   Step 1: Time Update: At each iteration, the current time t is updated based on the minimum of two possible events:

   $$t = \min(t_{M.front} - a, t_{i+1} - b)$$

   This ensures that the window moves forward either when a new sample enters the window (i.e., t_{i+1}) or when the earliest candidate $t_{M.front}$ exits the window.

   Step 2: Managing Candidates in the Sliding Window: At each iteration of the algorithm, we must update the set of candidates for the maximum value within the current window. This process involves two key operations: removing candidates that have exited the window and adding new candidates as they enter. These two operations are mutually exclusive at each time step, depending on whether a candidate exits or enters the window.

   Step 2.1: Removing Old Candidates (t correspond to $t_{M.front} - a$):

   If $t = t_{M.front} - a$, the earliest candidate $t_{M.front}$ has exited the window. Remove this candidate from the set M:

   $$M \leftarrow M \backslash \{t_{M.front}\}$$

   Then, update the start time s of the current segment to the new t:

   $$s = t$$

Step 2.2: Adding New Candidates:

If $t = t_{i+1} - b$, a new sample t_{i+1} has entered the window. Compare it with the current candidates in M, removing any smaller than $y(t_{i+1} - b)$. After this, add $i + 1$ to M:

$$M \leftarrow M \cup \{i + 1\}$$

Then, increment the index i:

$$i = i + 1$$

The steps just described are illustrated in Example 2.3.1

Step 3: Compute the Robustness Signal: If $s \geqslant t_0$, compute the robustness signal $z(t)$ over the time interval $[s, t)$ by taking the pointwise maximum between the shifted signals $y_a(t)$, $y_b(t)$, and a constant signal representing the current maximum value $y(t_{M.front})$:

$$z_{\restriction [s,t)} \leftarrow \max\left(y'_{\restriction [s,t)}, \overline{y(t_{M.front})}\right)$$

The notation $\restriction [s, t)$ represents the restriction of a signal to the time interval $[s, t)$. For example, $z_{\restriction [s,t)}$ means we are considering the robustness signal z only over the time interval from s (inclusive) to t (exclusive). This notation is particularly important in the algorithm as we compute the robustness values piece by piece, over consecutive time intervals as the window slides.

3. **Final Output:** Once the loop terminates, the algorithm returns the computed robustness signal $z(t)$, which contains the supremum of $y(t')$ over each interval $[t + a, t + b]$ for all times t. The final robustness signal $z(t)$ gives the degree of satisfaction of the $\Diamond_{[a,b]}$ formula.

For more details, refer to Algorithm 1 and the Example 2.3.1

As we established before, we will treat signals as Right-Closed Constant Piecewise Functions (RCPF) (Definition 7). This representation requires a slight modification to the previous algorithm [89]. While the original algorithm from [95] assumes that signals are piecewise linear and continuous (PLCF), our RCPF representation means that the signal is constant within each interval rather than linear, the intervals are of the form $[a_i, a_{i+1})$ for all but the last interval, which is of the form $[a_{n-1}, a_n]$, and the signal values change only at discrete time points.

This simplifies certain aspects of the computation as we don't need to consider interpolation between points, but it requires careful handling of interval endpoints. The modified algorithm (Algorithm 2) reflects these changes in several ways. When shifting signals ($y_a(t)$ and $y_b(t)$), we maintain the right-closed property of the intervals. The computation of the auxiliary signal $y'(t)$ involves taking the maximum of right-closed piecewise constant functions, and the sliding window operations preserve the RCPF structure of the signals. Consequently, the resulting robustness signal $z(t)$ is a right-closed piecewise constant function. The algorithm's accuracy is maintained because the RCPF signals have well-defined values at all points, including the interval endpoints, which is crucial for robustness computation.

Algorithm 1 Efficient Computation of Robustness for $\Diamond_{[a,b]}\varphi$ for continuous piecewise-linear signal

Input:

$y(t)$	Robustness signal of subformula φ, defined over time domain T
a, b	Time bounds defining the interval $[a, b]$ with $0 \leqslant a \leqslant b$
$\{t_i\}_{i=0}^{N}$	Time points where $y(t)$ is defined (sorted in increasing order)

Output:

$z(t)$: Robustness signal of $\Diamond_{[a,b]}\varphi$ over time domain T

// Initialise shifted signals
$y_a(t) \leftarrow y(t + a)$
$y_b(t) \leftarrow y(t + b)$
// Compute auxiliary signal
$y'(t) \leftarrow \max\left(y_a(t),\ y_b(t)\right)$
// Initialise deque to store indices of candidate maxima
$M \leftarrow \varnothing$
// Initialise variables
$i \leftarrow 0$ // index, refers to the sample being analysed
$t \leftarrow t_0 - b$ // time value
$s \leftarrow t$ // time value used to store the beginning of the window whenever the first element is removed
while $(\ t + a \ \leqslant\ t_N\ )$ **do**
  // Calculate the time of the next event encountered
  $t_{next} \leftarrow \min\left(\ t_{M.front} - a\ ,\ t_{i+1} - b\right)$
  // the next event encountered is the first sample IN the window comes out of it
  if $(\ t_{next}\ =\ t_{M.front} - a\ )$ **then**
    $M \leftarrow M \backslash \{\ M.front\ \}$ // remove the first element of the window
    $s \leftarrow t_{next}$
  // the next event encountered is the first sample OUT of the window
  if $(t_{next} = t_{i+1} - b)$ **then**
    // remove all the samples in the window that are smaller than the new sample
    while $(M \neq \varnothing)$ **and** $(\ y(t_{i+1} - b) \geqslant y(t_{M.last})\ )$ **do**
      $M \leftarrow M \backslash \{\ M.back\ \}$
    $M \leftarrow M \cup \{i + 1\}$
    $i \leftarrow i + 1$
  // The condition $s \geqslant t_0$ ensures that the sliding window is fully within the valid time domain. Initially, the window may cover time intervals before t_0.
  if $(s \geqslant t_0)$ **then**
    $z_{1[s,t)} \leftarrow \max\left(y'_{1[s,t)}, \overline{y(t_{M.\text{front}})}\right)$
  $t \leftarrow t_{\text{next}}$ // Update time
return $z(t)$

Efficiency Considerations The Sliding Window Algorithm significantly reduces the computational complexity of evaluating STL formulas. Direct evaluation of the robustness of temporal operators like $\Diamond_{[a,b]}$ or $\mathbf{U}_{[a,b]}$ could, in a naive implementation, require recalculating the robustness over the entire signal for each time step t. This would lead to a time complexity of $O(n^2)$, where n is the number of samples in the signal.

To overcome this, the Sliding Window Algorithm avoids repeated recalculations by

Algorithm 2 Efficient Computation of Robustness for $\Diamond_{[a,b]}\varphi$ for Right-Closed Constant Piecewise

Input:

$y(t)$	Robustness signal of subformula φ, defined over time domain T
a, b	Time bounds defining the interval $[a, b]$ with $0 \leqslant a \leqslant b$
$\{t_i\}_{i=0}^{N}$	Time points where $y(t)$ is defined (sorted in increasing order)

Output:

$z(t)$: Robustness signal of $\Diamond_{[a,b]}\varphi$ over time domain T

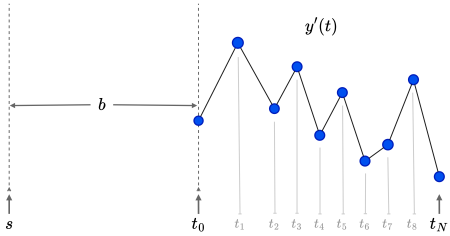
```
// Initialise shifted signals
y_a(t) ← y(t + a)
y_b(t) ← y(t + b)

y'(t) ← max (y_a(t), y_b(t))
M ← ∅
i ← 0
t ← t_0 − b
s ← t while ( t + a ⩽ t_N ) do
    if (M ≠ ∅) then
        t ← min ( t_{M.front} − a , t_{i+1} − b)
    else
        t ← t_{i+1} − b
    if t = t_{i+1} − b then
        while (M ≠ ∅) and ( y(t_{i+1} − b) ⩾ y(t_{M.last}) ) do
            M ← M\{ M.back }
        M ← M ∪ {i + 1}
        i ← i + 1
    else
        if (s ⩾ t_0) then
            z(s) ← y(t_{M.front})
        else
            z(t_0) ← y(t_{M.front})
        M ← M\{ M.front }
        s ← t
return z(t)
```

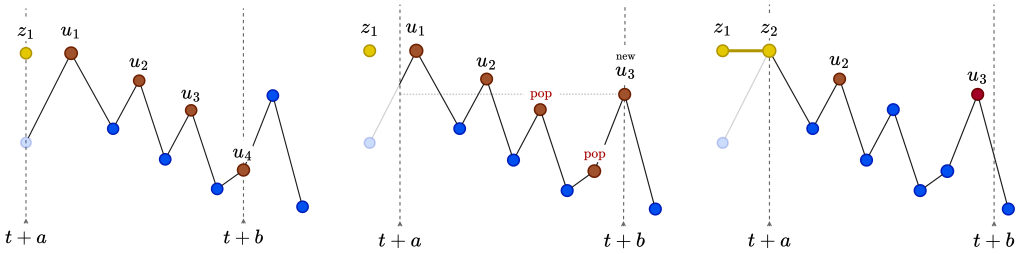
maintaining an efficient data structure (a deque) that tracks candidate points for the maximum robustness over the interval $[t + a, t + b]$. The algorithm slides this window across the signal, updating the robustness values incrementally as new samples enter and old samples exit the window.

For each time step, key operations (insertion, removal, and maximum calculation) are performed in constant time $O(1)$, ensuring that each sample is processed exactly once as the window slides. Consequently, the entire algorithm runs in linear time $O(n)$ concerning the length of the signal. This is because the sliding window ensures that each signal element is added to and removed from the window only once during the entire execution.

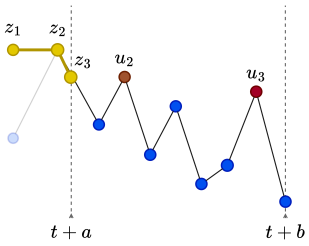
Example 2.3.1 (Sliding Window algorithm example)
This example illustrates the sliding window algorithm to compute the robustness of the eventually operator$\Diamond_{[a,b]}$. We present a visualisation of the process, from initialisation through various stages of window movement, culminating in the final robustness computation.



During initialisation, the auxiliary signal $y'(t)$ is evaluated as the point-wise maximum of the shifted signals $y_a(t)$ and $y_b(t)$. The window is placed in its starting position, ready to process the signal to compute the robustness signal $z(t)$.



The figures above illustrate three key events within a single iteration of the sliding window algorithm: **Step 1** shows the window state after the first point z_1 has been evaluated as a constant maximum (from the previous window exit of the first candidate). In **Step 2**, a new candidate u_3 enters at $t + b$, causing the currents u_3 and u_4 to be removed (popped) as they are inferior to the new candidate. **Step 3** shows when the next event occurs at $t + a$, where the front element exits the window, leading to the evaluation of the robustness signal (shown as z_1 and z_2)



In this last figure, we show the robustness signal computation $z(t)$ for the eventually operator $\Diamond_{[a,b]}$. Two evaluations are shown: (1) between z_1 and z_2, where the signal is constant due to the front element of M exiting the window, and (2) Between z_2 and z_3, where the signal follows $y'(t)$.

Rewriting the *Until* Operator The *until* operator $\varphi \mathbf{U}_{[a,b]} \psi$, which specifies that φ must hold at least until ψ becomes true within the interval $[t+a, t+b]$, can be computationally challenging to evaluate directly. To overcome this, *until* operator can be rewritten in terms of the *eventually* operator, enabling efficient monitoring algorithm previously discussed.

The key idea is to decompose the bounded *until* operator into simpler temporal components:

$$\varphi \mathbf{U}_{[a,b]} \psi \sim \Diamond_{[a,b]} \psi \wedge \varphi \mathbf{U}_{[a,+\infty)} \psi$$

This equivalence states that the formula $\varphi \mathbf{U}_{[a,b]} \psi$ can be rewritten as a conjunction of two parts:

- $\Diamond_{[a,b]} \psi$, which checks if ψ holds within the time interval $[t + a, t + b]$.
- $\varphi \mathbf{U}_{[a,+\infty)} \psi$, which checks that φ holds until ψ becomes true at some point in the future (without an upper bound on time).

For the unbounded case $\varphi \mathbf{U}_{[a,+\infty)} \psi$, the following equivalence holds:

$$\varphi \mathbf{U}_{[a,+\infty)} \psi \sim \square_{[0,a]} (\varphi \mathbf{U} \psi)$$

This means that the unbounded *until* operator can be evaluated by first checking that φ holds globally over the interval $[0, a]$, and then evaluating the simpler unbounded *until* operator $\varphi \mathbf{U} \psi$.

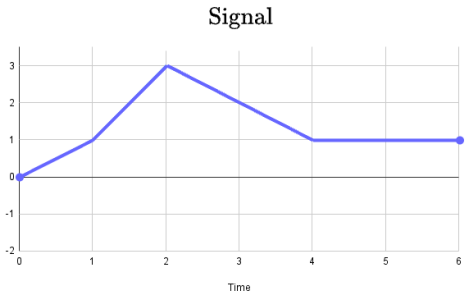
By expressing the *until* operator in terms of the *eventually* and *globally* operators, both of which can be efficiently evaluated using the sliding-window technique, the complexity of monitoring this temporal operator is significantly reduced.

Evaluating Other Temporal Operators In addition to the *until* operator, other temporal operators such as *globally* $\square_{[a,b]} \varphi$ are also derived from the *eventually* operator. Specifically, the *globally* operator can be understood as the negation of *eventually*:

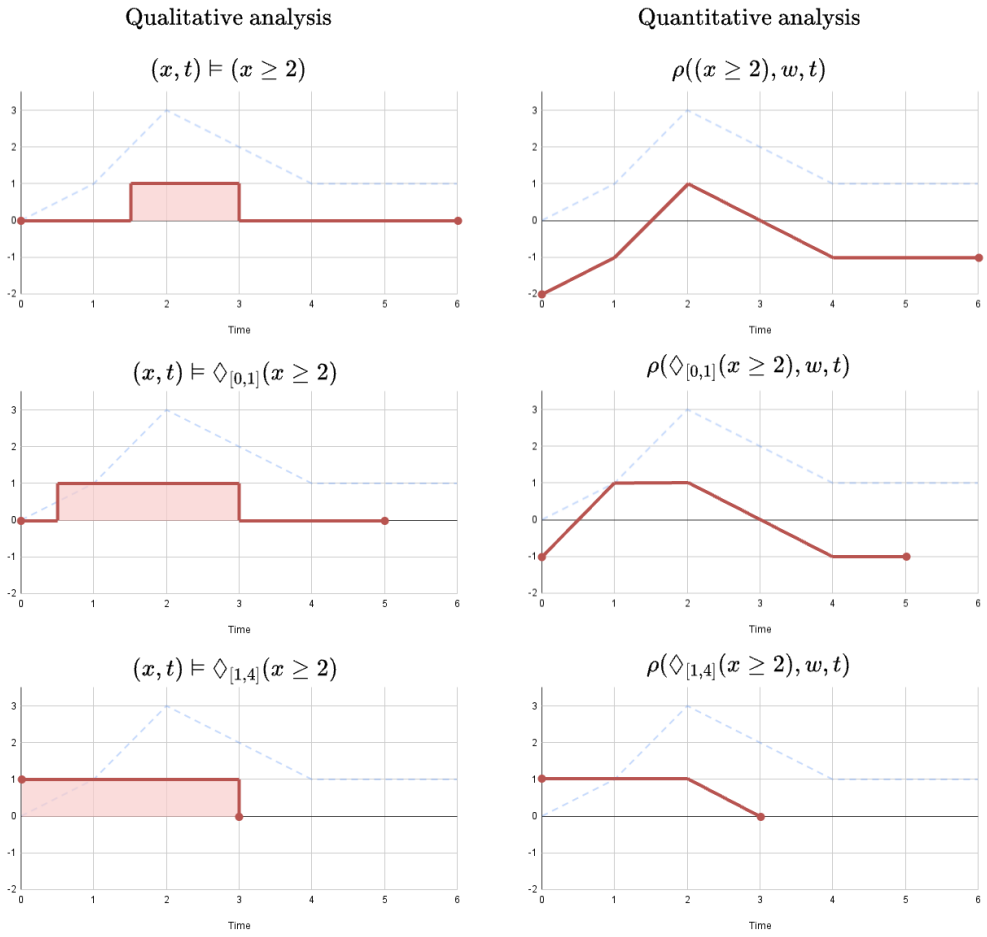
$$\square_{[a,b]} \varphi = \neg \Diamond_{[a,b]} \neg \varphi$$

Thus, the same sliding window technique used for efficiently computing the *eventually* operator can be applied to calculate the *globally* operator.

Example 2.3.2 (Qualitative and Quantitative evaluation)
Let w be the trace with the signal x, defined over $[0, 8]$ and represented by the sequence:
$(t_i, x(t_i), dx(t_i))_{i<4} := \{(0, 0, +1), (1, 1, +2), (2, 3, -1), (4, 1, 0)\}$.



Note that, for simplicity, the trace w in this example consists of a single signal x, although it could be composed of multiple distinct signals. Below, qualitative and quantiative analyses of the signal are shown.



2.4 Analytical Analysis

Before delving into simulation-based verification methods, we briefly discuss the class of analytical techniques available for solving quantitative properties of stochastic models 2.2. These methods rely on the ability to manipulate the full transition structure of the model, and are therefore only feasible for systems of moderate size. While the models considered in this thesis typically go beyond this threshold, the analytical foundations remain a valuable reference for understanding the semantics of simpler models, validating simulation techniques, and illustrating the theoretical limits of tractability. In particular, this section focuses on three fundamental types of stochastic models: DTMCs, CTMCs, and SHA. These formalisms form the theoretical backbone of the more advanced, high-level models treated in this thesis.

Analytical Solution Methods for DTMCs

As introduced in Section 2.2, we consider a DTMC $\mathcal{M} = (\mathcal{S}, \mathbf{P}, AP, L, s_0)$. Here, we focus on key quantitative properties such as reachability probability, expected first passage time, and steady-state distribution.

Reachability Probability. Given a non-empty set of target states $T \subseteq \mathcal{S}$, the reachability probability from a state $s \in \mathcal{S}$ is the probability that a path initiated in s will eventually enter T. This concept is formally defined below.

Definition 27 (Reachability Probability for DTMCs)
Let $\mathcal{M} = (\mathcal{S}, \mathbf{P}, AP, L, s_0)$ be a DTMC and let $T \subseteq \mathcal{S}$ be a non-empty set of target states. The reachability probability *function $r : \mathcal{S} \to [0, 1]$ is defined as:*

$$r(s) = \mathbb{P}_s(\Diamond T) = \mathbb{P}_s(\exists n \geqslant 0 \text{ such that } s_n \in T),$$

where $\{s_n\}_{n \geqslant 0}$ is the sequence of states visited starting from s. For $s \notin T$, the function r satisfies the linear system:

$$r(s) = \sum_{s' \in \mathcal{S}} \mathbf{P}(s, s') \cdot r(s'),$$

with boundary conditions $r(s) = 1$ for all $s \in T$.

The system of linear equations can be solved using standard numerical methods such as Gaussian elimination, but more efficient iterative methods are typically preferred in practice, including value iteration, Jacobi iteration, or Gauss-Seidel methods [197].

Expected First Passage Time. The expected number of steps required to reach a state in T for the first time is defined as follows.

Definition 28 (Expected First Passage Time for DTMCs)
Let $\mathcal{M} = (\mathcal{S}, \mathbf{P}, AP, L, s_0)$ *be a DTMC and* $T \subseteq \mathcal{S}$ *a target set. The* expected first passage time *function* $\mu : \mathcal{S} \to \mathbb{R}_{\geqslant 0}$ *is defined by:*

$$\mu(s) = \mathbb{E}_s[\tau_T],$$

where $\tau_T = \inf\{n \geqslant 0 \mid s_n \in T\}$. *The function* μ *satisfies:*

$$\mu(s) = 1 + \sum_{s' \in \mathcal{S} \setminus T} \mathbf{P}(s, s') \cdot \mu(s'),$$

for $s \notin T$, *and* $\mu(s) = 0$ *for* $s \in T$.

This metric is particularly important for analyzing systems where time-to-completion or time-to-failure is a critical performance measure. When T is unreachable from some states, the expected first passage time becomes infinite, and special techniques like bounded reachability may be required [144].

Steady-State Distribution. For Markov chains that are irreducible (all states can be reached from any other state) and aperiodic (the chain can return to a state at irregular times), the probability distribution of states converges to a unique limiting distribution as time approaches infinity. This steady-state (or stationary) distribution characterizes the long-term behavior of the system, independent of the initial state.

Definition 29 (Steady-State Distribution for DTMCs)
The steady-state distribution $\pi : \mathcal{S} \to [0, 1]$ *satisfies:*

$$\pi(s') = \sum_{s \in \mathcal{S}} \pi(s) \cdot \mathbf{P}(s, s'), \quad \forall s' \in \mathcal{S}, \quad \textit{and} \quad \sum_{s \in \mathcal{S}} \pi(s) = 1.$$

This system can be solved via matrix iteration methods. The convergence rate depends on the spectral gap of $\mathbf{P}$, which is determined by the difference between the largest and second-largest eigenvalues of the transition matrix [41]. For large systems, techniques like power iteration or Krylov subspace methods are commonly employed [198].

Analytical Solution Methods for CTMCs

As introduced in Definition 15, we consider a CTMC $\mathcal{M} = (\mathcal{S}, \mathbf{Q}, AP, L, s_0)$. Unlike DTMCs, where state transitions occur at discrete time steps, CTMCs model systems where state changes occur in continuous time, governed by exponential distributions. The infinitesimal generator matrix $\mathbf{Q}$ captures the rates at which the system transitions between states, with diagonal elements $\mathbf{Q}(s, s) = -\sum_{s' \neq s} \mathbf{Q}(s, s')$ representing the rate at which the system leaves state s.

In models with a large state space, the generator matrix $\mathbf{Q}$ can become prohibitively large to store explicitly. To address this challenge, symbolic representations such

as Multi-Terminal Binary Decision Diagrams (MTBDDs) are often employed. An MTBDD is a graph-based data structure that extends Binary Decision Diagrams (BDDs) by allowing terminal nodes to store real-valued outputs, rather than just Boolean values. This makes MTBDDs particularly suitable for encoding functions from state variables to transition rates. By exploiting regularities and sharing common substructures, MTBDDs provide a compact representation of large and sparse matrices. Analytical operations such as matrix-vector multiplication, required for transient and steady-state analysis, can then be performed directly on the symbolic structure, enabling the analysis of models that would be infeasible with explicit representations.

Transient Analysis. In many applications, we are interested in the probability distribution over states at a specific time point. Transient analysis allows us to compute this time-dependent behavior of the CTMC. The state probability vector $\pi(t) : \mathcal{S} \to [0, 1]$ evolves according to the Kolmogorov forward equations:

$$\frac{d\pi(t)}{dt} = \pi(t) \cdot \mathbf{Q}, \quad \pi(0) = \delta_{s_0},$$

where δ_{s_0} is the Dirac distribution at the initial state. The solution can be obtained via uniformization or numerical integration of the matrix exponential $e^{t\mathbf{Q}}$.

The technique of uniformization [116, 139] is particularly effective for computing transient probabilities. It transforms the CTMC into a discrete-time Markov chain (DTMC) by introducing self-loops, then weights the resulting state probabilities with Poisson probabilities. Specifically, if $\lambda = \max_{s\in\mathcal{S}} |Q(s, s)|$, then:

$$\pi(t) = \sum_{n=0}^{\infty} e^{-\lambda t} \frac{(\lambda t)^n}{n!} \cdot \delta_{s_0} \cdot \mathbf{P}^n$$

where $\mathbf{P} = \mathbf{I} + \frac{1}{\lambda}\mathbf{Q}$ is the transition matrix of the uniformized DTMC. In practice, the infinite sum is truncated based on error bounds derived from the Poisson tail [108].

Reachability Probability. A fundamental question in model checking is whether the system ever reaches a set of target states. For CTMCs, this is formalized as the probability of eventually reaching a set of states, regardless of the time taken.

Definition 30 (Reachability Probability for CTMCs)
Let $T \subseteq \mathcal{S}$ be a set of absorbing states. The reachability probability $r : \mathcal{S} \to [0, 1]$ *is defined as:*

$$r(s) = \mathbb{P}_s(\Diamond T) = \mathbb{P}_s(\exists t \geqslant 0 \text{ such that } s(t) \in T).$$

The function r satisfies:

$$\sum_{s'\in\mathcal{S}} \mathbf{Q}(s, s') \cdot r(s') = 0, \quad \text{for } s \notin T,$$

with $r(s) = 1$ for $s \in T$.

This computation can be reduced to solving a linear system by transforming the CTMC into an embedded DTMC [18]. This reduction preserves the reachability probabilities while simplifying the computational approach.

Expected First Passage Time. In reliability and performance analysis, it is often crucial to determine how long it takes for the system to reach a certain state or condition. The expected first passage time quantifies this duration.

Definition 31 (Expected First Passage Time for CTMCs)
Let $T \subseteq \mathcal{S}$ be a set of absorbing states. The expected first passage time $\mu : \mathcal{S} \to \mathbb{R}_{\geqslant 0}$ *satisfies:*

$$\mu(s) = \mathbb{E}_s[\tau_T],$$

where $\tau_T = \inf\{t \geqslant 0 \mid s(t) \in T\}$. For $s \notin T$, it holds that:

$$-\sum_{s' \in \mathcal{S}} \mathbf{Q}(s, s') \cdot \mu(s') = 1,$$

with $\mu(s) = 0$ for $s \in T$.

This system of equations captures the mean time until absorption in the set T. When T is not reachable from all states, special techniques are required to handle potentially infinite expected times [155].

Steady-State Distribution. For ergodic CTMCs (irreducible and positive recurrent), the probability distribution converges to a unique steady-state distribution independent of the initial state. This distribution represents the long-run proportion of time the system spends in each state and is central to performance evaluation.

Definition 32 (Steady-State Distribution for CTMCs)
The steady-state distribution $\pi : \mathcal{S} \to [0, 1]$ *satisfies:*

$$\pi \cdot \mathbf{Q} = 0, \quad \sum_{s \in \mathcal{S}} \pi(s) = 1.$$

Such π exists and is unique when the CTMC is irreducible. The linear system can be solved using sparse matrix techniques such as the preconditioned conjugate gradient method or iterative aggregation/disaggregation techniques [47, 197].

For ergodic CTMCs, the steady-state distribution provides a long-term characterization of the system's behavior, allowing for the computation of long-run average rewards and other performance metrics [124].

Uniformization and the Fox–Glynn Method. A widely adopted technique for computing the transient solution of a CTMC is *uniformization*, also known as the *randomization method*. This approach transforms the continuous-time dynamics into an equivalent discrete-time formulation by introducing a uniform rate λ such that $\lambda \geqslant \max_{s \in \mathcal{S}} |\mathbf{Q}(s, s)|$.

Defining the derived transition matrix $\mathbf{P} = \mathbf{I} + \frac{1}{\lambda}\mathbf{Q}$, the state probability vector at time t is then expressed as:

$$\pi(t) = \sum_{n=0}^{\infty} e^{-\lambda t} \frac{(\lambda t)^n}{n!} \cdot \pi(0) \cdot \mathbf{P}^n.$$

This series expansion enables the computation of transient distributions using repeated matrix-vector multiplications, weighted by the Poisson distribution with mean λt.

A seminal contribution to the practical implementation of this method is due to Fox and Glynn [108], who provided an efficient truncation strategy based on tail bounds of the Poisson distribution, enabling accurate and stable computation of the infinite sum with finite effort. Their method is purely numerical and deterministic—despite exploiting probabilistic identities—and is widely considered a cornerstone of the analytical toolkit for transient analysis of CTMCs. It is important to note that this technique, while probabilistically inspired, is not simulation-based in the sense of Statistical Model Checking: it does not rely on random sampling, but on exact numerical evaluation.

Analytical Solution Methods for SHA

As introduced in Section 2.2, we have explored a hierarchy of stochastic models with increasing complexity, from the purely stochastic DTMCs and CTMCs to more sophisticated formalisms. Referring to Figure 2.2, SHA occupy a position at the apex of this hierarchy, incorporating all elements of lower-level models while adding continuous dynamics governed by differential equations. SHA thus represent the most expressive and complex modeling formalism in our framework, capable of capturing both probabilistic transitions and continuous evolution of state variables over time.

This position at the top of the model hierarchy comes with significant analytical challenges. While DTMCs and CTMCs admit efficient solution methods for various properties as discussed in previous sections, the integration of continuous dynamics with discrete probabilistic jumps in SHA creates fundamental obstacles for analytical approaches.

The complexity of SHA stems from the interleaving of continuous dynamics with discrete, probabilistic state changes. The absence of a global Markovian structure, combined with:

- nonlinear dynamics that preclude closed-form solutions to differential equations [129];
- time-dependent guards and resets that create path-dependent behavior [6];
- stochastic delays beyond exponential distributions, which violate the memoryless property required for Markovian analysis [52];
- the undecidability of reachability questions even for relatively simple hybrid systems [130];

renders standard matrix-based approaches inapplicable.

Known Partial Approaches. Despite the general intractability, researchers have developed specialized techniques for analyzing subclasses of hybrid systems or for obtaining approximate results. Some subclasses of hybrid models (e.g., piecewise linear systems, timed automata with uniform rates) admit symbolic model checking or abstraction-based analysis. However, even in such cases, solutions are often approximate and rely on discretization or reachability over-approximation [58, 109].

In particular, approaches like:

- Abstraction refinement techniques that create finite-state approximations of hybrid systems [67];
- Barrier certificates that provide safety guarantees without explicitly computing reachable sets [187];
- Statistical model checking that uses simulation and hypothesis testing to evaluate probabilistic properties [211];

can provide partial answers but are either computationally expensive or provide probabilistic rather than definitive guarantees.

Therefore, analytical verification of general SHA is considered undecidable in most realistic settings [6].

2.4.1 Limitations of Analytical Methods

Although analytical techniques for DTMCs, CTMCs, and (to a lesser extent) SHA offer mathematically precise solutions, their applicability is constrained by several structural and computational limitations.

Limitations of Analytical Methods

The primary bottleneck lies in the size and complexity of the state space: even sparse matrices $\mathbf{P}$ or $\mathbf{Q}$ become infeasible to handle for models with more than 10^5 to 10^6 states [153]. Further complications arise in the presence of:

- **Nondeterminism**, requiring strategy synthesis and often increasing complexity to NP-hard or PSPACE-complete [57, 69].
- **Parametric uncertainty**, leading to symbolic expressions with exponential blow-up [77, 119].
- **Compositional models**, where parallelism causes state space explosion [154].

For SHA models, the integration of continuous dynamics with stochastic jumps leads to undecidability in even basic reachability queries [6, 130], precluding general analytical solutions.

Simulation-Based Analysis as a Practical Alternative

Simulation-based methods become essential when analytical techniques are infeasible due to model size, hybrid dynamics, or structural irregularities [68, 78, 185]. They offer several advantages:

- **Generality**: Applicable to models with arbitrary dynamics and distributions [193].
- **Scalability**: Parallelizable and tunable to available computational resources [160, 208].

Nonetheless, simulation methods provide statistical rather than absolute guarantees [196], may struggle with rare events [193], and require additional handling of nondeterminism [128].

Complementarity and Hybrid Use

Rather than being mutually exclusive, analytical and simulation approaches are often most effective when combined. Analytical results can serve as validation points or inform the design of more efficient simulations (e.g., via importance sampling). Conversely, simulations allow exploration of systems beyond the reach of exact methods.

In this thesis, simulation-based analysis is adopted as the primary methodology due to the structural complexity of the models considered, with analytical techniques used for theoretical grounding and selective validation.

2.5 Simulation-Based Statistical Verification

As discussed in Section 2.4.1, the analytical verification of complex stochastic systems is often infeasible due to state space explosion, hybrid dynamics, or undecidability. Simulation-based statistical methods, collectively referred to as Statistical Model Checking (SMC), offer a scalable alternative that enables the estimation of property satisfaction with formal confidence guarantees. This section presents the foundational components of the SMC framework, including Monte Carlo sampling, simulation algorithms, and statistical techniques for verifying temporal properties.

SMC will be widely applied in Chapter 5, where the Sibilla framework leverages statistical model checking to assess temporal properties over simulation traces.

2.5.1 Monte Carlo Methods in SMC

At the core of SMC are Monte Carlo methods, which rely on repeated random sampling to obtain numerical results. In the context of SMC, these methods are employed to generate multiple independent simulation traces of the system under study. The law of large numbers ensures that the statistical estimates converge to the true probabilities as the number of simulations increases. Monte Carlo methods thus provide a foundational tool for approximating the behaviour of complex stochastic systems where analytical solutions are unattainable.

2.5.2 Simulation Algorithms

Monte Carlo methods in this context are used to generate sample paths of the stochastic process under study using simulation. Some well-known simulation algorithms are commonly used in SMC to sample these paths. Some well-known simulation algorithms include the Stochastic Simulation Algorithm (SSA), the Gibson-Bruck method, and Tau-Leaping. These algorithms generate trajectories by balancing efficiency and

accuracy. While some provide exact simulations, others offer approximations that improve performance without significantly compromising accuracy. The choice of simulation algorithm can significantly impact the efficiency and feasibility of the verification process.

Stochastic Simulation Algorithm (SSA)

The Stochastic Simulation Algorithm, also known as Gillespie's algorithm [114], is fundamental in simulating chemical or biochemical systems of reactions. It provides an exact method for simulating the temporal evolution of a well-stirred chemically reacting system.

The basic steps of SSA are the following:

1. Initialise the state and time of the system.
2. Calculate the propensity function for each reaction.
3. Generate two random numbers to determine the next reaction and the time until it occurs.
4. Update the system state and time.
5. Repeat steps 2-4 until the simulation end time is reached.

Although SSA can be computationally expensive for systems with many reactions or molecules.

Gibson-Bruck Method

The Gibson-Bruck method [113], also known as the Next Reaction Method, is an optimised version of SSA. Reduce computational complexity by using a priority queue to store the following reaction times and only updating the propensities of affected reactions.

Key features of the Gibson-Bruck method include:

- Use of a dependency graph to identify affected reactions
- Maintenance of a priority queue for efficient selection of the next reaction
- Reuse of random numbers to reduce the number of random number generations

This method is particularly efficient for systems with many species and reactions.

Tau-Leaping

Tau-leaping [115] is an approximate acceleration technique for i SSA. It advances the simulation by a pre-selected time step τ (tau), during which multiple reaction events can occur.

The tau-leaping method:

1. Choose a τ value that satisfies the leap condition.
2. Approximate the number of occurrences of each reaction during τ using a Poisson distribution.

3. Updates the state of the system based on these approximations.

Tau-leaping can significantly speed up simulations, especially for systems with many molecules, at the cost of introducing some approximation error.

2.5.3 Verification Techniques

With simulation traces generated by these algorithms, SMC employs various techniques to verify system properties. These techniques utilise statistical hypothesis testing and estimation to conclude the system's behaviour with quantified confidence levels.

Reachability Analysis

Reachability analysis in SMC aims to estimate the probability of reaching a certain state or set of states within a given time bound. This is achieved by:

1. **Simulation Execution**: Run multiple independent simulations up to the specified time bound.
2. **State Checking**: Examine each simulation trace to determine whether the target state(s) were reached within the time frame.
3. **Probability Estimation**: Use statistical methods, such as confidence interval calculations or Chernoff bounds, to estimate the reachability probability and provide confidence levels.

Hypothesis Testing

Hypothesis testing in SMC is used to verify whether the probability of a property being true meets a certain threshold. Two common approaches are:

- **Sequential Probability Ratio Test**: Sequential analysis of simulation results to decide between two hypotheses, typically if the probability is above or below a specified threshold. The aim is to decide with a minimal number of samples while controlling error rates.
- **Bayesian Hypothesis Testing**: Use Bayesian inference to update the probability of each hypothesis as new data is observed, allowing for a probabilistic assessment based on prior beliefs and observed evidence.

2.6 Synthesis

Synthesis is generating or refining parameters for a system model to ensure that it meets the specified requirements. In formal verification and system design, synthesis plays a critical role in adjusting the system's parameters or configurations so that the desired properties are satisfied. This process often involves exploring an ample or uncountable space of potential parameter values to identify configurations that lead to correct or optimal system behaviour.

Parameter synthesis emerged as a natural extension to verification problems, where instead of merely checking if a system satisfies certain properties, we seek to determine

the parameter values that guarantee satisfaction. This approach is particularly valuable when designing systems from scratch or refining existing ones to meet changing requirements. While verification answers the question "Does the system work as expected?", synthesis addresses the more constructive query: "How should we build the system to work as expected?"

2.6.1 Definition of the Synthesis Problem

In this section, we formalise the synthesis problem. Consider a parametric model $\mathcal{M}(\Theta)$, where $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$ is a set of real-valued parameters. Each instantiation of Θ corresponds to a specific model configuration. Let φ be a logical specification that the system must satisfy. The synthesis problem can be defined as follows:

Definition 33 (Synthesis Problem)
Given a parametric model $\mathcal{M}(\Theta)$ and a specification φ, the synthesis problem *consists of finding a parameter instantiation Θ such that:*

$$\mathcal{M}(\Theta) \models \varphi.$$

Alternatively, it involves determining the set of all parameter values $\mathcal{H} \subseteq \mathbb{R}^n$ for which the specification φ holds, i.e.,

$$\mathcal{H} = \{\Theta \mid \mathcal{M}(\Theta) \models \varphi\}.$$

This set $\mathcal{H}$ is often referred to as the *feasible parameter space* or the *synthesis region*. It represents the subset of the parameter space where the model satisfies the desired specification.

Several variations of the synthesis problem are considered in the literature [138]:

- **Feasibility Problem:** Determine if there exists at least one parameter instantiation Θ such that $\mathcal{M}(\Theta) \models \varphi$.
- **Optimal Feasibility Problem:** Identify a parameter instantiation Θ that maximises or minimises a specific objective, such as robustness or probability of satisfying φ.
- **Region Verification Problem:** Verify whether all instantiations of parameters within a given region satisfy φ.
- **Partitioning Problem:** Partition the parameter space into subsets where φ holds (accepting) and where it does not (rejecting).
- **Feasibility Problem:** Determine whether there exists a parameter instantiation such that the model satisfies the property:

$$\exists \theta \in \Theta \text{ such that } \mathcal{M}(\theta) \models \varphi.$$

- **Optimal Feasibility Problem:** Identify a parameter instantiation that maximises or minimises an objective function f, such as robustness or satisfaction probability:

$$\theta^* = \arg\max_{\theta \in \Theta} f(\mathcal{M}(\theta), \varphi) \quad \text{or} \quad \arg\min_{\theta \in \Theta} f(\mathcal{M}(\theta), \varphi).$$

- **Region Verification Problem:** Verify whether all parameter instantiations within a given region satisfy the property:

$$\forall \theta \in R, \quad \mathcal{M}(\theta) \models \varphi.$$

- **Partitioning Problem:** Partition the parameter space into regions that satisfy or violate the property:

$$\Theta = R_1 \cup R_2, \quad R_1 \cap R_2 = \varnothing, \quad \begin{cases} \forall \theta \in R_1 : \mathcal{M}(\theta) \models \varphi \\ \forall \theta \in R_2 : \mathcal{M}(\theta) \not\models \varphi \end{cases}$$

- **Threshold Synthesis Problem:** Find parameters such that the probability of satisfying the property exceeds a given threshold λ:

$$\{\theta \in \Theta \mid \Pr[\mathcal{M}(\theta) \models \varphi] \geqslant \lambda\}.$$

- **Max-Min Optimization Problem:** Identify parameters that maximise the minimum satisfaction probability across a set of environment conditions $e \in E$:

$$\theta^* = \arg\max_{\theta \in \Theta} \min_{e \in E} \Pr[\mathcal{M}(\theta, e) \models \varphi].$$

- **Multi-Objective Optimization Problem:** Compute the set of Pareto-optimal parameters that balance multiple objectives $f_1, \ldots, f_k$:

$$\nexists \theta' \in \Theta \text{ such that } \forall i,\ f_i(\theta') \geqslant f_i(\theta) \text{ and } \exists j,\ f_j(\theta') > f_j(\theta).$$

Parameter synthesis is only meaningful when the underlying model is parametrisable, that is, when its structure explicitly depends on parameters whose values can be tuned. A prominent and well-studied example of such models is the class of Parametric Markov Models, which combine stochastic dynamics with symbolic parameter representations. These models are particularly suited to synthesis tasks, as they allow one to reason about how uncertainty and probabilistic outcomes vary across the parameter space [138].

2.6.2 Challenges in Parameter Synthesis

Parameter synthesis faces several significant challenges:

- **Dimensionality:** As the number of parameters increases, the parameter space grows exponentially, leading to the well-known "curse of dimensionality".
- **Non-linearity:** Many practical systems exhibit non-linear relationships between parameters and behaviours, complicating the search for feasible solutions.
- **Discontinuities:** The feasible parameter space $\mathcal{H}$ may be disconnected or contain sudden transitions where small parameter changes lead to qualitatively different behaviours.
- **Computational Complexity:** For many model classes, parameter synthesis problems are undecidable or computationally intractable in the general case.

2.6.3 Methods for Parameter Synthesis

Various approaches have been developed to tackle parameter synthesis problems:

- **Sampling-based Methods:** These methods explore the parameter space by evaluating the model at specific parameter instances, building a statistical approximation of the feasible region. While computationally efficient, they provide no formal guarantees.
- **Constraint-based Methods:** By encoding the synthesis problem as a constraint satisfaction problem, techniques from constraint programming or satisfiability modulo theories (SMT) can be applied to find feasible parameter values.
- **Symbolic Methods:** These approaches represent the feasible parameter space symbolically, often using geometric objects like polyhedra or more general semi-algebraic sets. The analysis is conducted on these symbolic representations rather than on concrete parameter values.
- **Region Refinement:** Starting with an initial region in the parameter space, refinement methods iteratively partition the space into smaller regions, classifying each as either satisfying, violating, or uncertain with respect to the specification φ.

2.6.4 Applications of Parameter Synthesis

Parameter synthesis has found applications in various domains:

- **Systems Biology:** Determining kinetic parameters in biochemical reaction networks that lead to desired cellular behaviours.
- **Robotics:** Finding control parameters that ensure robots complete tasks with specified reliability or performance metrics.
- **Communication Networks:** Tuning protocol parameters to guarantee quality of service properties under varying operating conditions.
- **Cyber-Physical Systems:** Ensuring safety and performance properties in systems where physical dynamics interact with computational elements.

2.7 Collective Adaptive Systems

In this chapter, we have explored various modeling techniques, verification methods, and analysis approaches for both low-level and high-level models. As introduced in the introduction Chapter, our research focuses specifically on Collective Adaptive Systems (CAS), which present unique challenges for quantitative analysis. Having established the theoretical foundations, we now place CAS in context and define their specific characteristics that inform our analytical approaches.

Definition and Characteristics

CAS represent a distinct class of complex systems that require specialized modeling and analysis techniques [135]. While we can use the high-level modeling formalisms

discussed earlier to represent CAS, these systems are defined not by the notation used to model them, but by their inherent properties and behaviors [60, 80].

CAS are characterised by their massive scale, heterogeneity, multi-scale operation, and emergent behaviour, involving numerous interacting components that can include machines, humans, and networks [135]. Due to these complex interactions, CAS can exhibit adaptive and dynamic behaviours, which may be difficult to predict and control. The components within a CAS may operate at varying temporal and spatial scales, with potentially conflicting objectives, in open environments where entities can dynamically join or leave the system.

Specifically, CAS are characterized by:

- **Massive Scale**: CAS typically involve thousands to millions of interacting components, making traditional exhaustive analysis methods computationally intractable.
- **Autonomy**: Individual entities within CAS possess autonomous decision-making capabilities, operating without centralized control and adapting their behavior based on local information [82].
- **Adaptivity**: Both individual components and the system as a whole dynamically adjust behavior in response to changing conditions, evolving their structure and function over time [99].
- **Emergence**: The collective behavior of the system emerges from local interactions, often exhibiting properties and patterns that cannot be easily predicted from the behaviors of individual components.
- **Heterogeneity**: Components may differ in their capabilities, behaviors, and evolution rules, requiring expressive modeling formalisms to capture this diversity [36].
- **Open-Endedness**: CAS boundaries are often fluid, with components joining or leaving the system during execution, making traditional closed-world assumptions inadequate [80, 132].
- **Spatial Embedding**: Many CAS operate within spatial contexts where position, distance, and topology influence interactions and system dynamics [60, 111].

These interrelated characteristics make CAS particularly challenging to analyze, as they combine scale issues, stochastic behavior, spatial dynamics, and emergent properties in ways that traditional verification approaches struggle to address.

Within the category of CAS, systems can exhibit a range of characteristics, from continuous to discrete. Moreover, these systems can manifest as hybrids, integrating both continuous and discrete elements, as seen in Cyber-Physical Systems (CPS) [4]. CPS are becoming prevalent in sectors like power, water, waste, transportation, healthcare, agriculture, and manufacturing [184,207]. The dynamic nature of CAS accommodates the diversity of system types, making it a versatile framework for modelling and understanding collective behaviours that transcend boundaries between the continuous and the discrete.

Application Domains

The concept of CAS spans numerous domains where collective intelligence and adaptive behavior are essential:

- **Biological Systems**: From cellular networks to bacterial colonies and ecosystems, biological systems often exhibit collective adaptive behaviors through decentralized coordination [32, 50].
- **Smart Cities and Infrastructure**: Urban systems involving traffic management, energy distribution, and public services increasingly rely on networks of interconnected sensors, actuators, and decision-making components [39, 177].
- **Swarm Robotics**: Coordination of multiple robotic entities to achieve complex tasks through emergent behaviors without centralized control [44, 163].
- **Socio-Technical Systems**: Systems where human and technological components interact, such as social networks, crowdsourcing platforms, and collaborative work environments [64, 106].
- **Service-Oriented Computing**: Dynamic service composition, discovery, and adaptation in large-scale distributed environments [132, 173].

Analytical Challenges for CAS

The quantitative analysis techniques discussed throughout this chapter face significant challenges when applied to CAS. These characteristics present significant research challenges, especially in effectively modelling and reasoning about the system's behaviour:

- **State Space Explosion**: The combinatorial growth of possible system states makes exhaustive analysis computationally infeasible, necessitating statistical and approximation techniques.
- **Multi-Scale Interactions**: CAS often operate across multiple temporal and spatial scales, requiring models that can bridge micro-behaviors with macro-level phenomena.
- **Stochasticity**: Randomness plays a crucial role in many CAS, influencing both individual behaviors and system-level properties, requiring probabilistic modeling approaches.
- **Environmental Dynamics**: CAS operate within and interact with dynamic environments, creating feedback loops that must be captured in models.
- **Emergent Properties**: The non-trivial relationship between component behaviors and system-level properties requires specialized formalisms and verification techniques.

The quantitative analysis techniques discussed throughout this chapter, particularly the methods for simulation-based verification and statistical approaches, provide a foundation for addressing the challenges presented by CAS.

In Chapter 4, we will explore how different research communities have approached the analysis of CAS, reviewing various tools and methodologies developed to address their unique challenges. Building upon both the theoretical foundations established here and the insights from existing approaches, we will then introduce the Sibilla framework in Chapter 5, which offers a practical and scalable solution for the quantitative analysis and synthesis of CAS.

CHAPTER 3

Surrogate Optimisation

"The purpose of models is not to fit the data but to sharpen the questions."

S. Karlin

Surrogate-based optimisation (SBO) has emerged as a powerful approach to solving complex optimisation problems, mainly when dealing with computationally expensive simulations or when the objective function is noisy or challenging to evaluate. This chapter delves into the fundamental concepts, techniques, and algorithms used in SBO, providing a comprehensive overview of this field.

In engineering and scientific computing, many optimisation problems involve objective functions that are costly to evaluate, either due to time-consuming simulations or expensive physical experiments. Traditional optimisation methods often require numerous function evaluations, making them impractical for such problems. SBO addresses this challenge by constructing and optimising surrogate models, approximations of the actual objective function that are faster and cheaper to evaluate.

The core idea behind SBO is to iteratively refine a surrogate model of the objective function and use this model to guide the optimisation process. This approach allows for efficient design space exploration, potentially reducing the number of expensive function evaluations required to find an optimal solution. SBO has found applications in various fields, including aerospace engineering, structural optimisation, chemical process design, and machine learning hyperparameter tuning.

Throughout this chapter, we will explore the key components of SBO:

- Sampling techniques for efficiently exploring the design space
- Construction and selection of surrogate models
- Optimisation algorithms tailored for surrogate models

- Infill strategies for iteratively improving surrogate models

By the end of this chapter, the reader will have a solid understanding of the principles and techniques used in surrogate-based optimisation, enabling them to apply these methods to their complex optimisation problems.

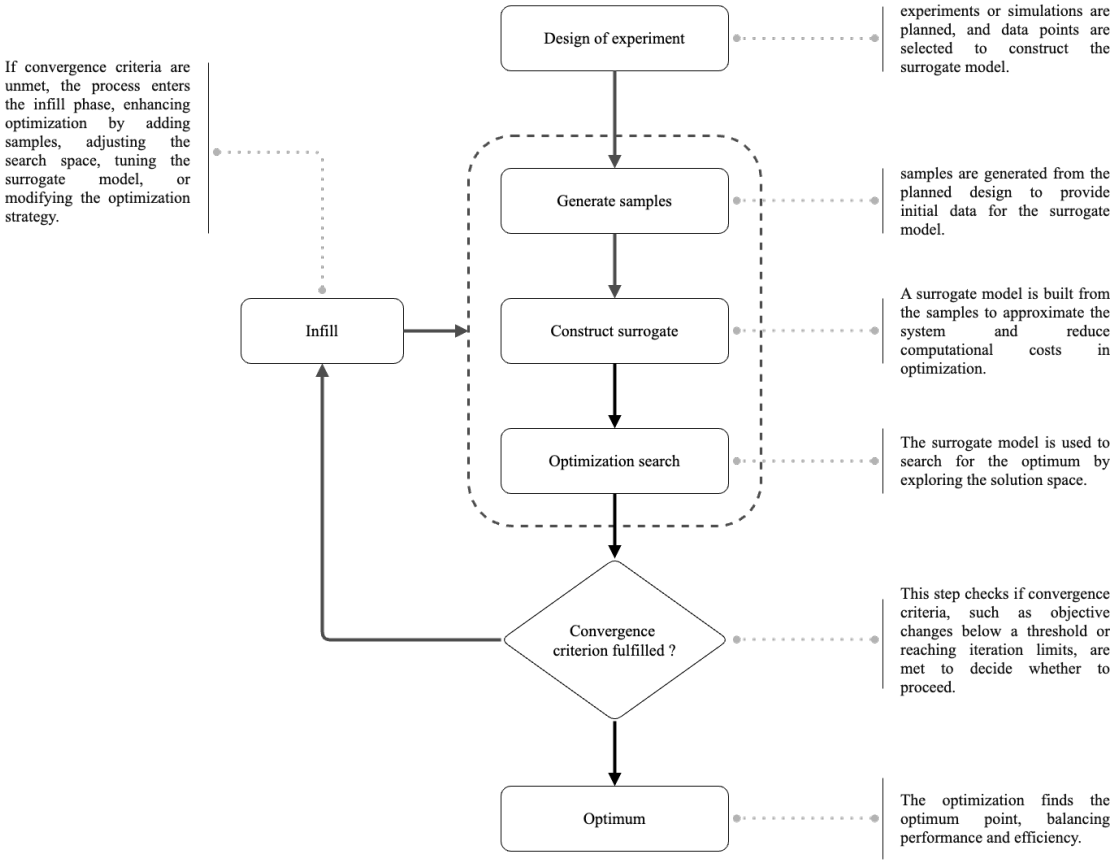


Figure 3.1: *Flowchart of the Surrogate-Based Optimisation (SBO) Process: The diagram illustrates the iterative cycle of SBO, starting with the design of experiments, generating samples, and constructing a surrogate model.*

In Figure 3.1, the flowchart of SBO can be observed.

3.1 Preliminary Concepts and Notation

Before delving into the specifics of surrogate-based optimisation, it is essential to establish a foundation of key concepts and notation. This section introduces the fundamental ideas and mathematical representations used throughout the chapter.

3.1.1 Optimisation Problem Formulation

We consider the general optimisation problem:

$$\begin{aligned} \min_{x \in \mathcal{X}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leqslant 0, \quad i = 1, \ldots, m \\ & h_j(x) = 0, \quad j = 1, \ldots, p \end{aligned} \tag{3.1}$$

where $x \in \mathbb{R}^n$ is the vector of design variables, $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $g_i : \mathbb{R}^n \to \mathbb{R}$ are inequality constraints, and $h_j : \mathbb{R}^n \to \mathbb{R}$ are equality constraints. The set $\mathcal{X}$ represents the design space, typically a hypercube defined by lower and upper bounds on each variable.

In the context of surrogate-based optimisation, we assume that evaluating $f(x)$, $g_i(x)$, or $h_j(x)$ is computationally expensive or time-consuming.

3.1.2 Design Space and Hyperrectangles

In optimisation problems, we often need to define the space where we search for solutions. This design space $\mathcal{X}$ is typically represented as a hyperrectangle, which generalises a rectangle to higher dimensions. Formally, we can define a hyperrectangle as follows:

Definition 34 (Design Space and Hyperrectangle)
Given a parameter vector $\Theta = (\theta_1, ..., \theta_j, ..., \theta_k)$ in $\mathbb{R}^k$, a hyperrectangle $\mathcal{H}$ is defined as the Cartesian product of closed intervals:

$$\mathcal{H} = \prod_{j=1}^{k} [l_j, u_j]$$

Where:

- *l_j is the lower bound for parameter θ_j*
- *u_j is the upper bound for parameter θ_j*
- *$[l_j, u_j]$ represents the closed interval containing all permissible values for θ_j*

The design space $\mathcal{X}$ is typically defined as such a hyperrectangle, representing the feasible region where:

$$\theta_j \in [l_j, u_j] \quad \forall j \in \{1, ..., k\}$$

This formulation provides a structured way to define the boundaries of our optimisation problem. Each dimension of the hyperrectangle corresponds to a design variable, and the bounds l_j and u_j reflect physical constraints, engineering requirements, or other practical limitations on the design parameters.

3.1.3 Surrogate Models

A surrogate model, denoted as $\hat{f}(x)$, is an approximation of the actual objective function $f(x)$. The goal is to construct $\hat{f}(x)$ such that it is: computationally inexpensive

to evaluate, reasonably accurate in approximating $f(x)$ and capable of capturing the global trends of $f(x)$

Common types of surrogate models include:

- Polynomial Response Surfaces
- Radial Basis Functions (RBF)
- Kriging (Gaussian Process Regression)
- Support Vector Regression (SVR)
- Artificial Neural Networks (ANN)

3.1.4 Error Metrics

Various error metrics are used to assess the accuracy of a surrogate model. Let $\hat{f}(x)$ be the surrogate model prediction at point x. Common error metrics include:

- Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - \hat{f}(x_i))^2 \tag{3.2}$$

- Root Mean Squared Error (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(f(x_i) - \hat{f}(x_i))^2} \tag{3.3}$$

- Maximum Absolute Error (MAE):

$$\text{MAE} = \max_{i=1,\ldots,N} |f(x_i) - \hat{f}(x_i)| \tag{3.4}$$

- Coefficient of Determination (R^2):

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(f(x_i) - \hat{f}(x_i))^2}{\sum_{i=1}^{N}(f(x_i) - \bar{f})^2} \tag{3.5}$$

  where $\bar{f}$ is the mean of the observed values.

3.2 Sampling Techniques

Sampling methods are crucial in surrogate-based optimisation to select a set of points within the hyperrectangular design space $\mathcal{H}$ to evaluate the expensive true function. These points form the training data for constructing the surrogate model. Let $\mathcal{D} = \{(x_i, f(x_i))\}_{i=1}^{N}$ denote the set of N sampled points and their corresponding function values, where $x_i \in \mathcal{H}$.

The goal of sampling is to efficiently explore the hyperrectangle $\mathcal{H}$, ensuring a good coverage of the design space while minimising the number of expensive function evaluations. Standard sampling techniques include: Full Factorial Design, Latin Hypercube Sampling, Orthogonal Arrays and Low-discrepancy Sequences (e.g., Sobol, Halton).

3.2.1 Random Sampling

Random sampling is one of the simplest methods for generating sample points within a hyperrectangle. It involves selecting points uniformly at random from the defined search space. While simple, this method can be effective for the initial design space exploration, primarily when little is known about the underlying function.

Algorithm 3 Random Sampling

Given:
$\mathcal{H}$: The hyperrectangle representing the search space
N : Number of samples to generate
Return:
$\mathcal{D}$: Set of N sampled points within $\mathcal{H}$

k : Dimensionality of $\mathcal{H}$
l_j, u_j : Lower and upper bounds for dimension j of $\mathcal{H}$
for $i = 1$ **to** N **do**
  for $j = 1$ **to** k **do**
    $x_{i,j} \sim \text{Uniform}(l_j, u_j)$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{x_i\}$
return $\mathcal{D}$

The Random Sampling algorithm is straightforward:

1. For each required sample point:
   (a) For each dimension of the hyperrectangle $\mathcal{H}$, generate a random value uniformly distributed between the lower and upper bounds of that dimension.
   (b) Combine these random values to form a point in the k-dimensional space.
   (c) Add this point to the set of sampled points $\mathcal{D}$.
2. Return the set of sampled points $\mathcal{D}$.

While simple to implement, random sampling has some limitations. It often fails to ensure uniform search space coverage, particularly when only a few samples are taken. This can result in clusters of points in some regions while other areas remain sparsely sampled. In addition, the quality of the sampling can vary considerably between different attempts, leading to inconsistent results.

Despite these limitations, random sampling can be useful for initial exploration or as a baseline for comparison with more sophisticated sampling methods. It is often used with other techniques or as part of more complex algorithms to introduce randomness and prevent getting stuck in local optima.

3.2.2 Full Factorial Sampling

Full Factorial Sampling, also known as grid sampling, is a systematic approach that places a grid of evenly spaced points over the search space. This method provides complete coverage of the design space but can become computationally expensive for high-dimensional problems due to the exponential growth in the number of sample points.

The Full Factorial Sampling algorithm works as follows:

Algorithm 4 Full Factorial Sampling

Given:
$\mathcal{H}$: The hyperrectangle representing the search space
n : Number of samples per dimension
Return:
$\mathcal{D}$: Set of sampled points within $\mathcal{H}$

k : Dimensionality of $\mathcal{H}$
l_j, u_j : Lower and upper bounds for dimension j of $\mathcal{H}$
for $j = 1$ **to** k **do**
  $\delta_j \leftarrow (u_j - l_j)/(n-1)$; // Step size for dimension j
  $S_j \leftarrow \{l_j + i \cdot \delta_j : i = 0, 1, \ldots, n-1\}$; // Set of points for dimension j
$\mathcal{D} \leftarrow S_1 \times S_2 \times \cdots \times S_k$; // Cartesian product of all dimension sets
return $\mathcal{D}$

1. For each dimension of the hyperrectangle $\mathcal{H}$:
   (a) Calculate the step size δ_j based on the number of samples per dimension and the dimension's bounds.
   (b) Generate a set of evenly spaced points S_j along this dimension.
2. Compute the Cartesian product of all these sets to generate the full grid of sample points.
3. Return the set of sampled points $\mathcal{D}$.

Full Factorial Sampling offers complete and uniform coverage of the search space. The total number of samples is determined by n^k, where n is the number of samples per dimension and k is the number of dimensions. This method is highly effective for low-dimensional problems, typically when $k \leqslant 3$. However, it struggles with the curse of dimensionality, as the number of samples increases exponentially with the number of dimensions, making it impractical for high-dimensional problems due to computational constraints. Full Factorial Sampling is a deterministic method that does not rely on randomness and ensures reproducibility. Despite these advantages, it can be inefficient if the objective function has varying sensitivities to different variables.

3.2.3 Latin Hypercube Sampling

Latin Hypercube Sampling (LHS) is a stratified sampling method that provides better coverage of the sample space than simple random sampling. It ensures that each of the input variables has all portions of its distribution represented by input values.

The Latin Hypercube Sampling algorithm works as follows:

1. For each dimension of the hyperrectangle $\mathcal{H}$:
   (a) Create a permutation of integers from 1 to N.
   (b) Shuffle this permutation randomly.
2. For each sample point:
   (a) For each dimension:

Algorithm 5 Latin Hypercube Sampling

Given:

$\mathcal{H}$: The hyperrectangle representing the search space

N : Number of samples to generate

Return:

$\mathcal{D}$: Set of N sampled points within $\mathcal{H}$

---

k : Dimensionality of $\mathcal{H}$

l_j, u_j : Lower and upper bounds for dimension j of $\mathcal{H}$

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

  $P_j \leftarrow \{1, 2, \ldots, N\}$ Shuffle P_j randomly

for $i = 1$ **to** N **do**

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

    $r \sim \text{Uniform}(0, 1)$

    $x_{i,j} \leftarrow l_j + \frac{P_j[i]-r}{N}(u_j - l_j)$

  $\mathcal{D} \leftarrow \mathcal{D} \cup \{x_i\}$

return $\mathcal{D}$

i. Generate a random number r between 0 and 1.

ii. Calculate the coordinate value using the formula:

$$x_{i,j} = l_j + \frac{P_j[i] - r}{N}(u_j - l_j)$$

(b) Add the generated point to the set of sampled points $\mathcal{D}$.

3. Return the set of sampled points $\mathcal{D}$.

Latin Hypercube Sampling offers several advantages over simple random sampling. It ensures more uniform coverage of the sample space and helps avoid the clustering of points, a common issue in random sampling. This method scales well with the number of dimensions, making it suitable for high-dimensional problems. Each variable is sampled independently, and fewer samples are required compared to simple random sampling to achieve a similar level of accuracy in estimating the mean of the output. Additionally, Latin Hypercube Sampling can be extended to control correlations between variables, such as with orthogonal sampling. It provides a balanced approach, combining the systematic nature of full factorial sampling with the randomness of simple random sampling.

Latin Hypercube Sampling is widely used in various fields, including computer experiments, uncertainty quantification, and sensitivity analysis. It is advantageous when the number of required samples is predetermined and when a good coverage of the input space is crucial for the accuracy of the surrogate model or the optimisation process.

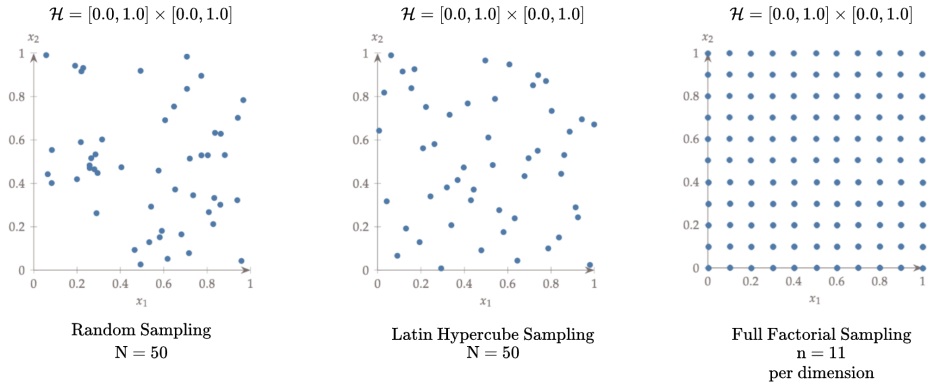


Figure 3.2: *Comparison of Random Sampling, Latin Hypercube Sampling, and Full Factorial Sampling techniques in a 2D space $\mathcal{H} = [0, 1] \times [0, 1]$. Random Sampling (left) shows an uneven distribution with potential clustering and gaps. Latin Hypercube Sampling (center) demonstrates improved space-filling properties with more uniform coverage across both dimensions. Full Factorial Sampling (right) provides a systematic grid-like coverage but requires more points (121) to achieve complete coverage compared to the other methods (50 points each).*

3.3 Surrogates

Surrogate models, or meta-models or emulators, are approximations of more complex and computationally expensive models or processes. In optimisation, surrogates are particularly useful when dealing with expensive objective functions, as they can provide quick estimates of the function's behaviour, thus accelerating the optimisation process.

3.3.1 Dataset, Training Set, and Test Set

When working with surrogate models, the available data is typically divided into two main subsets:

- **Training Set**: This is the portion of the data used to fit or train the surrogate model. It usually comprises the majority of the available data (e.g., 70-80%).
- **Test Set**: This is the remaining portion of the data used to evaluate the performance of the trained model on unseen data. It helps assess how well the model generalises.

Divining data into these sets is crucial for avoiding overfitting and ensuring that the surrogate model performs well on new, unseen data points.

3.3.2 Regression in Surrogate Modelling

Regression is a fundamental concept in surrogate modelling. It involves estimating the relationships between variables, typically predicting a continuous output variable based on one or more input variables. In the context of surrogate-based optimisation, regression models are used to approximate the complex objective function.

The general form of a regression model can be expressed as:

$$y = f(x) + \epsilon \tag{3.6}$$

where y is the output variable, x is the vector of input variables, $f(x)$ is the true underlying function, and ϵ is the error term.

3.3.3 Metrics for Surrogate Model Evaluation

To assess the quality and performance of surrogate models, several metrics are commonly used:

- **Mean Squared Error (MSE)**: Measures the average squared difference between predicted and actual values.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.7}$$

- **Root Mean Squared Error (RMSE)**: The square root of MSE, providing a measure in the same units as the output variable.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{3.8}$$

- **R-squared (R^2)**: Represents the proportion of variance in the dependent variable that is predictable from the independent variable(s).

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{3.9}$$

- **Mean Absolute Deviation (MAD)**: Measures the average absolute difference between predicted and actual values.

$$\text{MAD} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3.10}$$

These metrics provide different perspectives on the model's performance and should be considered collectively when evaluating surrogate models.

3.3.4 Random Forest Regression

Random Forest (RF) is an ensemble learning method that operates by constructing multiple decision trees during training and outputting the mean prediction of the individual trees for regression tasks. It was introduced by Leo Breiman in 2001 and has since become one of the most popular and effective machine-learning algorithms.

The Random Forest algorithm for regression can be summarised as follows:

1. Bootstrap Sampling: From the training set of size N, create n_trees bootstrap samples (sampling with replacement).
2. For each bootstrap sample:

(a) Grow an unpruned regression tree.

(b) At each node, rather than choosing the best split among all predictors, randomly sample m_try of the predictors and choose the best split among those variables.

3. Predict new data by aggregating the predictions of the n_trees trees (i.e., majority votes for classification, average for regression).

Key parameters in Random Forest regression include:

- **Number of Trees** : The number of decision trees in the forest. More trees can lead to better performance but increase computational cost.
- **Maximum Depth**: The maximum depth of each tree. Deeper trees can capture more complex patterns but may lead to overfitting.
- **Minimum Node Size**: The minimum number of samples required to split an internal node. Larger values prevent overfitting but might cause underfitting.
- **Number of Features to Consider**: The number of features to consider when looking for the best split. Usually set to the square root of the total number of features for regression.
- **Subsample Ratio**: The fraction of samples used to train each tree. Values less than 1.0 implement the Random Forest bagging ensemble.

Random Forests are known for their robustness and ability to handle high-dimensional data without overfitting.

3.3.5 Gradient Tree Boost

The Gradient Boosting algorithm for regression can be summarised as follows:

1. Initialise model with a constant value: $F_0(x) = \arg\min_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$
2. For m = 1 to M:
   (a) Compute pseudo-residuals: $r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ for $i = 1, \ldots, n$
   (b) Fit a regression tree to the pseudo-residuals, giving terminal regions R_{jm}, for $j = 1, \ldots, J_m$
   (c) For each terminal region R_{jm}, compute: $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$
   (d) Update the model: $F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$
3. Output $F_M(x)$

Where $L(y, F(x))$ is the loss function, ν is the learning rate, and I is the indicator function.

Key parameters in Gradient Tree Boost (GTB) include:

- **Number of Trees** : The number of boosting stages to perform. Gradient boosting is fairly robust for over-fitting, so a large number usually results in better performance.
- **Learning Rate** : Shrinks the contribution of each tree, helping to prevent overfitting.
- **Maximum Depth**: The maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree.
- **Minimum Samples Split** : The minimum number of samples required to split an internal node.
- **Subsample**: The fraction of samples to fit the individual base learners. If smaller than 1.0, this results in Stochastic Gradient Boosting.
- **Loss Function**: The loss function to be optimised. Common choices include least squares, least absolute deviation, and Huber loss.

Gradient Tree Boost models are particularly effective for handling complex, non-linear relationships in data.

3.3.6 Radial Basis Function Network

Radial Basis Function (RBF) Networks are artificial neural networks that use radial basis functions as activation functions. They were introduced by Broomhead and Lowe in 1988. RBF networks are useful for function approximation, time series prediction, and control.

An RBF network typically consists of three layers:

1. Input layer: Passes the input features to the hidden layer.
2. Hidden layer: Contains RBF neurones, each with centre and width.
3. Output layer: Computes the weighted sum of hidden layer outputs.

The output of an RBF network can be expressed as:

$$f(x) = \sum_{i=1}^{N} w_i \phi(\|x - c_i\|) \tag{3.11}$$

Where x is the input vector, c_i are the RBF centres, w_i are the output weights and ϕ is the radial basis function.

Common choices for the radial basis function include:

- Gaussian: $\phi(r) = e^{-(\epsilon r)^2}$
- Multiquadric: $\phi(r) = \sqrt{1 + (\epsilon r)^2}$
- Inverse quadratic: $\phi(r) = \frac{1}{1+(\epsilon r)^2}$
- Inverse multiquadric: $\phi(r) = \frac{1}{\sqrt{1+(\epsilon r)^2}}$

Where $r = \|x - c\|$ is the distance from the center, and ϵ is a shape parameter.
Training an RBF network typically involves three steps:

1. Determine the RBF centers (e.g., using k-means clustering on the input data).
2. Compute the RBF activations for each input.
3. Solve for the output weights using least squares.

Key parameters

Key parameters in RBF Networks include:

- **Number of Neurons** : The number of RBF neurons in the hidden layer. More neurons can capture more complex patterns but may lead to overfitting.
- **Normalised** : Whether to normalise the output of RBF neurons. Normalisation can help stabilise the network's output.

RBF Networks are known for their ability to model complex local relationships in the data and their relatively fast training process compared to other neural network architectures.

3.4 Optimisation algorithm

3.4.1 Particle swarm optimisation

Particle Swarm Optimisation (PSO) is a population-based stochastic optimisation technique inspired by the social behaviour of bird flocking or fish schooling. Developed by Kennedy and Eberhart in 1995, PSO has become a popular method for solving complex optimisation problems due to its simplicity, effectiveness, and ability to handle high-dimensional search spaces.

In PSO, a swarm of particles moves through the search space, with each particle representing a potential solution to the optimisation problem. The movement of these particles is influenced by both their personal best position (cognitive component) and the global best position found by the entire swarm (social component). This dual influence allows the algorithm to balance exploring new areas and exploiting well-known solutions.

The key components of the PSO algorithm are:

- **Particles**: Each particle represents a candidate solution and has a position x_i and velocity v_i in the search space.
- **Personal Best**: Each particle keeps track of its best position p_i found so far, based on the objective function value.
- **Global Best**: The best position g found by any particle in the entire swarm.
- **Velocity Update**: The velocity of each particle is updated based on its current velocity, personal best, and global best positions.
- **Position Update**: The position of each particle is updated based on its velocity.

The velocity update rule, which is the core of the PSO algorithm, is given by:

$$v_i^{t+1} = wv_i^t + c_1 r_1 (p_i - x_i^t) + c_2 r_2 (g - x_i^t) \tag{3.12}$$

where w is the inertia weight, c_1 and c_2 are acceleration coefficients, and r_1 and r_2 are random numbers between 0 and 1. The position is then updated as:

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{3.13}$$

PSO's strength lies in its ability to handle complex, non-linear optimisation problems with multiple local optima. It has been successfully applied in various fields, including engineering design, machine learning, and financial modelling. The algorithm's performance can be tuned through parameters such as the inertia weight, acceleration coefficients, and swarm size, allowing for adaptation to different problem domains.

Algorithm 6 Particle Swarm Optimisation (PSO)

Given:

f_Ω : Objective function to be minimised

$\mathcal{H}$: The hyperrectangle representing the search space

Return:

x^* : The global best position (solution)

$w \in (0,1)$ *default :* 0.72984

$c_1 \in (0,\infty)$ *default :* 1.49617

$c_2 \in (0,\infty)$ *default :* 1.49617

$N \in \mathbb{N}$ *default :* 100

$T \in \mathbb{N}$ *default :* 1000

for $i = 1$ **to** N **do**

  $x_i \sim \text{Uniform}(\mathcal{H})$ *initialise particle at random position*

  $p_i \leftarrow x_i$ *initial personal best for each particle is its starting position*

$g \leftarrow \arg\min_i f_\Omega(p_i) : \forall i \in N$ *initial global best is the best among all these starting positions*

for $t = 1$ **to** T **do**

  for $i = 1$ **to** N **do**

    Update velocity: $v_i = wv_i + c_1 r_1 (p_i - x_i) + c_2 r_2 (g - x_i)$

    Update position: $x_i = x_i + v_i$

    Ensure x_i remains within $\mathcal{H}$

    if $f_\Omega(x_i) < f_\Omega(p_i)$ **then**

      Update personal best: $p_i = x_i$

    if $f_\Omega(p_i) < f_\Omega(g)$ **then**

      Update global best: $g = p_i$

return $x^* = g$

3.4.2 Mesh adaptive direct search

Mesh Adaptive Direct Search (MADS) [14] is an advanced optimisation algorithm designed to overcome the limitations of earlier direct search methods. It builds upon the foundations laid by the Coordinate Search (CS) and Generalised Pattern Search (GPS) algorithms while introducing key innovations to enhance its applicability and convergence properties.

The Coordinate Search (CS) algorithm is a foundational method for unconstrained optimisation, relying on local exploration around the current solution by moving in the positive and negative directions along each coordinate axis. This approach was later extended into the GPS algorithm, which broadened the search to include a broader range of directions.

However, the convergence analysis of the GPS algorithm presents certain limitations. Firstly, the main result indicates that for an objective function f that is locally Lipschitz, the generalised directional derivatives are nonnegative in only a finite set of directions. Secondly, and perhaps more critically, the GPS analysis primarily addresses unconstrained optimisation problems, even though most practical problems involve constraints on the variables.

To overcome these limitations, the Mesh Adaptive Direct Search (MADS) algorithm was developed as an advance of the GPS algorithm. The MADS algorithm extends the GPS framework by enabling convergence analysis for both constrained and unconstrained optimisation problems. It addresses the GPS limitation of proving nonnegativity in only a finite set of directional derivatives. Unlike GPS, which relies on a finite positive spanning set for initialising and selecting future polling directions, MADS generalises this by allowing an infinite set of polling directions, thereby enhancing the convergence properties of the algorithm.

Definition 35 (**Mesh and Mesh Size Parameter** $\Delta_{\mathbf{m}}^k$)
Let $G \in \mathbb{R}^{n\times n}$ be invertible and $Z \in \mathbb{Z}^{n\times p}$ be such that the columns of Z form a positive spanning set for $\mathbb{R}^n$. Define $D = GZ$. The mesh of coarseness $\Delta_{\mathrm{m}}^k > 0$ generated by D, centred at the incumbent solution $x^k \in \mathbb{R}^n$, is defined by

$$\mathrm{M}^k := \left\{x^k + \Delta_{\mathrm{m}}^k Dy : y \in \mathbb{N}^p\right\} \subset \mathbb{R}^n$$

where Δ_{m}^k is called the mesh size parameter.

With the term coarseness, we refer to an intuitive sense of the granularity or resolution of the mesh. A coarse mesh has more significant gaps between points, while a fine mesh has smaller gaps. In the GPS algorithm, the mesh size parameter is also used to control the poll step. In MADS a new parameter, the pool frame size parameter Δ_{p}^k, that replace Δ_{m}^k in the pool step. The poll frame size parameter Δ_{p}^k defines the frame in which polling is done. At each iteration k these two parameters will satisfy $0 < \Delta_{\mathrm{m}}^k \leqslant \Delta_{\mathrm{p}}^k$

Definition 36 (Poll Frame and Poll Frame Size Parameter $\Delta_{\mathbf{p}}^k$)
Let $G \in \mathbb{R}^{n\times n}$ be invertible and $Z \in \mathbb{Z}^{n\times p}$ be such that the columns of Z form a positive spanning set for $\mathbb{R}^n$. Define $D = GZ$. Select a mesh size parameter $\Delta_{\mathrm{m}}^k > 0$ and let Δ_{p}^k be such that $\Delta_{\mathrm{m}}^k \leqslant \Delta_{\mathrm{p}}^k$. The poll frame of extent Δ_{p}^k generated by D, centred at the incumbent solution $x^k \in \mathbb{R}^n$, is defined by

$$\mathrm{F}^k := \left\{x \in M^k : \left\|x - x^k\right\|_\infty \leqslant \Delta_{\mathrm{p}}^k b\right\}$$

with $b = \max\{\|d'\|_\infty : d' \in \mathbb{D}\}$ and where Δ_{p}^k is called the poll frame size parameter.

Notice that $\mathrm{F}^k \subseteq \mathrm{M}^k$, in the MADS algorithm, the poll set is constructed by selecting mesh points from inside the poll frame P^k By having the mesh size parameter decrease more rapidly than the frame size parameter, this allows for asymptotically dense selections of poll directions.

The MADS algorithm is shown Algorithm 8, it consists of a series of iterations, each containing two main steps: the SEARCH step and the POLL step. The algorithm maintains a current best solution and explores the surrounding space for better solutions. The SEARCH step is flexible and optional. It allows any strategy to generate trial points on the current mesh. This step can incorporate problem-specific knowledge or use various heuristics to explore promising areas of the search space. If the SEARCH step fails to find an improved solution, the algorithm proceeds to the POLL step. The POLL step is more rigidly defined and is the core of the MADS algorithm. It explores the space around the current best solution systematically. The key feature of MADS is its use of a poll size parameter (Δ_p^k) that is distinct from the mesh size parameter (Δ_m^k). This allows for an asymptotically dense set of poll directions, which is crucial for the convergence properties of the algorithm. The POLL step generates a set of poll directions based on the current poll size parameter. These directions form a positive spanning set, ensuring that all directions in the space can be approximated. The algorithm then evaluates the objective function at mesh points along these directions within the poll frame. If an improved solution is found, it becomes the new best solution, and the poll size parameter is increased. If no improvement is found, the poll size parameter is decreased. The mesh and poll size parameters are updated after each iteration based on whether an improved solution was found. This adaptive mesh refinement allows MADS to balance between the broad exploration of the search space and the fine-tuning of promising solutions.

LtMads: Lower Triangular MADS

LtMads [13] generates positive spanning sets using a randomised approach based on lower triangular matrices. It constructs a random integer lower triangular matrix B_k at each iteration. The columns of this matrix form a basis for $\mathbb{R}^n$. To create a positive spanning set, LtMads completes this basis in one of two ways:

1. For a minimal positive basis, it adds a column b that is the negative sum of all columns in B_k, resulting in $D_k = [B_k\ b]$.

Algorithm 8 Mesh adaptive direct search (MADS)

Given:

f_Ω : Barrier function

$\mathcal{H}$: The Hyperectangle representing the search space

Return:

x^k : The argmin, the input to a function that yields the minimum

0. Initialisation

$k \leftarrow 0$

$k_{\text{max}} \in \mathbb{N}$ *default :* 500

$x_0 \in \mathcal{H}$ *default :* $x_0 \sim \text{Uniform}(\mathcal{H})$

$\Delta_0^p \in (0, \infty)$ *default :* 1.0

$\tau \in (0, 1)$ *default :* $1/4$

$\epsilon_{\text{stop}} \in [0, \infty)$ *default :* $1.0E - 50$

$D \leftarrow GZ$ *positive spanning matrix*

1. Parameter Update

$\Delta_k^m = \min\{\Delta_k^p, (\Delta_k^p)^2\}$

2. Search

$S^k \leftarrow \text{getSearchPointsSet}(\mathcal{H}) : S^k \subseteq M^k$

for each $t \in S^k$ **do**

  if $f_\Omega(t) < f_\Omega(x_k)$ **then**

    $x_{k+1} \leftarrow t$

    $\Delta_{k+1}^p \leftarrow \tau^{-1}\Delta_k^p$

    go to 4

  else

    go to 3

3. Poll

$\mathbb{D}_p^k \leftarrow \text{getPositiveSpannigSet}(\Delta_p^k)$

$P^k \leftarrow \{x_k + \Delta_m^k d : d \in \mathbb{D}_p^k\} : P^k \subseteq F^k$

for each $t \in P^k$ **do**

  if $f_\Omega(t) < f_\Omega(x_k)$ **then**

    $x_{k+1} \leftarrow t$

    $\Delta_{k+1}^p \leftarrow \tau^{-1}\Delta_k^p$

  else

    $x_{k+1} \leftarrow x^k$

    $\Delta_{k+1}^p \leftarrow \tau\Delta_k^p$

4. Termination

if $\Delta_p^{k+1} \geqslant \epsilon_{stop} \wedge k \leqslant k_{\text{max}}$ **then**

  $k \leftarrow k + 1$

  go to 1

else

  return x^k

2. For a maximal positive basis, it includes the negatives of all columns, giving $D_k = [B_k \ -B_k]$.

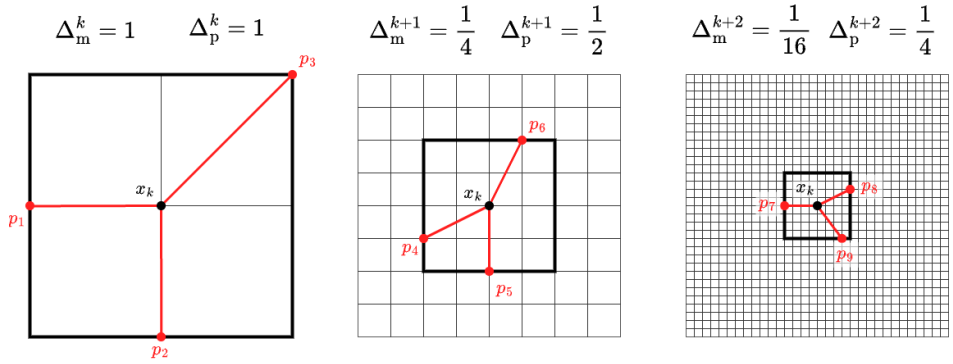


Figure 3.3: *LtMADS poll directions and mesh refinement over successive iterations. The lower triangular poll directions are shown as the mesh size parameter Δ_m^k and poll size parameter Δ_p^k decrease, resulting in a finer mesh and more focused polling around the incumbent solution x_k.*

The randomness in constructing B_k ensures that, with probability one, the set of normalised poll directions becomes dense on the unit sphere as the number of iterations approaches infinity. This property is crucial for the convergence analysis of LtMads. Figure 3.3 illustrates LtMADS poll directions and mesh refinement across iterations, showing increasingly localised search around x_k.

OrthoMads: Orthogonal MADS

OrthoMADS [2] takes a deterministic approach to generate orthogonal positive spanning sets. The process involves several steps:

1. It uses a Halton sequence to generate a vector in $[0, 1]^n$.
2. This vector is then scaled and rounded to create an integer vector $q_{t,\ell}$, called the adjusted Halton direction.
3. A Householder transformation is applied to $q_{t,\ell}$ to produce an orthogonal basis $H_{t,\ell}$ for $\mathbb{R}^n$.
4. The positive spanning set is formed as $D_k = [H_{t,\ell} \ -H_{t,\ell}]$.

Halton sequences ensure a uniform distribution of directions over the long run, while the Householder transformation guarantees orthogonality. This deterministic approach leads to more consistent performance across different runs and potentially better search space coverage. Figure 3.4 illustrates OrthoMADS poll directions and mesh refinement across iterations, showing increasingly localised search around x_k.

Both LtMads and OrthoMADS maintain the core principles of MADS, including the distinction between mesh size and poll size parameters, but their different strategies for generating poll directions can lead to varying performance depending on the specific optimisation problem being solved.

3.4.3 Simulated Annealing

Simulated Annealing (SA) is a probabilistic technique to approximate the global optimum of a given function. Inspired by the annealing process in metallurgy, this algo-

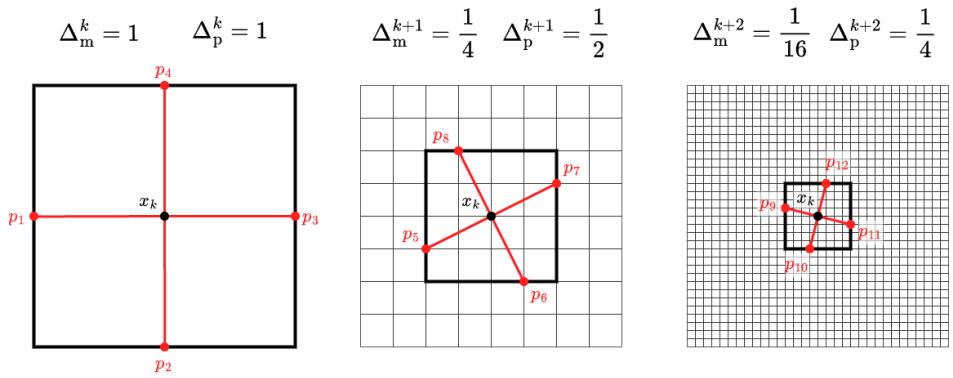


Figure 3.4: *OrthoMADS poll directions and mesh refinement over successive iterations. As the mesh size parameter Δ_m^k and poll size parameter Δ_p^k decrease, the mesh becomes finer, and polling is more localised around the incumbent solution x_k.*

rithm was introduced by Kirkpatrick, Gelatt, and Vecchi in 1983. SA is particularly effective for solving complex optimisation problems with large, discrete search spaces.

The key idea behind SA is to mimic the physical process of heating a material and slowly lowering the temperature to decrease defects, thus minimising the system energy. In optimisation, this translates to initially allowing an exhaustive exploration of the search space (high temperature) and gradually focusing on local optimisation as the algorithm progresses (low temperature).

The main components of the Simulated Annealing algorithm are:

- **Current Solution**: The current state in the search space.
- **Neighbor Generation**: A method to create a new candidate solution near the current one.
- **Acceptance Probability**: A function that decides whether to accept a new solution based on the current temperature and the difference in objective function values.
- **Cooling Schedule**: A strategy for decreasing the temperature over time.

The acceptance probability is typically calculated as:

$$P(\text{accept}) = \begin{cases} 1 & \text{if } \Delta E < 0 \\ e^{-\Delta E/T} & \text{otherwise} \end{cases} \tag{3.14}$$

Where ΔE is the change in the objective function value and T is the current temperature.

A crucial aspect of SA is the cooling schedule, which determines how the temperature decreases over time. Three common cooling schedules are:

1. **Linear Cooling**:

$$T_{k+1} = T_k(1 - \alpha) \tag{3.15}$$

where α is the cooling rate.

2. **Exponential Cooling**:
$$T_k = T_0 e^{-\alpha k} \tag{3.16}$$
where T_0 is the initial temperature and k is the current iteration.

3. **Logarithmic Cooling**:
$$T_k = \frac{T_0}{1 + \log(1 + k)} \tag{3.17}$$

The choice of cooling schedule can significantly impact the algorithm's performance. Linear and exponential cooling are often preferred in practice due to their simplicity and effectiveness. At the same time, logarithmic cooling ensures theoretical convergence to the global optimum but may be too slow for many practical applications.

Simulated Annealing's strength lies in its ability to escape local optima through probabilistic acceptance of worse solutions, especially in the early stages of the search. This makes it particularly useful for problems with many local optima or noisy objective functions. SA has been successfully applied in various fields, including very large-scale integration (VLSI) design, image processing, and scheduling problems.

Algorithm 9 Simulated Annealing (SA)

Given:
f_Ω : Objective function to be minimised
$\mathcal{H}$: The hyperrectangle representing the search space
Return:
x^* : The best solution found

T_0	$\in (0, \infty)$	*default :* 1000.0
α	$\in (0, 1)$	*default :* 0.003
$k_{\max}$	$\in \mathbb{N}$	*default :* 1000
cooling	$\in$ {linear, exponential, logarithmic}	*default : linear*

$x_{\text{current}} \leftarrow$ generateInitialSolution($\mathcal{H}$)
$x^* \leftarrow x_{\text{current}}$
$T \leftarrow T_0$
for $k = 1$ **to** $k_{\max}$ **do**
  $x_{\text{new}} \leftarrow$ generateNeighborSolution($x_{\text{current}}, \mathcal{H}$)
  $\Delta E \leftarrow f_\Omega(x_{\text{new}}) - f_\Omega(x_{\text{current}})$
  if $\Delta E < 0$ ***or*** *random*$(0, 1) < e^{-\Delta E/T}$ **then**
    $x_{\text{current}} \leftarrow x_{\text{new}}$
  if $f_\Omega(x_{current}) < f_\Omega(x^*)$ **then**
    $x^* \leftarrow x_{\text{current}}$
  $T \leftarrow \text{cool}(T, k, k_{\max})$
return x^*

3.5 Infill Strategies

Infill strategies play a crucial role in surrogate-based optimisation by guiding the selection of new sample points for evaluation. These strategies aim to balance the exploration of the design space with the exploitation of promising regions identified by the surrogate model. This section discusses some common infill strategies and introduces a simple yet effective technique implemented in this work.

3.5.1 Common Infill Strategies

Two popular infill criteria in surrogate-based optimisation are:

Expected Improvement (EI)

EI quantifies the potential improvement a new point could offer over the current best solution. For a minimisation problem, the EI at a point x is defined as:

$$\text{EI}(x) = \mathbb{E}[\max(f_{\text{min}} - Y(x), 0)] \tag{3.18}$$

where f_{min} is the current best function value, and $Y(x)$ is a random variable representing the prediction at x according to the surrogate model.

Lower Confidence Bound (LCB)

LCB provides a balanced approach to exploration and exploitation. For a minimisation problem, the LCB at a point x is defined as:

$$\text{LCB}(x) = \hat{f}(x) - \kappa \hat{s}(x) \tag{3.19}$$

Where $\hat{f}(x)$ is the predicted mean, $\hat{s}(x)$ is the predicted standard deviation, and $\kappa > 0$ is a parameter that controls the trade-off between exploration and exploitation.

3.5.2 Shrinking Trust Region with Increased Sampling

In this work, we implement a simple infill strategy called "Shrinking Trust Region with Increased Sampling". This technique focuses on progressively refining the search around the best solution and increasing the sampling density. The strategy operates as follows:

1. Center the search space around the current best solution.
2. Scale down the search space by a factor α (default $\alpha = 0.5$).
3. Increase the number of samples by a factor β (default $\beta = 1.5$).
4. Repeat until a stopping condition is met.

The stopping condition is typically based on reaching a maximum number of infill iterations or achieving convergence between the true objective function and the surrogate model prediction.

This approach balances exploitation (focusing on the region around the current best solution) and exploration (increasing the sampling density in this region). It is particularly effective in problems where the global optimum is likely to be near the current best solution.

This algorithm provides a simple yet effective way to refine the search space and increase sampling density around promising solutions. Iteratively shrinking the trust region and increasing the number of samples allows for a focused exploration of the most promising areas of the design space.

Algorithm 10 Shrinking Trust Region with Increased Sampling

Given:

f_Ω : Objective function to be minimised
$\hat{f}$: Surrogate model
$\mathcal{H}$: The hyperrectangle representing the search space
x_{best} : Current best solution
N : Initial number of samples
α : Scaling factor for search space (default: 0.5)
β : Multiplier for number of samples (default: 1.5)
$i_{\max}$: Maximum number of infill iterations
ϵ : Convergence threshold

$i \leftarrow 0$
while $i < i_{\max}$ **do**
  for $\mathrm{I} \in \mathcal{H}$ **do**
    $\mathrm{I}.\text{center} \leftarrow x_{\text{best,I}}$
    $\text{length} \leftarrow \mathrm{I}.u - \mathrm{I}.l$
    $\mathrm{I}.l \leftarrow \mathrm{I}.\text{center} - \text{length}/2$
    $\mathrm{I}.u \leftarrow \mathrm{I}.\text{center} + \text{length}/2$
  $\mathcal{H}' \leftarrow \varnothing$
  for $\mathrm{I} \in \mathcal{H}$ **do**
    $\mathrm{I}'.\text{center} \leftarrow (\mathrm{I}.u + \mathrm{I}.l)/2$
    $\text{newLength} \leftarrow (\mathrm{I}.u - \mathrm{I}.l) * \alpha$
    $\mathrm{I}'.l \leftarrow \mathrm{I}'.\text{center} - \text{newLength}/2$
    $\mathrm{I}'.u \leftarrow \mathrm{I}'.\text{center} + \text{newLength}/2$
    $\mathcal{H}' \leftarrow \mathcal{H}' \cup \{\mathrm{I}'\}$
  $\mathcal{H} \leftarrow \mathcal{H}'$
  $N' \leftarrow \lceil \beta \cdot N \rceil$

  $x'_{\text{best}} \leftarrow \text{SBO.evaluatesOptimal}()$

  if $|f_\Omega(x'_{best}) - \hat{f}(x'_{best})| \leqslant \epsilon$ **then**
    break

  $x_{\text{best}} \leftarrow x'_{\text{best}}$

  $i \leftarrow i + 1$

CHAPTER 4

State of the Art

"Hic sunt leones."

— Ancient cartographers, marking unexplored territory

THIS chapter presents a survey of current research in modelling, verification, and synthesis of Collective Adaptive Systems (CAS). Before delving into the Sibilla framework in the next chapter, we provide an overview of existing approaches to CAS analysis, contextualizing our contribution within the broader research landscape.

The theoretical foundations established in previous chapters—quantitative analysis techniques in Chapter 2 and surrogate-based optimisation methods in Chapter 3—serve as the mathematical underpinnings of our work. Building on these foundations, we now examine how different research communities have approached CAS analysis, focusing on existing methodologies, algorithms, and tools.

This state-of-the-art review covers several key aspects of CAS research: modelling formalisms and languages, verification techniques, parameter synthesis methods and available software tools.

By analyzing the current research landscape, we identify significant gaps in existing approaches, particularly concerning integrated workflows that connect modelling, verification, and synthesis; scalable methods for handling large-scale heterogeneous systems; and practical tools that bridge theoretical advances with engineering needs. These limitations directly motivate the development of the Sibilla framework, which addresses these challenges through novel integration of statistical model checking with surrogate-based optimization techniques.

4.1 Modeling Formalisms and Languages

In Section 2.2, we introduced fundamental modeling approaches which provide the mathematical foundation for representing stochastic systems. For CAS, however, more sophisticated high-level formalisms are typically needed to capture their distinctive characteristics of massive scale, emergent behavior, and spatial embedding.

We will list several modeling formalisms used in the literature, each offering different capabilities for representing the complex dynamics of CAS:

Stochastic Process Algebras Process algebras provide compositional frameworks for specifying concurrent systems through algebraic operators:

- **PEPA** (Performance Evaluation Process Algebra) extends classical process algebras with rates, enabling compositional specification of Markovian systems [131]. Extensions like Bio-PEPA [62] incorporate specialized operators for biochemical reactions.
- **CARMA** (Collective Adaptive Resource-sharing Markovian Agents) [164] introduces explicit spatial distribution, attribute-based communication, and environment-mediated interactions specifically designed for CAS.
- **PALOMA** (Process Algebra for Located Markovian Agents) [105] focuses on location-aware communication patterns and message passing.
- **StocS** (Stochastic Concurrent Strategy Language) [34] integrates game-theoretic elements with stochastic processes for modeling strategic behaviors in CAS.

Process algebras offer compositional reasoning and abstraction mechanisms but often lack intuitive representations for spatial dynamics and complex agent behaviors.

Rule-Based Languages Rule-based approaches specify system evolution through transformation rules applied to structured states:

- **Kappa** [71] and **BioNetGen** [103] represent biochemical systems through site graphs and rules, enabling compact representation of combinatorially complex systems.
- **Bigraphs** [151, 172] combine topology and connectivity in a unified framework, supporting both spatial embedding and communication links.
- **Attributed Graph Grammars** [22, 125] extend graph transformation systems with attributes and constraints, enabling rich structural specifications.
- **PDEVS** (Parallel Discrete Event System Specification) [210] provides a hierarchical formalism for discrete event simulations with explicit timing.

Rule-based approaches excel at representing complex state structures and transformation patterns but may lack modularity and become unwieldy for systems with intricate behavioral logic.

Agent-Based and Individual-Based Models Agent-based approaches center on individual entities with autonomous behaviors:

- **NetLogo** [205] and **MASON** [166] provide programming environments for agent-based simulations with spatial embedding.
- **X-Machines** [133] extend finite state machines with memory and functional transformations, enabling rich agent behavioral specifications.
- **Cellular Automata** [206] offer simplified agent models with local interaction rules, particularly useful for spatial pattern formation analysis.
- **ABMS** (Agent-Based Modeling and Simulation) frameworks like **Repast** [179] and **FLAME** [191] support large-scale simulations with sophisticated visualization and analysis capabilities.

Agent-based approaches provide intuitive representations of individual behaviors but often lack formal semantics for rigorous verification and may face scalability challenges.

Hybrid and Multi-Scale Models Complex CAS frequently require integration of continuous dynamics with discrete events across multiple scales:

- **Stochastic Hybrid Automata** [32] combine continuous evolution with discrete jumps, suitable for biochemical and cyber-physical systems.
- **DEVS/DESS Integration** [210] couples discrete event and differential equation system specifications.
- **ML-Rules** [170] supports hierarchical nesting of entities with rule-based transformations across scales.
- **PyCATSHOO** [38] implements hybrid stochastic models with multi-scale temporal dynamics.
- **HYPE** [110] provides a process-algebraic approach to hybrid systems with influenced flow semantics.

Hybrid approaches address the multi-scale nature of many CAS but introduce additional complexity in both specification and analysis.

Spatial and Topological Models Explicit representation of space and topology is crucial for many CAS applications:

- **Spatialized Process Algebras** like **PALPS** [141] and **Space** π [142] incorporate continuous space into process calculi.
- **Spatial Logics** including **SLCS** (Spatial Logic for Closure Spaces) [59] and **STREL** (Spatio-Temporal Reach and Escape Logic) [24] specify spatial patterns and reachability properties.
- **Geographic Cellular Automata** [21] extend classical cellular automata with irregular spatial arrangements.

- **Dynamic Bayesian Networks** with spatial dependencies [175] model evolving spatial relationships probabilistically.
- **Topological Graph Dynamics** [49] focus on connection patterns and rewriting of network structures.

Spatial models capture essential locality constraints but introduce computational challenges for verification and analysis.

4.2 Verification and Analysis Techniques

The verification of CAS spans a spectrum from exact analytical methods to approximation and simulation-based approaches, each with distinct computational properties and applicability domains. Building on the verification principles introduced in Section 2.5.3, we now examine how these techniques have been adapted and extended to address the specific challenges posed by CAS.

We will list several verification techniques used in the literature, highlighting their approaches, strengths, and limitations in the context of complex stochastic systems such as CAS:

Analytical Solutions and Symbolic Model Checking Exact analytical methods provide provable guarantees for quantitative and qualitative properties:

- **Numerical Solution of Markov Models**: Computing transient and steady-state distributions through matrix operations, uniformization [203], and Kronecker algebra [46].
- **Symbolic Model Checking**: Representing state spaces through BDDs (Binary Decision Diagrams) [45] and MTBDDs (Multi-Terminal BDDs) [17] to mitigate state-space explosion.
- **Compositional Verification**: Exploiting model structure through bisimulation minimization [145], assume-guarantee reasoning [159], and symmetry reduction [91].
- **Parameterized Verification**: Verifying systems with arbitrary numbers of similar components through cutoff techniques [101], induction [143], and abstraction [65].
- **Fluid Model Checking**: Approximating large-scale discrete systems with continuous deterministic or stochastic approximations [34, 202].

Despite their mathematical precision, analytical methods face fundamental limits for complex CAS:

- State-space explosion remains severe despite symbolic and compositional techniques.
- Hybrid systems with nonlinear dynamics often yield undecidable verification problems [130].

- Parameter synthesis over rational functions is ETR-complete (Existential Theory of Reals) [84].
- Compositional approaches struggle with emergent properties that cannot be decomposed.

Simulation and Statistical Model Checking Simulation-based approaches provide scalability through sampling of execution trajectories:

- **Discrete Event Simulation**: Event-scheduling algorithms [210] efficiently traverse possible system trajectories.
- **Statistical Model Checking (SMC)**: Hypothesis testing frameworks [161, 209] establish probabilistic guarantees for temporal properties.
- **Rare Event Simulation**: Importance sampling [193], splitting techniques [112], and cross-entropy methods [79] address low-probability events.
- **Runtime Verification**: Monitoring execution traces against temporal specifications [26] with online algorithms.
- **Approximate Model Checking**: Bounded satisfaction [147], three-valued semantics [158], and probabilistic abstraction [73] trade precision for computational efficiency.
- **Fluid Limit Approximations**: Approximating discrete stochastic processes with ordinary differential equations [35, 202].

SMC offers several advantages for complex CAS:

- **Model Agnosticism**: Works with arbitrary simulatable models regardless of their internal complexity.
- **Sequential Analysis**: Adjusts sample sizes dynamically based on observed statistical evidence [204, 209].
- **Bounded Error Guarantees**: Provides statistical guarantees with configurable error bounds [161].
- **Parallelizability**: Naturally distributes across computing resources [209].

However, SMC approaches face challenges with rare events, unbounded properties, and nondeterministic systems [161, 209].

Hybrid and Compositional Verification Recent approaches combine multiple verification strategies to balance precision and scalability:

- **Three-Valued Abstraction**: Combining over- and under-approximations with refinement [158].
- **Statistical-Numerical Verification**: Using simulation to guide exact analysis of critical subsystems [209].

- **Fluid Model Checking**: Mean-field approximations of large populations with continuous semantics [35].
- **Moment-Based Verification**: Analyzing truncated moment equations instead of full distributions [36].
- **Counting-Based Abstractions**: Tracking population counts rather than individual entities [51].
- **Pattern-Oriented Verification**: Focusing analysis on specific interaction patterns rather than global states [82].

These approaches are particularly relevant for CAS, where traditional verification methods often fail due to scale, heterogeneity, and emergent behaviors.

4.3 Parameter Synthesis and Optimisation

Parameter synthesis (Section 2.6) aims to characterize or optimize the set of model configurations that satisfy design requirements, a critical challenge for CAS where manual parametrization is infeasible. In Section 5.5, we introduced surrogate-based optimization as our approach to parameter synthesis, but it's worth examining other techniques developed in the literature that address this challenging problem.

Symbolic and Exact Methods Exact approaches to parameter synthesis offer provable guarantees:

- **Symbolic Parameter Inference**: Using rational function representations derived through state elimination [76, 118], these methods compute closed-form expressions for satisfaction probabilities as functions of model parameters.
- **SMT-Based Approaches**: Encoding synthesis as satisfiability modulo theories over nonlinear real arithmetic [61], leveraging powerful SMT solvers to find parameter values that satisfy complex constraints.
- **Parametric Bisimulation Minimization**: Exploiting structural symmetries to reduce model complexity [86], making exact analysis feasible for larger systems through state space reduction.
- **QE-Based Methods**: Applying quantifier elimination to parameter constraints [109], transforming existential queries into equivalent quantifier-free formulas that characterize feasible parameter regions.
- **Region-Based Computation**: Characterizing feasible parameter regions through exact polynomial constraints [190], providing precise descriptions of the parameter space where properties are satisfied.

Exact methods provide comprehensive guarantees but face fundamental scalability limitations, particularly for nonlinear parametric dependencies and high-dimensional parameter spaces. These approaches typically become intractable for systems with more than a handful of parameters.

Approximate and Learning-Based Methods Approximation techniques trade theoretical guarantees for practical scalability:

- **Sequential Convex Programming (SCP)**: Approximating non-convex feasibility regions through iterative linearization [37], allowing efficient optimization using well-established convex programming algorithms.
- **Parameter Lifting**: Abstracting parametric Markov chains to interval Markov decision processes [190], providing bounds on satisfaction probabilities across parameter ranges.
- **Statistical Parameter Synthesis**: Using statistical model checking with adaptive sampling [25, 140], estimating satisfaction probabilities through simulation rather than explicit state-space exploration.
- **Counterexample-Guided Parameter Synthesis**: Refining parameter regions based on generated counterexamples [61], focusing computational effort on areas where property violations occur.
- **Gaussian Process Surrogate Models**: Approximating verification functions through statistical learning [25, 37], similar to the surrogate-based approach detailed in Chapter 3.
- **Evolutionary and Genetic Algorithms**: Population-based optimization for complex parameter landscapes [194], leveraging nature-inspired search strategies to explore high-dimensional parameter spaces.
- **Reinforcement Learning**: Framing synthesis as a sequential decision process with policy learning [169], particularly useful for systems where parameters influence behavior over time.

These approaches have enabled synthesis for models with hundreds of parameters and complex behaviors, though they typically provide statistical rather than formal guarantees. The surrogate-based approach implemented in Sibilla, as described in Section 5.5, falls within this category, offering practical scalability for complex CAS models.

Synthesis Beyond Markov Models Parameter synthesis extends beyond traditional Markov models to diverse CAS formalisms:

- **Parametric Timed Automata**: Synthesizing timing parameters for real-time systems [10], ensuring temporal requirements are met with formal guarantees.
- **Parametric Hybrid Systems**: Finding parameter regions for safety and reachability [109], particularly relevant for cyber-physical systems with mixed continuous and discrete behaviors.
- **Rule-Based Systems**: Optimizing rate constants in rule-based biochemical models [104], tuning reaction rates to achieve desired system behaviors in computational biology applications.

- **Agent-Based Models**: Calibrating agent parameters against observed collective behaviors [53], aligning simulation outputs with empirical data in social and ecological systems.
- **Neural Network Controllers**: Synthesizing robust neural policies with formal guarantees [3], combining machine learning with verification for safe autonomous systems.

These extensions bridge formal methods with domain-specific modeling frameworks relevant to CAS. The diversity of synthesis approaches reflects the varied requirements across application domains and modeling formalisms.

The synthesis approach implemented in Sibilla, described in Section 5.5, builds upon these established techniques, particularly the surrogate-based optimization methods, while addressing specific challenges of CAS through efficient integration with simulation-based verification.

storm

4.4 Software Tools and Frameworks

In Sections 2.5.3 and 2.6, we introduced the theoretical foundations of verification and synthesis techniques. These approaches have been implemented in various software tools that support the modeling, verification, and synthesis of stochastic systems including CAS. Here, we survey this rich ecosystem of tools, organizing our analysis by primary functionality while acknowledging the increasingly blurred boundaries between tool categories.

General-Purpose Probabilistic Model Checkers Several mature tools provide comprehensive verification capabilities for probabilistic models:

- **PRISM** [156]: The most widely used probabilistic model checker, supporting DTMC, CTMC, MDP, and PTA models with PCTL, CSL, and LTL properties. PRISM combines symbolic (MTBDD-based) and explicit state-space exploration with statistical approximation.
- **Storm** [87]: A high-performance probabilistic model checker offering multiple engine backends (sparse, DD, symbolic) and advanced algorithms for parametric models and multi-objective properties.
- **MRMC** [146]: A minimalistic model checker focused on time-bounded reachability analysis and CSL/CSRL model checking with particular emphasis on reward models.
- **Modest Toolset** [123]: An integrated framework for stochastic timed systems with compositional modeling and multiple analysis backends.
- **Möbius** [83]: A multi-formalism modeling and evaluation tool supporting stochastic Petri nets, stochastic process algebras, and Markov chains with both analytical and simulation-based solution techniques.

These tools provide solid foundations for verifying stochastic properties but generally lack specialized support for CAS-specific concerns like spatial embedding and large-scale agent populations.

Statistical Model Checking Tools Building upon the simulation-based verification approach discussed in Section 2.5.3, specialized SMC platforms have gained prominence:

- **PLASMA-Lab** [42]: A modular platform for SMC with plugin-based model support, distributed simulation, and rare event analysis.
- **UPPAAL SMC** [75]: Extends the UPPAAL real-time model checker with statistical simulation and verification of stochastic timed automata.
- **MultiVeStA** [195]: A distributed statistical analyzer supporting multiple simulation engines with property specification via MultiQuaTEx.
- **COSMOS** [20]: Specializes in rare event simulation for stochastic Petri nets with importance sampling.

SMC tools excel at analyzing complex models beyond the reach of exact methods but provide probabilistic rather than absolute guarantees, as outlined in our discussion of Statistical Model Checking in Section 2.5.3.

Parameter Synthesis Tools Related to the synthesis approaches discussed in Section 2.6, specialized tools focus on parameter identification and optimization:

- **PARAM** [117]: Performs symbolic parameter synthesis for DTMCs and CTMCs using rational function extraction.
- **ProPhESY** [85]: Implements various parameter synthesis techniques including sampling-based approaches, SMT solving, and region refinement.
- **PROPhESY** [54]: Focuses on synthesis for rule-based biochemical models with specialized abstractions.
- **PRISM-PSY** [190]: Extends PRISM with parameter synthesis capabilities based on region verification and refinement.
- **COMPASS** [43]: Targets parameter synthesis for safety-critical systems with fault tolerance properties.

Synthesis tools bridge verification and optimization but often restrict model classes or property specifications to maintain tractability, reflecting the fundamental challenges discussed in our treatment of parameter synthesis in Section 2.6.

Domain-Specific CAS Tools Several frameworks address specific CAS domains with tailored languages and analysis capabilities:

- **BioNetGen** [122] and **KaSim** [40]: Rule-based modeling platforms for biochemical systems with specialized simulation algorithms.

- **BioDiVinE** [23]: Parallel explicit-state model checker for biological systems with support for spatial extensions.
- **CARMA Eclipse Plug-in** [164]: Integrated environment for CARMA specification and analysis focusing on collective adaptive systems.
- **Breach** [97] and **S-TaLiRo** [11]: Falsification-based testing frameworks for cyber-physical systems with temporal logic specifications.
- **NetLogo** [205] with **BehaviorSpace**: Agent-based modeling environment with parameter sweeping capabilities.

Domain-specific tools provide intuitive abstractions and efficient algorithms but may lack formal verification guarantees or generalizability across domains.

Comparative Analysis A comprehensive comparison of tools across key dimensions is provided in Table 4.1. This comparison includes:

- **Modeling Formalism**: The primary mathematical representation and language support.
- **Verification Method**: Whether the tool uses exact, approximate, or statistical analysis.
- **Property Specification**: Supported temporal logics and specification languages.
- **Parameter Handling**: Capabilities for parametric analysis and synthesis.
- **Spatial Support**: Explicit handling of spatial dimensions and topologies.
- **Scalability**: Demonstrated system sizes that can be analyzed.
- **Extensibility**: API support and integration capabilities.
- **Active Development**: Current maintenance and community support.

Table 4.1 presents a comprehensive overview of these tools, highlighting their strengths across different modeling approaches and analysis techniques. As noted by reviewers, this comparison is quite revealing about the current landscape of tools for CAS analysis, though some notable tools like Möbius could be added to further enhance the comparison.

Distribution of Techniques Across Tools Mapping the landscape of tools against capabilities reveals specialization patterns and coverage gaps:

- **Exact model checking** dominates in PRISM, Storm, and Modest, with sophisticated symbolic data structures and algorithms for Markovian systems.
- **Statistical verification** is the core strength of PLASMA-Lab and UPPAAL SMC, with increasing adoption in previously exact-only tools.
- **Parametric analysis** and **region-based synthesis** are specialized capabilities in PARAM, ProPhESY, and PRISM-PSY.

	Modelling Language												Execution			Formal Analysis			
Tool	Agent-based	Boolean/Quantitative Networks	Biochemical Networks	Compartment-based	Hybrid System	Lattice-based	Markov Chains	Petri nets	Process Algebra	Rule-Based	State Charts	Continuous Semantics	Discrete Boolean Semantics	Stochastic Semantics	Model Checking	Parameter Synthesis	Runtime Verification / Monitoring	Static Analysis
BAM [174]				•					•					•				•
BETAWB [88]									•			•	•	•				•
BIOCHAM [55]										•		•	•	•	•	•	•	•
BIODIVINE [23]			•		•		•					•	•	•	•	•		•
BIONETGEN + BIOLAB [103]										•		•		•	•			
BIO-PEPA WB [63]									•			•		•	•			•
BOOLNET [176]		•											•	•				
BMA [28]		•											•		•			
BNS [98]		•											•		•			
BREACH [92]					•							•				•	•	
CARMA [33]							•		•					•	•			
COMPUCELL3D [199]	•					•								•				
COPASI [136]			•									•		•		•		
DREACH [150]					•							•			•	•		
FLAME [191]	•													•				
GINSIM [56]		•											•		•			
GREATSPN [16]								•				•	•	•	•			
IBM RATIONAL RHAPSODY [134]											•		•					
KASIM + KASA [72]										•		•		•				•
MATHWORKS SIMULINK [70]					•							•						
MELA [178]							•		•					•	•			
MÖBIUS [83]			•				•	•	•			•	•	•	•	•		
PALOMA [105]							•		•					•	•			
PATHWAY LOGIC [100]								•					•		•			
PLASMA-Lab [42]							•							•	•	•	•	
PRISM [157]							•							•	•	•		
ROVERGENE [1]					•								•		•	•		•
SCEL [81]									•					•	•			•
SNOOPY + MARCIE [127]								•				•	•	•	•			
SPIM [183]									•					•				
S-TALIRO [11]					•							•		•		•	•	
STORM [87]							•							•	•	•		
REPAST [180]	•													•				
UPPAAL SMC [74]					•		•				•	•		•	•	•	•	

Table 4.1: *Summary of the features for selected tools in modelling and analysing complex systems [27]*

- **Hybrid system analysis** is addressed by UPPAAL SMC, Breach, and S-TaLiRo with complementary approaches.
- **Spatial modeling and analysis** remain relatively fragmented across domain-specific tools without standardized approaches.
- **Large-scale agent systems** are primarily handled through statistical means or fluid approximations without formal guarantees.

Most tools address specific fragments of the modeling-analysis-synthesis pipeline, requiring manual integration for end-to-end workflows. This fragmentation highlights a key opportunity for frameworks like Sibilla, which aim to provide more comprehensive coverage of the CAS analysis pipeline.

4.4.1 Novel Contributions of This Work

The principal contribution of this thesis is the design and implementation of a **general-purpose, extensible synthesis module** integrated within the Sibilla framework — a simulation and statistical verification platform tailored to the analysis of stochastic Collective Adaptive Systems (CAS). Rather than being limited to a fixed synthesis pipeline, the module is designed as a first-class architectural component that naturally fits within the modeling-analysis-verification loop, enabling the synthesis of parameters satisfying quantitative specifications.

The contribution is articulated in the following key innovations:

- **Unified synthesis-analysis workflow.** The synthesis module leverages Sibilla's existing simulation and statistical model checking (SMC) infrastructure, enabling seamless transitions between model specification, performance evaluation, and parameter synthesis — all within a consistent and self-contained toolchain.
- **Modular and extensible architecture.** While the current implementation supports surrogate-based optimization strategies, the architecture is explicitly designed to accommodate alternative synthesis paradigms — such as symbolic synthesis, counterexample-guided inductive synthesis (CEGIS), or constraint-based approaches — with minimal changes to the core framework. In the current version, the modularity enables the combination of different surrogate models, acquisition functions, and optimization heuristics, fostering comparative experimentation within this synthesis strategy class.
- **Surrogate-based synthesis for CAS.** The implemented synthesis loop exploits the smooth, probabilistic behavior typical of collective dynamics by using regression-based surrogate models and adaptive sampling. This allows for efficient exploration of high-dimensional parameter spaces with a limited number of simulations, while guiding the search toward regions of interest with respect to a quantitative objective.
- **Statistical validation of synthesized configurations.** To ensure reliability despite the approximation introduced by surrogate models, candidate solutions are subject to a final validation phase via statistical model checking. This step provides empirical guarantees with quantifiable confidence bounds, preserving the soundness required for application in safety-critical settings.

By embedding this synthesis module within a framework specifically designed for stochastic CAS, this work delivers a reusable and extensible foundation for parameter synthesis in complex adaptive systems. It enables controlled experimentation, reproducibility, and architectural scalability in line with future research and application needs.

Significance and Impact : The significance of this work lies in its contribution to the architectural integration of synthesis capabilities within a general-purpose modeling and verification platform. By embedding a parameter synthesis engine into the Sibilla framework, this thesis provides a foundation for the scalable and extensible design of adaptive systems governed by stochastic dynamics.

The practical impact is twofold. First, it provides a scalable toolchain for parameter synthesis in CAS, supporting applications with non-trivial quantitative requirements. Second, it fosters research reproducibility and future extensibility, thanks to a clean architectural separation between modeling, simulation, verification, and synthesis.

In broader terms, this work contributes to the long-term goal of unifying modeling, analysis, and synthesis within a single extensible platform — a crucial step toward the practical adoption of formal methods in complex, safety-critical, and adaptive system engineering.

CHAPTER 5

Sibilla

"The Sibyl, speaking with a hundred tongues, A hundred mouths, and voices prophetic"

Virgil, Aeneid, Book VI

THIS chapter presents the architecture and key components of Sibilla, a modular framework designed for SMC and analysis of Collective Adaptive Systems (CAS). Sibilla's architecture supports flexible modelling, efficient simulation, and comprehensive analysis of complex stochastic systems through a layered design that separates concerns while maintaining cohesive integration between components. The framework follows a three-tier architecture consisting of a back-end layer that provides core modelling and analysis capabilities, a runtime layer that coordinates system operations, and a front-end layer that enables user interaction through multiple interfaces. This architectural approach allows Sibilla to support numerous modelling formalisms while providing a consistent environment for simulation, analysis, and synthesis tasks. Throughout this chapter, we explore each layer in detail, beginning with an overview of Sibilla's high-level architecture and the interactions between its components. We then examine the back-end layer's support for various modelling approaches, from population-based models to agent-oriented specifications. The chapter covers Sibilla's simulation capabilities, including its SMC features and measure sampling techniques. We also discuss the analysis components that enable verification of system properties and assessment of temporal behaviours. The synthesis module, which supports parameter optimisation and model adjustment, is then presented. Finally, we explore the front-end interfaces that make Sibilla's capabilities accessible to users through command-line and programmatic interfaces. This architectural design enables Sibilla to serve as a comprehensive framework for studying CAS, providing researchers and practitioners with the tools to model, simulate, analyse, and optimise

complex stochastic systems. The modular nature of the architecture also facilitates the framework's evolution and extension to accommodate new modelling formalisms and analysis techniques as the field advances.

5.1 Sibilla Architecture

Sibilla adopts the Model-View-Controller (MVC) architectural pattern, a proven design approach that ensures the separation of concerns while maintaining system cohesion. This architectural decision and strict adherence to SOLID design principles establish a robust foundation for the framework's implementation. The system is developed in Java 21, leveraging the language's advanced features and robust ecosystem.

The framework's architecture comprises three layers: a back-end layer implementing core computational functionality, a front-end layer providing multiple user interaction interfaces, and a runtime layer serving as the controller. This tripartite structure aligns with the MVC pattern, where the back-end represents the Model (managing data and business logic), the front-end implements the View (handling user interaction through Java API, Shell, and Python API), and the runtime layer functions as the Controller (coordinating system operations and managing data flow).

The back-end layer consists of four primary components: Models, Simulation, Analysis, and Synthesis. The Models component, in particular, supports multiple specification languages, including Language of Interactive Objects (LIO), Population Models (LPM), YODA, and SLAM, each designed to address specific modelling requirements. The runtime layer orchestrates the interaction between these back-end components and the various front-end interfaces, enabling seamless integration of the framework's capabilities.

This implementation approach, grounded in established software engineering principles, enables Sibilla to maintain high cohesion while minimising coupling between components. The framework's deployment flexibility is enhanced through Docker containerisation and Gradle build system integration, enabling consistent execution across diverse operating environments without manual configuration requirements—the following sections detail each architectural layer, examining their specific implementations and interactions within the broader framework.

In Figure 5.1, we present a comprehensive view of Sibilla's architecture, illustrating the relationships between its components and the hierarchical organisation of its implementation.

5.1.1 Sibilla front-end

These interfaces allow users to interact with Sibilla through multiple approaches, catering to different user needs and expertise levels. The framework provides three main interaction methods: a command-line interface, a Python front-end, and Java APIs. Additionally, containerised deployment options ensure consistent execution environments across different platforms.

Sibilla shell This command-line interpreter provides a domain-specific script language for straightforward interaction with Sibilla. The shell implementation follows established patterns from popular command-line tools, making it intuitive for users familiar

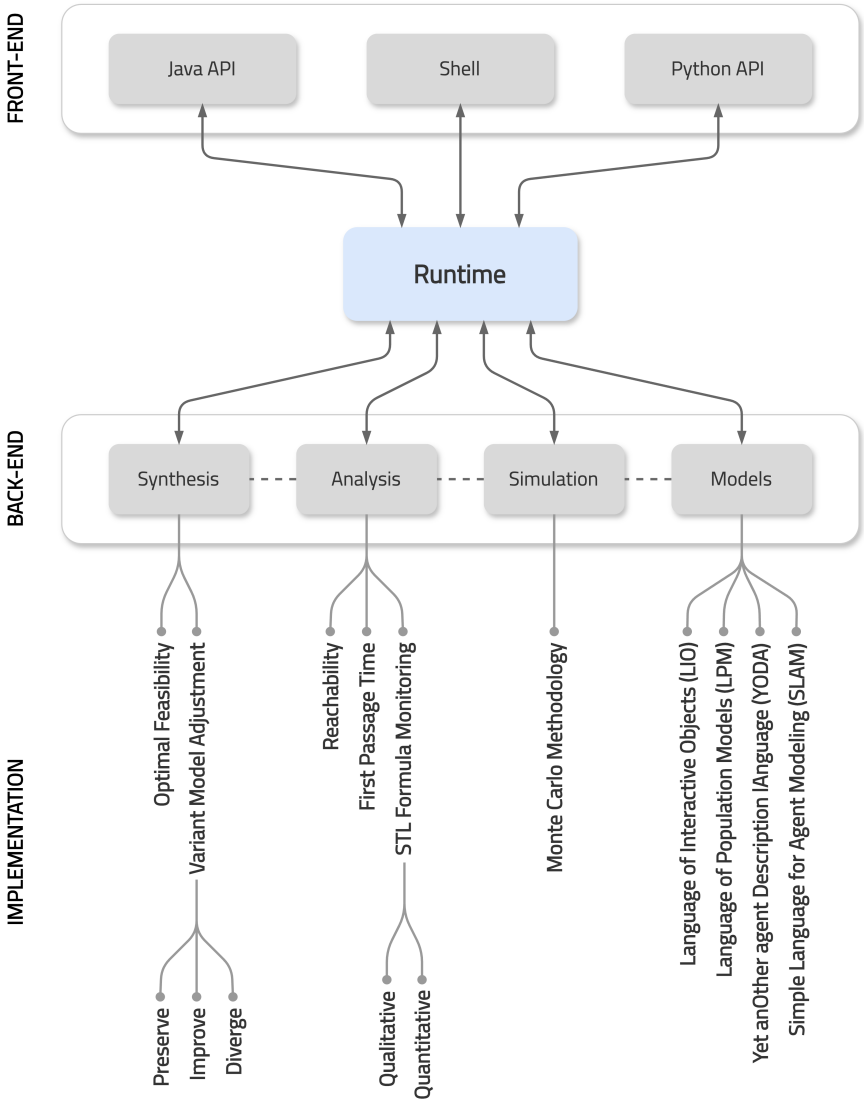


Figure 5.1: *Sibilla Framework Architecture*

with terminal interfaces.

The shell supports two operational modes: interactive and batch. In interactive mode, users can input commands directly and receive immediate feedback, making it ideal for exploratory analysis and rapid prototyping. The batch mode enables automation through script execution, which is particularly useful for reproducible experiments and automated testing pipelines. A comprehensive list of all available shell commands and their detailed usage is provided in Appendix B.

The following script demonstrates a typical workflow for model simulation and data collection, showcasing key shell commands:

```
module "population"                    /* Load population model */
load "examples/seir/seir.pm"           /* Load system specification */
init "initial_2"                       /* Set initial configuration */
```

```
add all measures                                    /* Add measures to collect */
deadline 100                                        /* Set simulation deadline */
dt 0.5                                              /* Set time interval to collect data */
replica 100                                         /* Set number simulation replica */
simulate                                            /* Start simulation */
save output "./results" prefix "seir" postfix "__"  /* Save results on file */
```

Python Front-end Python's widespread adoption in data science and scientific computing makes it ideal for Sibilla's high-level interface. The Python front-end, built upon the *Pyjnius* library [200], provides seamless integration between Python's ecosystem and Sibilla's Java-based core functionality. This integration preserves performance while offering Python's expressiveness and extensive library ecosystem.

The front end enables users to leverage popular data science tools such as Matplotlib[1] for visualisation, pandas for data manipulation, and scikit-learn for advanced analysis. A key advantage is the ability to use Sibilla within web-based Interactive Development Environments (IDEs), particularly Jupyter notebook/lab [149] and Google Colaboratory [30], eliminating local installation requirements.

The following Python code illustrates the straightforward API for model simulation and visualisation:

```
sr = sibilla.SibillaRuntime()
sr.load_module('population')
sr.load_from_file('examples/seir/seir.pm')
sr.add_all_measures()
sr.set_deadline(100)
sr.set_dt(0.5)
sr.set_replica(100)
res = simulation_result = sr.simulate("initial_2")
res.plot(show_sd = True)
```

Cloud-Based Development Environments The Python front-end's compatibility with cloud-based notebooks significantly enhances accessibility and collaboration. Users can run Sibilla simulations directly in Jupyter notebooks or Google Colab without local installation, making it particularly suitable for educational settings and collaborative research. These platforms provide persistent storage, computational resources, and built-in version control integration, facilitating reproducible research workflows.

Java APIs As a Java-based framework (requiring Java 21 at the time of writing), Sibilla provides comprehensive Java APIs for deep integration into existing Java applications. The framework's core functionality is exposed through the `SibillaRuntime` class, which implements a controller pattern to manage simulation configuration and execution.

The following Java code demonstrates the API's parallel structure to both shell and Python interfaces:

```
SibillaRuntime sr = new SibillaRuntime();
sr.loadModule("population");
sr.load("examples/seir/seir.pm");
sr.setConfiguration("initial_2");
sr.addAllMeasures();
sr.setDeadline(100);
sr.setReplica(100);
sr.setDt(0.5);
```

[1] `https://matplotlib.org`

```
sr.simulate("res");
sr.save("./results", "seir", "__");
```

Containerised Deployment Sibilla provides an official Docker configuration to ensure reproducibility and simplified deployment across different environments. This containerised solution bundles all required dependencies and integrates a JupyterLab environment, enabling immediate access to Sibilla's Python interface through an interactive web-based development environment.

The Docker deployment can be initiated through two methods, depending on the user's requirements and environment configuration. The primary approach utilises Docker Compose, which automatically handles container orchestration.

```
docker-compose up
```

This command, when executed in the directory

```
/shell/src/dist/scripts/sibilla_docker
```

Initialising a container with all necessary components. Upon successful deployment, users can access the JupyterLab interface through their web browser at

```
localhost:8888
```

For cases where the default port configuration is unavailable or when users prefer manual container management, an alternative deployment method is provided:

```
docker build -t sibilla_docker .
docker run -p 8888:8888 -d sibilla_docker
```

The first command constructs a Docker image containing the Sibilla environment, while the second initiates a container instance, mapping the container's JupyterLab port to the host system. This approach offers greater flexibility in port configuration and container management.

The containerised environment provides several advantages for users:

- Immediate access to a fully configured JupyterLab environment
- Consistent execution environment across different platforms
- Pre-installed dependencies and correct Java version

Within the Docker container, users can leverage all features of Sibilla's Python interface as described in the Python Front-end section, benefiting from the combination of JupyterLab's interactive development features and Sibilla's simulation capabilities. This containerised approach particularly benefits teams requiring consistent environments across different development machines and deployment scenarios.

Figure 5.2 illustrates the three-tiered front-end architecture of Sibilla, along with its deployment capabilities.

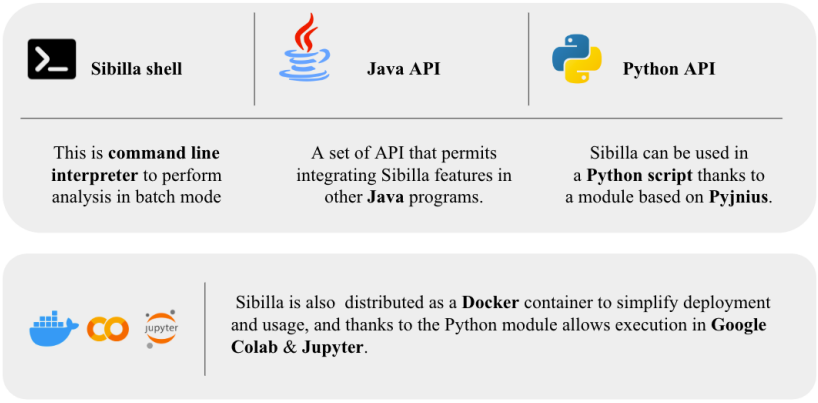


Figure 5.2: *Sibilla front-end main interfaces: a command line interpreter (Sibilla shell) for batch processing and interactive analysis, a comprehensive Java API for seamless integration with existing Java applications, and a Python API powered by Pyjnius that enables script-based interaction. The containerised deployment option, implemented through Docker, further enhances accessibility by enabling execution in cloud-based environments such as Google Colab and Jupyter, eliminating installation complexities and ensuring consistent behaviour across different platforms.*

5.1.2 Sibilla back-end

The Sibilla back-end forms the core foundation of the framework, implemented in Java 21 and following SOLID design principles. This layer consists of four components: *Models*, *Simulation*, *Analysis*, and *Synthesis*, represented as grey boxes in the bottom part of Figure 5.1.

Models This component provides interfaces and classes that can be used to describe a *Stochastic Process*. The component *Models* also provides utility classes for handling specific features of the considered models. These classes provide the base on which specification languages can be integrated into Sibilla. Four specification languages are already integrated, described below and represented by red blocks in Figure 5.1. The ability to include other specification languages by implementing appropriate classes makes our framework *extensible*.

Simulation When a system composed of many entities is considered, exact numerical analysis is often hard or even impossible. For this reason, Sibilla provides a set of classes that support the simulation of stochastic processes. These classes permit sampling a *path* from a model. Moreover, classes are provided to extract *measures* from a *path* and collect statistical information. Simulation samplings and statistical analyses rely on the widely used and well-known API *The Apache Commons Mathematics*.

Analysis This component provides a set of analysis techniques that can be used to extract data collected from the Sibilla simulator to assess the quantitative properties of analysed systems. These analyses rely on *statistical inference techniques*. Currently, the integrated techniques are the computation of *first-passage-time*, *reachability analysis* and *monitoring of properties*.

Synthesis This component focuses on automated system design and optimisation. It provides tools for finding optimal system configurations or parameters based on specified objectives and constraints. The main synthesis features are *Optimal Feasibility* and *Variant Model Adjustment*.

5.1.3 Sibilla Runtime

The role of this layer is to coordinate and control the interactions between the front-end and back-end. *Runtime layer* plays, the rol,e of the *controller* in the standard MVC pattern and it ,is used by all the components in the Sibilla *front-end*.

The Runtime implementation, encapsulated in the `SibillaRuntime` class, provides a comprehensive interface for managing simulation configurations, execution, and data collection.

Core functionalities provided by the Runtime layer include module management, parameter configuration, measure handling, and simulation execution control. The layer supports various simulation modes, including standard simulation runs, formula evaluation, and trace generation. Data collection and persistence are handled through dedicated mechanisms, supporting both CSV output and in-memory storage of simulation results.

The Runtime layer's implementation emphasises extensibility through its modular design. New simulation capabilities can be integrated by implementing appropriate command handlers and registering them with the command adapter. This architecture ensures the framework can evolve to accommodate new requirements while maintaining backward compatibility with existing functionality.

Error handling is implemented through a dedicated exception hierarchy, which provides detailed feedback on configuration and execution errors. This robust error handling ensures that users receive clear feedback when issues arise during simulation setup or execution.

The Runtime layer's careful balance of flexibility and control makes it an effective mediator between the framework's user interfaces and computational core, fulfilling its role as the controller in Sibilla's MVC architecture.

5.2 Models Definition

Sibilla's modular and versatile architecture enables the integration of different modelling paradigms, from low-level to high-level models, making it a powerful tool for analysing and simulating complex systems. While it is primarily designed for SMC, Sibilla can accommodate various models due to its flexible structure.

As defined in Section 2.7, Collective Adaptive Systems (CAS) present unique modelling challenges due to their massive scale, heterogeneity, autonomy, adaptivity, emergence, open-endedness, and spatial embedding characteristics. The high-level modelling approaches implemented in Sibilla are specifically designed to address these challenges, providing the expressiveness and flexibility needed to capture the complex dynamics of CAS that would be cumbersome or infeasible to model using low-level approaches alone.

The following subsections describe the main modelling approaches implemented and fully functional within Sibilla, each representing different types of CAS and demon-

strating how Sibilla's architecture supports the diverse requirements outlined in Section 2.7. These modelling languages align with the tool's focus on SMC and analysis, providing practical solutions for the analytical challenges identified for CAS.

For a comprehensive reference of the formal syntax and specification languages used by each modelling approach, readers are directed to Appendix A.

5.2.1 Language of Population Model (LPM)

The Language of Population Model (LPM) is a high-level formalism designed to describe the behaviour and dynamics of population systems, based on the CTMC. This formalism allows for modelling complex interactions within a population, where each entity follows probabilistic rules.

Definition 37 (Population Model)
A population model is a tuple $\mathcal{M} = (\mathbf{S}, \mathbf{X}, \mathcal{D}, x_0, \mathcal{T})$ *where:*

- $\mathbf{S} = \{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_i, \ldots, \boldsymbol{s}_n\}$ *is the set of species (i.e., the different agent states),*
- $\mathbf{X} = (X_1, \ldots, X_i, \ldots, X_n)$ *is a vector of population variables, counting the number of entities of each kind, and taking values in* $\mathcal{D} \subseteq \mathbb{N}^n$*,*
- $\mathcal{D} = \bigtimes_{i=1}^{n} (\mathcal{D}_i)$ *is the (countable) state space,*
- $x_0 \in \mathcal{D}$ *is the initial state of the model,*
- $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_m\}$ *is a set of transition rules, describing how the system state can change. Each rule is a tuple* $\tau = (\ell, \mathbf{u}, \mathbf{g}, \lambda)$*:*
  - ℓ *is the label of the rule,*
  - $\mathbf{u}$ *(used), is a vector encoding how many agents are consumed in the reaction,*
  - $\mathbf{g}$ *(generated), is a vector encoding how many agents are produced in the reaction,*
  - $\lambda = \lambda(\mathbf{X})$ *is the rate function* ($\mathcal{D} \to \mathbb{R}_{\geqslant 0}$)*, associating to each reaction the rate of an exponential distribution, depending on the global state of the model.*

Each reaction can be seen as:

$$\tau \;:\; \mathbf{u}_1\boldsymbol{s}_1 + \ldots + \mathbf{u}_n\boldsymbol{s}_n \xrightarrow{\lambda(\mathbf{X})} \mathbf{g}_1\boldsymbol{s}_1 + \ldots + \mathbf{g}_n\boldsymbol{s}_n$$

The essential characteristic of a population model is its basis in CTMC, which incorporates real-time information into the states. This means that each state transition occurs according to an exponential distribution, reflecting the stochastic nature of the processes within the population. The LPM framework extends theCTMC by including multiple interacting entities and complex transition rules, making it a powerful tool for

modelling and analysing population dynamics in various applications.

5.2.2 Language of Interactive Object (LIO)

A Language of Interactive Object (LIO) builds upon the foundation of a DTMC by incorporating multiple interacting agents within the system. This high-level formalism allows for the modelling of complex multi-agent systems where each agent operates under stochastic rules influenced by their environment and interactions with other agents.

Definition 38 (Language of Interactive Object (LIO))
A Language of Interactive Object (LIO) is a tuple $\mathcal{M} = (\mathcal{S}, \boldsymbol{P}, AP, L, s_0, N)$*, where:*

- $\mathcal{S}$ *is a finite, non-empty set of states*
- s_0 *is the initial state*
- $\mathbf{P} : \mathcal{S}^N \times \mathcal{S}^N \to [0,1]$ *is a transition probability matrix such that* $\sum_{\mathbf{s}' \in \mathcal{S}^N} \mathbf{P}(\mathbf{s}, \mathbf{s}') = 1, \forall \mathbf{s} \in \mathcal{S}^N$*, where* $\mathbf{P}(\mathbf{s}, \mathbf{s}')$ *denotes the probability of transitioning from state vector* $\mathbf{s}$ *to state vector* $\mathbf{s}'$
- $L : \mathcal{S} \to 2^{AP}$ *is a labeling function mapping each agent state,* $s \in \mathcal{S}$*, to the set* $L(s) \subseteq AP$ *of atomic propositions* AP*, that hold true in* s
- N *represents the number of agents in the system, and each state in* $\mathcal{S}^N$ *is a vector representing the states of all* N *agents.*

The key characteristic of an LIO is that it extends the Markov property to a multi-agent context. Each agent's state transition is influenced not only by its current state but also by the states of the other agents in the system. Despite the complexity introduced by multiple agents, the process retains the memoryless property at the global level.

$$\begin{aligned} P(\mathbf{X}_{t+1} = \mathbf{s}_j \mid \mathbf{X}_t = \mathbf{s}_i, \mathbf{X}_{t-1} = (\mathbf{s}_i)_{t-1}, \ldots, \mathbf{X}_0 = \mathbf{s}_0) \\ = P(\mathbf{X}_{t+1} = \mathbf{s}_j \mid \mathbf{X}_t = \mathbf{s}_i) \\ = P_{\mathbf{ij}} \end{aligned}$$

Here, $P_{\mathbf{ij}}$ denotes the probability that the system transitions from state vector $\mathbf{s}_i$ to state vector $\mathbf{s}_j$ in one time step. The matrix $\mathbf{P} = (P_{\mathbf{ij}}), \mathbf{i}, \mathbf{j} \in \mathcal{S}^N$, is called the one-step transition matrix, with each row summing to one, reflecting a complete probability distribution for each state vector.

The defining Markov property for LIOs can be succinctly captured by the principle that the future states depend only on the current global state vector and not on the preceding states. Formally, if the current state vector $\mathbf{X}_n = \mathbf{i}$ is known, then the future evolution of the system is stochastically independent of the past state vectors $\{\mathbf{X}_0, \ldots, \mathbf{X}_{n-1}\}$. This characteristic facilitates the modelling and analysis of complex multi-agent systems under the probabilistic framework.

5.2.3 Yet anOther agent Description lAnguage (YODA)

YODA (Yet anOther agent Description lAnguage) is a formalism for modelling and analysing Multi-Agent Systems (MASs) that extends the concepts of DTMCs and Interpreted Systems. YODA addresses limitations in modelling complex multi-agent scenarios by explicitly representing the environment and agent-environment interactions. The formalism is built upon several key components, beginning with defining a YODA agent.

Definition 39 (YODA Agent)
A YODA agent is a tuple $A = (N, \sigma)$, where:

- $N \in AgName$ *is the agent name*
- $\sigma = (\sigma_c, \sigma_f, \sigma_\omega) \in \Sigma$ *is the agent store, where:*
  - $\sigma_c : Attr_C \to Val$ *associates conditions to values*
  - $\sigma_f : Attr_F \to Val$ *associates features to values*
  - $\sigma_\omega : Attr_\Omega \to Val$ *associates observations to values*

Here, $Attr_C$, $Attr_F$, and $Attr_\Omega$ are disjoint sets of attributes representing conditions, features, and observations respectively, such that $Attr = Attr_C \uplus Attr_F \uplus Attr_\Omega$.

The agent store σ plays a crucial role in defining the state of an agent. Conditions are attributes under the direct control of the agent, features are environmental properties controlled by the environment, and observations are used to perceive the surrounding environment. This structure allows for a clear separation of internal state, external influences, and perceptual information.

To model the dynamic behaviour of agents, YODA defines two key functions that govern action selection and execution.

Definition 40 (YODA Agent behaviour)
Two functions define the behaviour of a YODA agent:

- $\beta \in B = \Sigma_C \times \Sigma_\Omega \to Dist(ActName)$
- $\alpha \in A = ActName \to \Sigma_C \times \Sigma_\Omega \to Dist(\Sigma_C)$

Where β selects the next action to execute based on current conditions and observations, and α describes the probabilistic effects of actions on the agent's conditions.

The function β captures the decision-making process of an agent, mapping its current conditions and observations to a probability distribution over possible actions. The function α then models the outcomes of these actions, potentially introducing nondeterminism in the agent's behaviour.

To complete the agent specification, YODA introduces the concept of an agent defi-

nition.

Definition 41 (YODA Agent Definition)
An agent definition Δ is a function that associates each agent name with its behaviour:

$$\Delta : AgName \to B \times A$$

For any $N \in AgName$ such that $\Delta(N) = (\beta, \alpha)$, we use Δ^N_β to denote β and Δ^N_α to denote α.

The agent definition provides a way to specify different types of agents within a system, each with its behaviour functions. This allows for the modelling of heterogeneous agent populations.

Moving from individual agents to their environment, YODA provides a structured way to represent the context in which agents operate.

Definition 42 (YODA Environment)
A YODA environment E is a function associating to each agent name $A \in AgName$:

- *A feature dynamic function $F : \Sigma_C \times \Sigma_F \to \Sigma_F$*
- *An observation function $\Omega : Ag^* \times Ag \to Dist(\Sigma_\Omega)$*

where $Ag = AgName \times \Sigma$ denotes the set of YODA agents.

The feature dynamic function F models how an agent's features evolve based on current conditions and features. The observation function Ω represents how an agent perceives its environment, potentially with some uncertainty, given the current system state.

To model the overall state of a multi-agent system, YODA introduces the concept of a system configuration.

Definition 43 (YODA System Configuration)
A YODA system configuration $S \in Ag^*$ consists of a sequence of agents. For $S \in Ag^*$:

- *$|S|$ denotes the number of agents in S*
- *$S[i]$ denotes the agent in position i, where $1 \leqslant i \leqslant |S|$*
- *S/i denotes the configuration obtained from S by removing the agent in position i*

This representation allows for a flexible description of the system state, accommodating varying numbers of agents and their states.

Finally, to capture the evolution of the entire system over time, YODA defines a

DTMC based on the system configurations.

Definition 44 (**YODA System behaviour**)
Given agent definitions Δ and an environment E, the behaviour of YODA configurations is described by a DTMC $M_E^\Delta = (Ag^*, P_E^\Delta)$, where $P_E^\Delta : Ag^* \to Dist(Ag^*)$ is the probability transition function.

The transition function P_E^Δ captures the probabilistic evolution of the entire system, considering the behaviours of individual agents, their interactions, and the influence of the environment.

YODA's formalism thus provides a comprehensive framework for modelling complex MASs, incorporating probabilistic behaviours, explicit environment representation, and agent-environment interactions while maintaining the Markovian property at the global level. This approach allows for detailed analysis and simulation of a wide range of multi-agent scenarios, from robotic swarms to social systems.

5.2.4 Stochastic Agent-based Modelling Language (SLAM)

SLAM, an agent-oriented specification language, provides a high-level formalism for modelling complex multi-agent systems. A SLAM system consists of agents interacting to reach local and global goals. When operating inside the enclosing environment, these agents are designed to run and execute actions parallelly. Agents are equipped with attributes and communicate with each other via message exchange.

SLAM builds on the low-level formalism of Stochastic Hybrid Automata (SHA) by introducing higher-level constructs to specify agent behaviours, interactions and goals. While SHA focuses on the continuous and discrete dynamics of hybrid systems, SLAM enhances this model by providing a framework for agent-based simulation and analysis.

Definition 45 (**SLAM System**)
A SLAM system $\mathcal{M}$ is a tuple $(\mathcal{A}, \mathcal{E}, \mathcal{M}sg, \mathcal{C}, \mathcal{P})$, where:

- *$\mathcal{A}$ is a set of agents, each defined by a unique identifier, attributes, behaviours, and message handlers,*
- *$\mathcal{E}$ is the environment in which agents operate, characterised by its attributes and dynamics,*
- *$\mathcal{M}sg$ is the set of messages that agents can exchange to interact with each other,*
- *$\mathcal{C}$ is the set of communication channels through which messages are transmitted,*
- *$\mathcal{P}$ is a set of properties that specify the goals and constraints of the system.*

Agents in SLAM are equipped with attributes that define their state, behaviours

that govern their actions, and message handlers that manage their interactions. An agent's state's evolution is influenced by its internal dynamics and interactions with other agents and the environment.

Definition 46 (**SLAM Agent**)
A SLAM agent $a \in \mathcal{A}$ is defined by a tuple $(id, attr, beh, msgH)$, where:

- *id is a unique identifier for the agent,*
- *attr is a set of attributes that describe the agent's state,*
- *beh is a set of behaviours that determine the agent's actions,*
- *msgH is a set of message handlers that process incoming messages.*

The behaviour of a SLAM agent is defined by a set of rules and states that dictate how the agent acts over time. The behaviour is specified as a finite state machine, where each state defines possible actions and transitions based on the agent's attributes and incoming messages.

Definition 47 (**Agent behaviour**)
An agent's behaviour beh is described by:

- ***States**: Each agent has a finite set of states S.*
- ***Initial State**: An initial state $s_0 \in S$ where the agent starts.*
- ***Transitions**: State transitions occur based on conditions evaluated on the agent's attributes and received messages.*
- ***Actions**: In each state, the agent can perform actions such as updating its attributes, sending messages, or changing states.*

Example 5.2.1 (**Agent behaviour example**)
For example, an agent's behaviour can be described as:

- **State**: "Searching"
  - **Condition**: If the goal is not found, continue moving.
  - **Action**: Update position based on movement dynamics.
- **State**: "Signaling"
  - **Condition**: If the goal is found, change state to signaling.
  - **Action**: Send a signal message to other agents.

Message handling in SLAM allows agents to communicate and coordinate their actions. Each agent has a set of message handlers that define how it processes incoming

messages and what actions it takes in response.

Definition 48 (**Message Handling**)
Message handling msgH for an agent involves:

- ***Message Types**: A set of messages $\mathcal{M}sg$ that the agent can send or receive.*
- ***Handlers**: For each message type, a handler specifies:*
  - ***Condition**: The condition under which the message is processed (e.g., when received from a specific agent or when a certain attribute value is met).*
  - ***Actions**: Actions to perform upon receiving the message, such as updating attributes, changing states, or sending a response.*

Example 5.2.2 (**Message Handling**)
For example, a message handler could be:

- **Message**: "Goal Found"
  - **Condition**: Received from any agent.
  - **Actions**: Update local knowledge to mark the goal's location, send acknowledgment.

The dynamics of SLAM agents are influenced by stochastic processes, modeled using the same stochastic differential equations (SDEs) as in SHA, with additional mechanisms for agent interactions and message handling.

Definition 49 (**SLAM Execution**)
A stochastic process $(X(t), A(t)) \in \mathbb{R}^N \times \mathcal{A}$ is called a SLAM execution if it satisfies the following conditions:

- *Each agent $a \in \mathcal{A}$ evolves according to an SDE within its current state:*

$$dX_a(t) = f_a(X_a(t), t)dt + g_a(X_a(t), t)dB_t,$$

  where B_t is a standard Brownian motion.
- *Agent interactions and message exchanges occur at discrete event times $\{\tau_n\}$, where $\tau_{n+1} = \inf\{t > \tau_n : \text{event occurs at } t\}$.*
- *The events and state transitions are governed by the system's dynamics and the agents' behaviours.*

SLAM systems are analysed using properties that capture both local behaviours of individual agents and global behaviours of the entire system.

Definition 50 (**SLAM Property**)
A SLAM property $\phi \in \mathcal{P}$ is a logical statement that can be evaluated on a SLAM execution. Properties are categorised as:

- *Local properties, which describe the behaviour of individual agents,*
- *Global properties, which describe the collective behaviour of the system.*

Examples of SLAM properties include reachability (the probability that a specific state is reached within a time bound), safety (the likelihood that a particular state is never reached), and liveness (the likelihood that a specific state is eventually reached).

The SLAM framework implemented in Sibilla shares significant conceptual similarities with the Markovian Agent Models (MAMs) proposed by Bobbio, Bruneo, Cerotti, and Gribaudo [31]. Both approaches aim to describe complex systems composed of spatially distributed agents whose behaviour evolves according to a continuous-time stochastic semantics. In MAMs, each agent is modeled as a finite-state CTMC whose infinitesimal generator combines a local dynamics component and an interaction-induced component, the latter being modulated by a spatially dependent perception function. Analogously, SLAM provides mechanisms for modeling agents embedded in a spatial domain, where interactions are governed by distance-based perception ranges and rule-based transitions. While SLAM adopts a simulation-oriented approach—based on a rule-based specification language—rather than direct numerical solution, it retains the key modeling constructs of MAMs: spatial awareness, local stochastic behaviour, and indirect interactions via emitted messages. This correspondence highlights SLAM as an executable modeling framework aligned with the MAM paradigm, particularly effective in scenarios where the dimensionality of the system precludes analytical tractability and necessitates simulation-based verification.

In summary, SLAM extends SHA by providing a higher-level formalism for modelling, simulating, and analysing stochastic multi-agent systems. By building on the foundations of SHA, SLAM leverages stochastic differential equations for continuous dynamics while incorporating mechanisms for discrete events, agent interactions, and message handling. This makes SLAM a powerful tool for studying complex systems with interacting agents in uncertain environments.

Example 5.2.3 (SEIR Population Model)
The SEIR (Susceptible-Exposed-Infected-Recovered) model is a classic epidemiological model that can be expressed as a Population Model using the LPM formalism. Let $\mathcal{M}_{\text{SEIR}} = (\mathbf{S}, \mathbf{X}, \mathcal{D}, x_0, \mathcal{T})$ be the SEIR model, where:

- $\mathbf{S} = \{S, E, I, R\}$ is the set of species, representing Susceptible, Exposed, Infected, and Recovered individuals, respectively.
- $\mathbf{X} = (X_S, X_E, X_I, X_R)$ is the vector of population variables.
- $\mathcal{D} = \mathbb{N}^4$ is the state space.
- $x_0 = (99, 0, 1, 0)$ is the initial state, representing 99 susceptible and one infected individual.
- $\mathcal{T} = \{\tau_{\text{exposure}}, \tau_{\text{infection}}, \tau_{\text{recovery}}\}$ is the set of transition rules:
  - $\tau_{\text{exposure}} = (\text{"exposure"}, (1,0,1,0), (0,1,1,0), \lambda_{\text{exposure}})$ where $\lambda_{\text{exposure}}(\mathbf{X}) = X_S \cdot X_I \cdot 10.0 \cdot 1.00$
  - $\tau_{\text{infection}} = (\text{"infection"}, (0,1,0,0), (0,0,1,0), \lambda_{\text{infection}})$ where $\lambda_{\text{infection}}(\mathbf{X}) = X_E \cdot 2.00$
  - $\tau_{\text{recovery}} = (\text{"recovered"}, (0,0,1,0), (0,0,0,1), \lambda_{\text{recovery}})$ where $\lambda_{\text{recovery}}(\mathbf{X}) = X_I \cdot 0.5$

The transition rules can be expressed in the reaction notation as:

$$\tau_{\text{exposure}} : S + I \xrightarrow{X_S \cdot X_I \cdot 10.0 \cdot 1.00} E + I$$
$$\tau_{\text{infection}} : E \xrightarrow{X_E \cdot 2.00} I$$
$$\tau_{\text{recovery}} : I \xrightarrow{X_I \cdot 0.5} R$$

This SEIR model captures the dynamics of disease spread in a population, where individuals transition from susceptible to exposed, to infected and finally to recovered states.

5.3 Simulating Models

Simulation is the foundation of SMC, where Monte Carlo methods are used to avoid the state explosion problem. SMC estimates the probability that a system satisfies a property using several statistically independent simulation traces of an executable model. It allows for the estimation of various system measures and properties over time, providing insights into the system's behaviour under different conditions.

Before explaining what the simulation consists of, it is therefore useful to reiterate the definition of trajectory previously set out in Defintion 6

Given a stochastic model $\mathcal{M}$ and an initial state X_{init}, a trajectory w is a sequence of

state-time pairs:

$$w = (X_0, t_0), (X_1, t_1), \ldots, (X_n, t_n)$$

where $X_0 = X_{\text{init}}$, $t_0 = 0$, and for each $i > 0$, X_i is the state of the system at time t_i, with $t_i < t_{i+1}$.

In stochastic models, the behaviour of a system over time is represented through traces or trajectories, which capture the sequence of states through which the system moves at specific points in time. These states are often multidimensional, incorporating different aspects of the system. Each dimension, or aspect, can be described using signals that map time to values within a specific state space. Signals can represent various system properties, such as real values for quantitative measurements like population sizes or performance metrics.

A trace is a collection of such signals that evolves, and each state in the trajectory contains one or more signals that describe the system at that point. To extract meaningful information from these signals during simulation, we use measures, which provide a way to retrieve the value of a specific signal at any given time during the system's execution.

Before proceeding with the simulation process, it is important to formally define a measure, as this plays a central role in interpreting the simulation's results.

A measure serves as a lens through which we can focus on specific aspects of the system at any given time. It allows us to map the system's multidimensional state to a single, interpretable value. This abstraction is handy for quantifying specific system properties throughout its execution.

Measures can be thought of as functions that operate on signals within a trace. They can extract values directly from individual signals or compute more complex metrics based on multiple signals or the overall system state.

Definition 51 (**Measure**)
A measure $\mathfrak{m}$ *is a function that maps a state to a real number:*

$$\mathfrak{m} : X_t \to \mathbb{R}$$

Where X_t *is the state of the system at time* t*.*
More specifically, given a trace $w = ((t_0, X(t_0)), (t_1, X(t_1)), \ldots)$ *and a time* t*, a measure* $\mathfrak{m}$ *can be applied as follows:*

$$\mathfrak{m}(X(t_i)) = f(s_1^w(t_i), s_2^w(t_i), \ldots, s_k^w(t_i))$$

Where:

- $s_j^w(t_i)$ *is the value of the* j*-th signal in trace* w *at time* t_i
- f *is a function that computes the measure based on the values of one or more signals*

The simulation process in Sibilla involves generating multiple trajectories (replicas) of the system and sampling measures at regular intervals. The data collected are then used to compute statistical summaries for each measure. Algorithm 11 describes this

process in detail.

Algorithm 11 Measure Sampling from Trajectories with Statistical Analysis

Given:

$\mathcal{M}$: A Model

X_{init} : Initial state

$\mathfrak{M}$: $\{\mathfrak{m}_1, \mathfrak{m}_2, \ldots, \mathfrak{m}_k \mid \mathfrak{m}_i \in \texttt{Str}\}$ the Set of measures

$\mathrm{R} \in \mathbb{N}^+$: Number of replicas

$\mathrm{T} \in \mathbb{R}^+$: Simulation deadline

$\Delta t \in \mathbb{R}^+$: Sampling time steps

Return:

$\texttt{Result}$: $\{\, k \mapsto \mathrm{M}_k \mid k \in \texttt{Str}\ ,\ \mathrm{M}_k \in \mathbb{R}^{m \times n}\ ,\ m = \lceil T/\Delta_t \rceil, n = 4\}$
A collection of statistical summaries for each measure

$\texttt{Replicas} \leftarrow [\,]$

for $r = 1$ **to** R **do**
  $X_0 \leftarrow X_{\text{init}}$
  $t \leftarrow 0$
  $\texttt{Traj} : \{\, k \mapsto [(t, v)] \mid k \in \texttt{Str}\ ,\ v \in \mathbb{R}\ , t \in \mathbb{R}^+\} \ \leftarrow\ \{\,\}$
  while $t \leqslant \mathrm{T}$ **do**
    $X_{t'}, t' \leftarrow$ next state of $\mathcal{M}$ at time t'
    for each $\mathfrak{m} \in \mathfrak{M}$ **do**
      $v = X_{t'}(\mathfrak{m})$
      $\texttt{Traj}(\mathfrak{m}).\texttt{add}((t', v))$
    $t \leftarrow t'$
  $\texttt{Replicas.add(}\ \texttt{Traj}\ \texttt{)}$

$\texttt{Result} : \{\, k \mapsto \mathrm{M}_k \mid k \in \texttt{Str}\ ,\ \mathrm{M}_k \in \mathbb{R}^{m \times n}\ ,\ m = \lceil T/\Delta_t \rceil, n = 4\} \ \leftarrow\ \{\,\}$

for each $\mathfrak{m} \in \mathfrak{M}$ **do**
  $j \leftarrow 0$
  for $t = 0$ **to** T ***step*** Δt **do**
    $v_{\mathfrak{m}} \leftarrow [\,]$
    for each $\mathit{Traj} \in \mathit{Replicas}$ **do**
      $i \leftarrow \max\{\ k\ \mid \texttt{Traj}(\mathfrak{m})[k].t \leqslant t\ \wedge k \in \{0, 1, ..., |\texttt{Traj}(\mathfrak{m})| - 1\}\ \}$
      $v_{\mathfrak{m}}.\texttt{add}(\texttt{Traj}(\mathfrak{m})[i].v)$
    $j \leftarrow j + 1$
    $\texttt{Result}(\mathfrak{m})[j] \leftarrow [\ t\ ,\ \texttt{mean}(v_{\mathfrak{m}})\ ,\ \texttt{sd}(v_{\mathfrak{m}})\ ,\ \texttt{ci}(v_{\mathfrak{m}})\ ]$

return `Result`

It is important to note that Algorithm 11 is a high-level framework to conduct simulations and statistical analyses in SMC. It serves as a general structure that can incorporate any simulation method within the simulation step. The actual process used to generate the next state and time in the simulation (i.e. the computation of $X_{t'}$ and t') is abstracted in the algorithm, allowing flexibility in choosing the most appropriate simulation algorithm for the studied model.

In Section 2.5.3, we introduced several well-known simulation algorithms, such as the Stochastic Simulation Algorithm (SSA), the Gibson-Bruck method, and Tau-Leaping. These algorithms provide different trade-offs between accuracy and compu-

tational efficiency. For each model introduced in Sibilla, we implement specific simulation algorithms to simulate models, some of which are based on previously introduced methods. For example, the Tau-Leaping method can inspire the simulation of models in LPM to handle large populations efficiently.

The choice of the specific simulation algorithm is encapsulated in the abstracted function that computes the next state and time, allowing flexibility in how simulations are performed within the overall SMC process. This modular approach ensures that Sibilla can adapt to different modelling formalisms and system requirements, leveraging the most suitable simulation technique for each scenario.

The simulation process in Sibilla consists of two main phases:

1. **Trajectory Generation**: For each replica, the algorithm generates a trajectory by evolving the system state from X_{init} until the simulation deadline T. At each step, it records the values of all measures in $\mathfrak{M}$.
2. **Statistical Analysis**: After generating all replicas, the algorithm computes statistical summaries for each measure at regular time intervals (defined by Δt). These summaries include the measurement values' mean, standard deviation, and confidence interval across all replicas.

The output of the simulation is a collection of statistical summaries for each measure, represented as a map `Result` where each key is a measure name and each value is a matrix $\mathrm{M}_k \in \mathbb{R}^{m \times n}$. In this matrix:

- $m = \lceil T/\Delta_t \rceil$ is the number of time points.
- $n = 4$, representing time, mean, standard deviation, and confidence interval.

This approach allows for a comprehensive analysis of the system's behaviour over time, capturing the average behaviour and the variability across different simulation runs. The statistical summaries can be used to:

- Estimate the expected values of various system properties over time.
- Analyse the stability and variability of the system.
- Identify trends and patterns in the behaviour of the system.
- Provide confidence intervals for the estimated measures, quantifying the uncertainty in the results.

This modular approach makes Sibilla a versatile framework for studying a wide range of CAS under various conditions and analysis requirements, providing rich statistical data for each measure of interest.

The simulation of models in Sibilla encompasses a diverse range of approaches, each tailored to the specific characteristics and requirements of the underlying formalisms. This section elucidates the simulation processes for the four primary modelling languages supported by Sibilla: LPM, SLAM, LIO, and YODA. While these formalisms have been described in earlier sections, here we focus on the practical aspects of their simulation within the Sibilla framework.

The simulation process in Sibilla follows these key steps:

1. **Load Model**: The process begins by loading the stochastic model to be simulated. This model defines the system's behaviour and state space.
2. **Create SimulationManager**: A SimulationManager is created to oversee the entire simulation process. It is responsible for managing multiple simulation runs and coordinating the various components involved.
3. **Create SimulationUnit**: A SimulationUnit is created for each replica. This unit encapsulates the model, the initial state, and the SamplingFunction (which defines the measures to be collected).
4. **Create SimulationTask**: The SimulationManager creates a SimulationTask for each SimulationUnit. This task represents a single run of the simulation.
5. **Simulation Loop**: Each SimulationTask enters a loop that continues until the simulation deadline is reached:
   (a) **Create SimulationCursor**: A SimulationCursor steps through the simulation, advancing the system state over time.
   (b) **Compute Next State**: The Simulation Cursor computes the next state of the system based on the model's dynamics.
   (c) **Sample Measures**: A SamplingHandler (derived from the SamplingFunction) is called to collect data on the specified measures at each step.
6. **Replica Loop**: Steps 3-5 are repeated for each replica, as controlled by the SimulationManager.
7. **Compute Statistics**: Once all replicas are complete, the collected data are processed to compute statistical summaries (e.g., mean, standard deviation, confidence intervals) for each measure.
8. **Results**: The final results are produced, which contain statistical information for all measures throughout the simulation time.

The SimulationEnvironment provides the overall context for this process, handling the creation of the SimulationManager and coordinating the execution of multiple SimulationTasks. It ensures the simulation is carried out efficiently, potentially leveraging parallel processing for multiple replicas.

This process allows Sibilla to generate comprehensive statistical data about the system's behaviour over time, providing insights into its dynamics and performance under various conditions.

5.3.1 Language of Population Models (LPM)

The Language of Population Models (LPM) is grounded in the theory of CTMC and adopts a population-centric approach to simulation. In LPM simulations, the system's state is encapsulated in PopulationState objects that quantify the number of individuals in each species. The evolution of this state is governed by a set of reaction rules, represented as PopulationRule objects.

The simulation algorithm for LPM proceeds as follows:

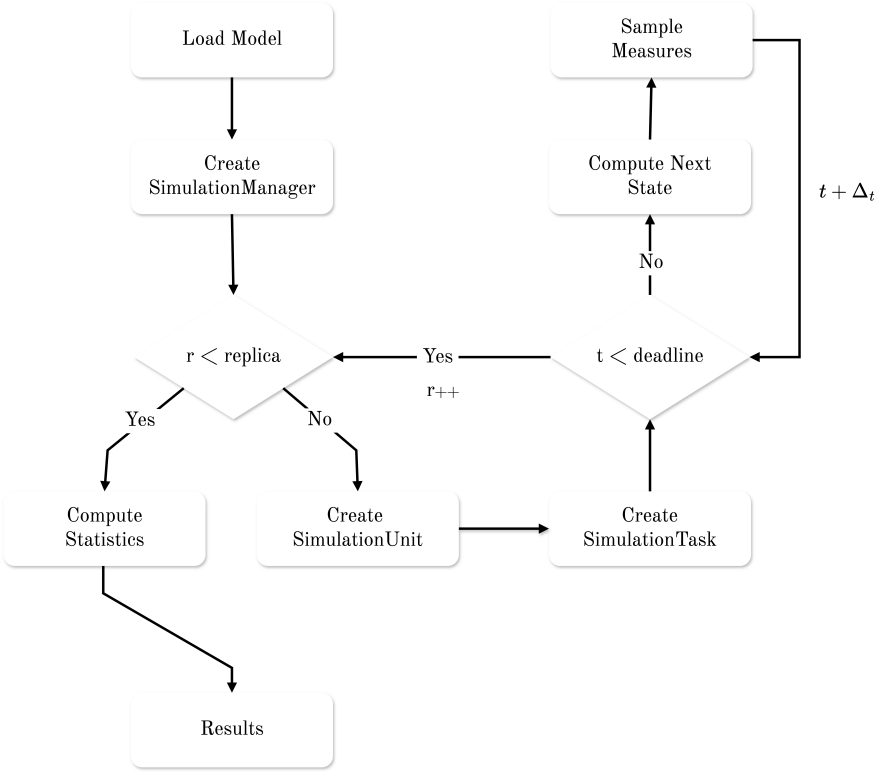


Figure 5.3: *simulation workflow in Sibilla*

1. Initialise the system with a PopulationState object representing the initial population counts.
2. Identify all enabled transitions (reaction rules) based on the current state.
3. Calculate the total rate λ_{tot} by adding the rates of all enabled transitions.
4. Sample the time until the next event from an exponential distribution with rate λ_{tot}.
5. Probabilistically select a transition to execute, with probability proportional to its rate.
6. Update the PopulationState by applying the selected transition.
7. Advance the simulation time and return to Step 2 until a termination condition is met.

This approach effectively captures the stochastic nature of population dynamics, making LPM particularly suitable for modelling systems in fields such as epidemiology, ecology, and biochemistry.

5.3.2 Simple Language for Agent Modelling (SLAM)

SLAM offers a more granular, agent-centric simulation paradigm. SLAM simulations operate on a collection of agents, each defined by a SlamAgentPrototype that specifies

the agent's internal state, observational capabilities, and behavioural logic. The global system state is represented by a SlamSystemState, which encompasses the collective state of all agents and elements of the shared environment.

The SLAM simulation process is inherently time-driven, progressing through discrete time steps:

1. Initialise SlamSystemState with a collection of agents and environment elements.
2. For each discrete time step:
   (a) Update the environment based on the current state of the system.
   (b) For each agent:
      i. Update the agent's observations based on the current environment.
      ii. Determine the agent's action based on its current state and observations.
      iii. Execute the selected action, updating the agent's state and potentially the environment.
   (c) Process any inter-agent messages generated during the step.
   (d) Update global system measures and check for termination conditions.
3. Repeat step 2 until a termination condition is met.

SLAM's flexibility in defining agent behaviours and environmental dynamics makes it particularly adept at modelling complex, multi-agent systems where individual decision-making processes are paramount.

5.3.3 Language of Interactive Objects (LIO)

The Language of Interactive Objects (LIO) bridges the gap between population-level and individual-based modelling approaches. In LIO simulations, the system can be represented at various levels of abstraction, from individual LIOAgent objects to aggregated population counts or fractional representations.

The core LIO simulation algorithm is as follows:

1. Initialise the system state, which may be represented as individual agents, population counts, or fractions.
2. Compute or update the probability matrices governing state transitions based on the current system state.
3. For each agent or population fraction:
   (a) Apply the probability matrix to determine the new state.
   (b) Update the agent or population counts accordingly.
4. Process any inter-agent interactions or message passing.
5. Update global system measures and advance the simulation time.
6. Repeat steps 2-5 until a termination condition is met.

This approach allows LIO to efficiently simulate large-scale systems while retaining the ability to model individual interactions when necessary. The flexibility in state representation enables LIO to adapt to different modelling requirements, making it suitable for a wide range of applications, from social network dynamics to large-scale ecological systems.

5.3.4 Yet anOther agent Description lAnguage (YODA)

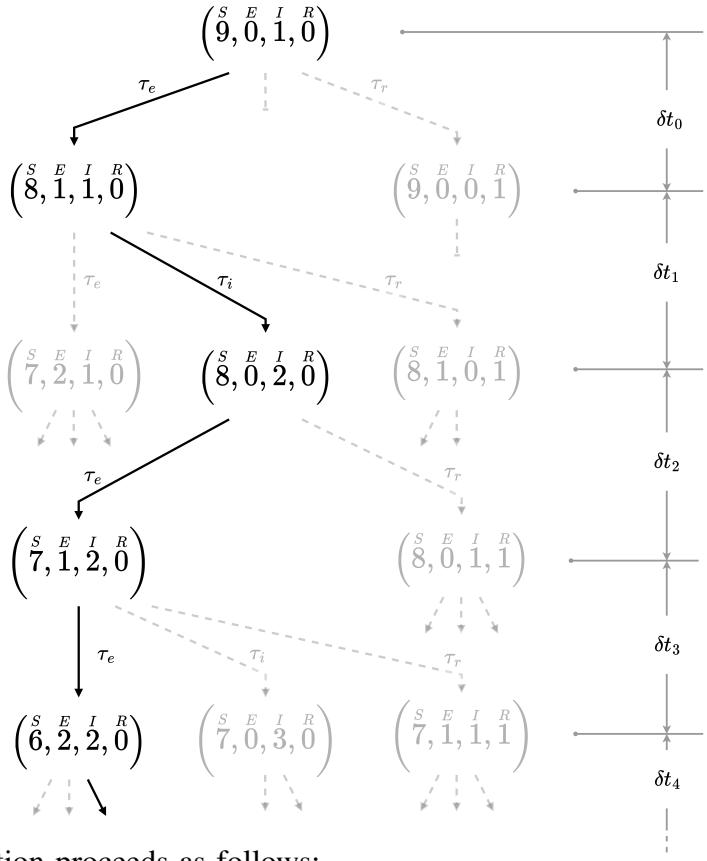
YODA emphasises modelling complex decision-making processes within multi-agent systems. YODA simulations revolve around YodaAgent objects, each equipped with a rich set of attributes, observational capabilities, and behavioural rules. The YodaSystemState encapsulates both the agents and the environmental elements.

The YODA simulation process is distinctly two-phased:

1. Initialise the YodaSystemState with agents and environmental elements.
2. For each discrete time step:
   (a) Action Execution Phase:
      i. For each agent, evaluate its current state and observations.
      ii. Determine the possible actions based on the agent's behaviour rules.
      iii. Probabilistically select and execute an action, updating the agent's state.
   (b) Environmental Update Phase:
      i. Update environmental conditions based on the agents' actions and the progression of time.
      ii. Update agent attributes influenced by the environment.
      iii. Recompute agent observations based on the new environmental state.
   (c) Evaluate system-wide properties and check for termination conditions.
3. Repeat step 2 until a termination condition is met.

YODA's strength lies in its expressive power for defining complex agent behaviours and its rich querying capabilities, enabling detailed system properties analysis during simulation.

Example 5.3.1 (Simulating the SEIR Model)
Building upon the SEIR model defined in Example 5.2.3, we can illustrate how this model is simulated. Here is an example of a trajectory:



The simulation proceeds as follows:

1. Initialise with state $(9, 0, 1, 0)$.
2. At each step, calculate the total rate λ_{tot} as the sum of rates for all enabled transitions:
   - $\lambda_e = X_S \cdot X_I \cdot 10.0 \cdot 1.00$ (for exposure)
   - $\lambda_i = X_E \cdot 2.00$ (for infection)
   - $\lambda_r = X_I \cdot 0.5$ (for recovery)
3. Sample the time until the next event, $\delta t_i \sim \text{Exp}(\lambda_{tot})$.
4. Probabilistically select and apply a transition:
   - τ_e with probability λ_e/λ_{tot}
   - τ_i with probability λ_i/λ_{tot}
   - τ_r with probability λ_r/λ_{tot}
5. Update the state and continue until the simulation deadline is reached.

Example 5.3.2 (Analysing SEIR Model Measures)
Referring to the simulation of the SEIR model illustrated in Example 5.3.1, we can define and analyse specific measures that capture key aspects of the dynamics of the epidemic. Let's focus on two measures:

- $\mathfrak{m}_{\text{sus}}(X_t) = S(t)$: Number of susceptible individuals
- $\mathfrak{m}_{\text{hill}}(X_t) = E(t) + I(t)$: Combined number of exposed and infected individuals (representing the "hill" of the epidemic curve)

Using these measures, we can analyse the state of the system at various time points:

$$\mathfrak{m}_{\text{sus}}(X(0)) = 9 \qquad \mathfrak{m}_{\text{hill}}(X(0)) = 1$$

$$\mathfrak{m}_{\text{sus}}\left(X(\textstyle\sum_{i=0}^{0}(\delta t_i) + \epsilon)\right) = 8 \qquad \mathfrak{m}_{\text{hill}}\left(X(\textstyle\sum_{i=0}^{0}(\delta t_i) + \epsilon)\right) = 2$$

$$\mathfrak{m}_{\text{sus}}\left(X(\textstyle\sum_{i=0}^{1}(\delta t_i) + \epsilon)\right) = 8 \qquad \mathfrak{m}_{\text{hill}}\left(X(\textstyle\sum_{i=0}^{1}(\delta t_i) + \epsilon)\right) = 2$$

$$\mathfrak{m}_{\text{sus}}\left(X(\textstyle\sum_{i=0}^{2}(\delta t_i) + \epsilon)\right) = 7 \qquad \mathfrak{m}_{\text{hill}}\left(X(\textstyle\sum_{i=0}^{2}(\delta t_i) + \epsilon)\right) = 3$$

$$\mathfrak{m}_{\text{sus}}\left(X(\textstyle\sum_{i=0}^{3}(\delta t_i) + \epsilon)\right) = 6 \qquad \mathfrak{m}_{\text{hill}}\left(X(\textstyle\sum_{i=0}^{3}(\delta t_i) + \epsilon)\right) = 4$$

Here ϵ is defined as $0 \leqslant \epsilon < \delta t_{k+1}$ and k is the upper bound of each summation (respectively, $k = 0, 1, 2, 3$).

5.4 Analysis

The analysis component of the Sibilla framework provides a comprehensive set of techniques to extract and interpret the data collected from the simulator. These methods enable a deep investigation of the behaviour of CAS. In this section, we explore three key analysis techniques: *Reachability Analysis*, *First Passage Time Analysis*, and *STL Formula Monitoring*. Each approach offers distinct insights into system dynamics, safety properties, and temporal behaviour, supporting a thorough evaluation of complex stochastic models.

While analytical techniques for evaluating stochastic models—such as those described in Section 2.4—are effective for low-level models with a compact state space, their applicability diminishes in the presence of scale, compositionality, and hybrid dynamics. Although Sibilla includes support for efficient numerical procedures such as the Fox–Glynn method for transient analysis of CTMCs, our focus is on high-level population models featuring large numbers of interacting entities. For such models, the analytical approach becomes impractical, and simulation-based methods are the only viable option for analysis.

By leveraging these simulation-driven techniques, users of Sibilla can verify system properties, assess performance criteria, and make informed decisions in the design and optimisation of collective adaptive systems.

5.4.1 Reachability Analysis

Reachability analysis is a technique used to verify stochastic systems. It aims to determine the probability of reaching a set of states satisfying a given property within a specified time bound. In the context of CAS, reachability analysis provides insights into the system's behaviour and helps verify safety and liveness properties.

As already stated, for such complex stochastic models, analytical solutions are intractable. Therefore, we resort to SMC techniques, which use Monte Carlo simulation-based approaches to estimate this probability. The core idea is to run multiple independent simulations of the model and use the results to estimate the likelihood of reaching the target states.

Definition 52 (Reachability Probability)
Given a stochastic model $\mathcal{M}$, an initial state s_0, a predicate ϕ over the state space, and a time bound T, the probability of reaching a state satisfying ϕ within time T is defined as:

$$P(\text{reach}_{\leqslant T}(\phi)) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{\{\sigma_i \models \phi\}}$$

where σ_i is the i-th simulation trajectory and $\mathbb{1}$ is the indicator function.

Given the stochastic nature of these models, analytical solutions are intractable. Instead, we use statistical model checking techniques, relying on Monte Carlo simulations. The number of simulations N is determined using Chernoff bounds to achieve a desired level of confidence δ and error probability ϵ, so the number of simulations N is determined by:

$$N \geqslant \frac{\ln(2/\delta)}{2\epsilon^2}$$

Algorithm 12 describes the algorithm for reachability analysis

Finally, some remarks on the reachability analysis: first, the target state Ψ is identified using a predicate, ϕ, which evaluates to true if a state satisfies the specified properties. Additionally, a transient condition, ψ, can optionally be defined to restrict the analysis to paths that meet certain conditions throughout the simulation—moreover, employing an adaptive sampling strategy based on Chernoff bounds, enabling early termination if sufficient confidence in the estimated probabilities is achieved.

The output of the reachability analysis is an estimated probability p along with statistical guarantees. Specifically, with probability at least $1 - \delta$, the true reachability probability lies within the interval $[p - \epsilon, p + \epsilon]$.

This information can be used to verify the system's properties, such as ensuring that the probability of reaching an unsafe state is below a certain threshold, verifying that desired states are reached with sufficiently high probability, and estimating the likelihood of the system meeting certain performance criteria within a given time frame.

Algorithm 12 Reachability Analysis

Given:

$\mathcal{M}$	:	A Model
X_{init}	:	Initial state
$\Psi : X_t \to \mathbb{B}$	:	Target condition to be satisfied
$\psi : X_t \to \mathbb{B}$	:	Transient condition (optional)
$\mathrm{T} \in \mathbb{R}^+$	:	Time bound
$\delta \in (0, 1)$	:	Confidence level
$\epsilon \in (0, 1)$	:	Approximation parameter
$\mathrm{N} \in \mathbb{N}^+$	:	Maximum number of samples

Return:

p : Estimated probability of reaching ϕ within time T

```
n ← 0 ;                                                    // Number of successful runs
m ← 0 ;                                                    // Total number of runs
a ← ln(2/δ)/(2ε²) ;                                        // Chernoff bound parameter
while m < N and (m < 4a or m < 4a · ln(a)/n) do
    X_0 ← X_init
    t ← 0  while t ⩽ T do
        X_t, t ← next state of M at time t
        if ∃(ψ) and X_t ⊭ ψ then
            break ;                                        // Transient condition violated
        if X_t ⊨ Ψ then
            n ← n + 1
            break ;                                        // Target condition reached
    m ← m + 1
p ← n/m ;                                                  // Estimated probability
return p
```

5.4.2 First Passage Time Analysis in Stochastic Models

First Passage Time (FPT) analysis is a technique in the study of stochastic processes, relevant in the context of CAS. It focuses on determining the time it takes for a system first to reach a state satisfying a specific condition, providing insights into the temporal dynamics of the system.

Definition 53 (First Passage Time)
Given a stochastic model $\mathcal{M}$, an initial state s_0, a predicate Ψ over the state space, and a time bound T, the first passage time is defined as:

$$\text{FPT}(\Phi \mid \mathcal{M}, s_0) = \inf\{t \leqslant T : X_t \models \Phi\}$$

where X_t is the state of the system at time t, and $\inf$ denotes the infimum.

A First Passage Time Distribution can be used to fully characterise the First Passage Time by considering its probabilistic nature

Algorithm 13 First Passage Time Analysis

Given:

$\mathcal{M}$	:	A Model
X_{init}	:	Initial state
$\Psi : X_t \to \mathbb{B}$	:	Predicate over states
$\mathrm{R} \in \mathbb{N}^+$	:	Number of replicas
$\mathrm{T} \in \mathbb{R}^+$	:	Simulation deadline
$\Delta t \in \mathbb{R}^+$	:	Sampling time step

Return:

`FPT` : First Passage Time statistics

```
tests ← 0 ;                                          // Number of test runs
values ← {} ;                                        // Collection of first passage times

for r = 1 to R do
    X ← X_init
    t ← 0
    tests ← tests + 1
    while t ⩽ T do
        X_t, t ← next state of M at time t
        if X_t ⊨ Ψ then
            values.add(t)
            break ;                                  // condition reached

FPT.tests ← tests                    FPT.hits ← |values|
FPT.min ← min(values)                FPT.max ← max(values)
FPT.mean ← mean(values)              FPT.sd ← standardDeviation(values)
FPT.Q1 ← percentile(values, 25)      FPT.Q2 ← percentile(values, 50)
FPT.Q3 ← percentile(values, 75)

return FPT
```

Definition 54 (First Passage Time Distribution)
The cumulative distribution function of the First Passage Time is given by:

$$F_{\text{FPT}}(t) = P(\text{FPT}(\Phi \mid \mathcal{M}, s_0) \leqslant t)$$

This represents the probability that the system reaches a state satisfying Φ within time t.

As for the reachability analysis, analytical solutions for FPT distributions are often intractable. Therefore, we again rely on a statistical approach based on Monte Carlo simulations. The core idea is to run multiple independent simulations of the model and collect data on the times at which states satisfying the target predicate are first reached. The algorithm for FPT analysis is set out in Algorithm 13

In Sibilla, the FPT analysis is implemented as part of the simulation framework. The key components include: a predicate representing the target condition is defined using a predicate Φ that evaluates to true for states satisfying the desired property, the

number o replica, the number of multiple independent simulations (replicas) to gather a statistically significant sample of FPTs and a time bound (deadline) that bound by a maximum time T, ensuring termination even if the target condition is not met.

The output of the FPT analysis provides a rich statistical summary of the time it takes for the system to reach a state satisfying the specified predicate. This information can be used for various purposes:

- *System Performance*: The mean FPT indicates the average time to reach a state satisfying the predicate.
- *Worst-Case Analysis*: The maximum FPT gives an upper bound on the time to reach a satisfying state.
- *Distribution Analysis*: The quartiles (Q1, Q2, Q3) provide information about the shape of the FPT distribution.
- *Reachability*: The ratio of hits to total tests indicates the probability of reaching a satisfying state within the given time bound.

FPT analysis is particularly valuable in the study of CAS, where it can provide insights into the convergence time, response time, resilience, and scalability of the system. However, it's important to note that the results are based on a finite number of simulations and are thus subject to statistical variation. The results' accuracy depends on the underlying model's fidelity and the appropriateness of the chosen predicate.

The algorithm returns comprehensive statistics on the collected FPT data, including minimum, maximum, quartiles, mean, and standard deviation.

5.4.3 STL Formula Monitoring

Signal Temporal Logic (STL) formula monitoring is used to evaluate the satisfaction of temporal properties in stochastic systems. It provides a framework for specifying and verifying complex temporal behaviours, allowing for qualitative (Boolean) and quantitative (real-valued) analysis of system properties over time.

In the context of CAS, STL monitoring offers a powerful tool for verifying system behaviours, ensuring safety properties, and evaluating performance criteria. By leveraging the expressive power of STL, we can specify intricate temporal patterns and analyse them across multiple system trajectories.

Qualitative Monitoring

Qualitative monitoring focuses on determining whether a given STL formula is satisfied by a system trajectory. It provides a Boolean outcome, indicating whether the specified property holds at each time point.

The algorithm for qualitative monitoring is presented in Algorithm 14. This algorithm evaluates a set of STL formulas over multiple system trajectories, computing the satisfaction probability at each time step.

The qualitative monitoring algorithm works as follows:

1. For each STL formula φ_i, a qualitative monitor $\mathcal{Q}_i$ is created.
2. Multiple system trajectories are sampled using the given model $\mathcal{M}$.

Algorithm 14 Qualitative Monitoring

Given:

$\mathcal{M}$: Model
X_{init} : Initial state
Φ : $\{\varphi_1, \varphi_2, \ldots, \varphi_k\}$: Set of STL formulas
Θ : $\{\Theta_1, \Theta_2, \ldots, \Theta_k\}$: Set of formula parameters
$\Delta t \in \mathbb{R}^+$: : Time step
$\mathrm{T} \in \mathbb{R}^+$: : The simulation deadline
$\mathrm{R} \in \mathbb{N}^+$: : Number of replicas

Return:

`Result` : $\{\varphi_i \mapsto \mathrm{P}_i \mid \varphi_i \in \Phi,\ \mathrm{P}_i \in \mathbb{R}^{m_i \times 2}\}$
A collection of qualitative evaluation for each formula

`Result` $\leftarrow \{\}$
for $i \leftarrow 1$ **to** k **do**
  $\mathcal{Q}_i \leftarrow$ `GetQualitativeMonitor`(φ_i, Θ_i)
  $\mathrm{T}_{\text{eff}} \leftarrow \mathrm{T} - hrz(\varphi_i)$
  $m_i \leftarrow \lceil \mathrm{T}_{\text{eff}} / \Delta t \rceil$
  $\mathrm{P}_i \leftarrow [\,]\,[\,]$ matrix of size $m_i \times 2$
  `sat_counts` $\leftarrow \begin{bmatrix} 0 & 1 & & m_i-2 & m_i-1 \\ 0\,, & 0\,, \ldots, & 0 & , & 0 \end{bmatrix}$
  for $r \leftarrow 1$ **to** R **do**
    $w \leftarrow$ `SampleTrajectory`$(\mathcal{M}, X_{\text{init}}, \mathrm{T})$
    boolean_signal $\leftarrow \mathcal{Q}_i$.`monitor`(w)
    for $j \leftarrow 0$ **to** $m_i - 1$ **do**
      $t_j \leftarrow j \cdot \Delta t$
      if *boolean_signal*$(\ t_j\ )$ **then**
        `sat_counts`$[j] \leftarrow$ `sat_counts`$[j] + 1$
  for $j \leftarrow 0$ **to** $m_i - 1$ **do**
    $t_j \leftarrow j \cdot \Delta t$
    $p_j \leftarrow$ `sat_counts`$[j]/\mathrm{R}$
    $\mathrm{P}_i[j] \leftarrow [t_j, p_j]$
  `Result`$[\varphi_i] \leftarrow \mathrm{P}_i$
return `Result`

3. Each trajectory is evaluated against the STL formula at discrete time points.
4. The algorithm counts the number of satisfying trajectories at each time point.
5. Finally, it computes the probability of satisfaction at each time point by dividing the satisfaction count by the total number of trajectories.

The output `Result` maps each formula φ_i to a matrix P_i, where each row contains a time point and the corresponding satisfaction probability.

For cases where we are only interested in the probability of satisfaction at the initial time point (t = 0), we can use a simplified algorithm as shown in Algorithm 15.

Algorithm 15 Expected Probability At Time 0

Given:

$\mathcal{M}$: Model
X_{init} : Initial state
φ : STL formula
Θ : Formula parameters
$\mathrm{R} \in \mathbb{N}^+$: Number of replicas

Return:

p : Expected probability of satisfaction at time 0

$\mathcal{Q} \leftarrow$ `GetQualitativeMonitor`(φ, Θ)
$\mathrm{T} \leftarrow hrz(\varphi)$
`sat_count` $\leftarrow 0$
for $r \leftarrow 1$ **to** R **do**
  $w \leftarrow$ `SampleTrajectory`$(\mathcal{M}, X_{\text{init}}, \mathrm{T})$
  boolean_signal $\leftarrow \mathcal{Q}$.`monitor`(w)
  if *boolean_signal*(`time : 0`) **then**
    `sat_count` $\leftarrow$ `sat_count` $+ 1$
$p \leftarrow$ `sat_counan`$/\mathrm{R}$

return p

This algorithm focusses on satisfying the STL formula at the initial time point, providing a single probability value as a output.

Quantitative Monitoring

While qualitative monitoring provides boolean outcomes, quantitative monitoring offers a more nuanced evaluation of STL formula satisfaction. It computes a real-valued robustness degree, indicating whether a property is satisfied and how well it is satisfied.

Algorithm 16 presents the procedure for the quantitative monitoring of STL formulas.

The quantitative monitoring algorithm operates as follows:

1. For each STL formula φ_i, a quantitative monitor ρ_i is created.
2. Multiple system trajectories are sampled using the given model $\mathcal{M}$.
3. Each trajectory is evaluated against the STL formula at discrete time points, computing a robustness value.
4. The algorithm calculates the mean and standard deviation of robustness values at each time point across all trajectories.
5. The output `Result` maps each formula φ_i to a matrix M_i, where each row contains a time point, mean robustness, and standard deviation of robustness.

For cases where we are only interested in the robustness at the initial time point (t = 0), we can use a simplified algorithm as shown in Algorithm 17. This algorithm

Algorithm 16 Quantitative Monitoring

Given:

$\mathcal{M}$: Model
X_{init} : Initial state
Φ : $\{\varphi_1, \varphi_2, \ldots, \varphi_k\}$: Set of STL formulas
Θ : $\{\Theta_1, \Theta_2, \ldots, \Theta_k\}$: Set of formula parameters
$\Delta t \in \mathbb{R}^+$: : Time step
$\mathrm{T} \in \mathbb{R}^+$: : The simulation deadline
$\mathrm{R} \in \mathbb{N}^+$: : Number of replicas

Return:

`Result` : $\left\{\varphi_i \mapsto \mathrm{M}_i \mid \varphi_i \in \Phi,\ \mathrm{M}_i \in \mathbb{R}^{m_i \times 3}\right\}$
A collection of quantitative evaluation for each formula

```
Result ← {}
for i ← 1 to k do
    ρ_i ← GetQuantitativeMonitor(φ_i, Θ_i)
    T_eff ← T − hrz(φ_i)
    m_i ← ⌈T_eff/Δt⌉
    Trajectories ← [ ]
    for r ← 1 to R do
        w_r ← SampleTrajectory(M, X_init, T)
        Trajectories.add(w_r)
    M_i ← [ ] [ ]  matrix of size m_i × 3
    for j ← 0 to m_i − 1 do
        t_j ← j · Δt
        robustness_values ← [ ]
        for w ∈ Trajectories do
            robustness_values.add(ρ_i(w, t_j))
        μ_j ← Mean( robustness_values )
        σ_j ← StandardDeviation( robustness_values )
        M_i[j] ← [t_j, μ_j, σ_j]
    Result[φ_i] ← M_i
return Result
```

focuses on computing robustness's mean and standard deviation at the initial time point, providing a concise summary of the quantitative satisfaction.

STL formula monitoring, through both qualitative and quantitative approaches, provides valuable insights into the behaviour of CAS. It allows for rigorous verification of temporal properties, assessment of system performance, and evaluation of design choices.

5.4.4 Validation Against Analytical Solutions

In the previous sections, we have discussed various types of analysis capabilities of Sibilla, including reachability analysis of specific states, first passage time analysis, and both quantitative and qualitative temporal logic verification. While these techniques

Algorithm 17 Mean and Sd Robustness At Time 0

Given:

$\mathcal{M}$: Model
X_{init} : Initial state
φ : STL formula
Θ : Formula parameters
$\mathrm{R} \in \mathbb{N}^+$: Number of replicas

Return:

$[\mu, \sigma]$: Mean and standard deviation of robustness at time 0

$\rho \leftarrow$ `GetQuantitativeMonitor`(φ, Θ)
$\mathrm{T} \leftarrow hrz(\varphi)$
`Trajectories` $\leftarrow [\,]$

for $r \leftarrow 1$ **to** R **do**
  $w_r \leftarrow$ `SampleTrajectory`$(\mathcal{M}, X_{\text{init}}, \mathrm{T})$
  `Trajectories.add`(w_r)

`robustness_values` $\leftarrow [\,]$
for $w \in$ *`Trajectories`* **do**
  `robustness_values.add`$(\rho(w, 0))$
$\mu \leftarrow$ `Mean( robustness_values )`
$\sigma \leftarrow$ `StandardDeviation( robustness_values )`
return $[\mu, \sigma]$

demonstrate the versatility of our statistical simulation approach, it is crucial to validate the accuracy of our methods against exact analytical solutions when available.

To address this validation requirement, we compare our statistical simulation results with those obtained from PRISM [157], a well-established probabilistic model checker that provides analytical solutions for continuous-time Markov chains. PRISM, introduced in our state-of-the-art review, employs symbolic model checking techniques to compute exact probability distributions and temporal properties analytically, making it an ideal reference for validating the accuracy of our simulation-based approach.

Case Study: Molecular Reaction

For this validation study, we analyze a chemical reaction network case study from the PRISM model checker repository [189]. The system models the reversible ionization reaction between sodium and chlorine molecules, representing a fundamental chemical equilibrium process.

The molecular system under consideration involves the reversible ionization reaction:

$$\mathrm{Na} + \mathrm{Cl} \rightleftharpoons \mathrm{Na}^+ + \mathrm{Cl}^- \tag{5.1}$$

This reaction system is particularly well-suited for validation purposes because it represents a scalable finite-state continuous-time Markov chain that can be tested with both small and large state spaces while remaining easy to understand.

In this approach, the time until a reaction between two molecules is modelled as an exponential distribution, leading to a CTMC model where states correspond to the number of molecules of each type and transitions represent possible reactions between molecules. The rate of each reaction is determined by a base rate multiplied by the concentrations of the reactants.

For each chemical reaction, the rate is calculated as the base rate multiplied by the product of the number of molecules of each type that participates in the reaction. This formulation naturally leads to mass-action kinetics, where reaction rates are proportional to the product of reactant concentrations.

The forward reaction rate is given by:

$$\lambda_{\text{forward}} = k_1 \cdot [\text{Na}] \cdot [\text{Cl}] \tag{5.2}$$

The backward reaction rate is:

$$\lambda_{\text{backward}} = k_2 \cdot [\text{Na}^+] \cdot [\text{Cl}^-] \tag{5.3}$$

where $k_1 = 100$ and $k_2 = 10$ are the base reaction rates for the forward and backward reactions, respectively.

The primary property under investigation is the probability that exactly i sodium molecules are present at time T, providing a direct measure of the system's transient behaviour that can be compared between analytical and simulation-based approaches.

$$P(\text{Na}(T) = i) = \mathbb{P}[\text{Na}(T) = i] \tag{5.4}$$

This provides a direct measure of the system's transient behaviour that can be compared between analytical and simulation-based approaches.

PRISM Model Implementation

The PRISM implementation of the sodium-chlorine ionization system is structured as a modular CTMC model:

```
ctmc

const int N1;
const int N2;

module na
  na : [0..N1] init N1;
  [e1] na>0  ->     na  : (na'=na-1);
  [e2] na<N1 -> (N1-na) : (na'=na+1);
endmodule

module cl
  cl : [0..N2] init N2;
  [e1] cl>0  ->     cl  : (cl'=cl-1);
  [e2] cl<N2 -> (N2-cl) : (cl'=cl+1);
endmodule

const double e1rate = 100; // Na + Cl -> Na+ + Cl-
const double e2rate = 10; // Na+ + Cl- -> Na + Cl
```

```
module base_rates
  [e1] true -> e1rate : true;
  [e2] true -> e2rate : true;
endmodule

rewards "percentage_na"
  true : 100*na/N1;
endrewards
```

Listing 5.1: *Sodium-Chlorine Ionization Model in PRISM*

The PRISM model employs a modular structure where each molecular species is represented by a separate module. The synchronization labels `e1` and `e2` coordinate the reactions between modules, ensuring that forward and backward reactions occur simultaneously across all participating species. The base rates module defines the kinetic parameters, while the rewards structure enables the computation of derived quantities such as molecular percentages.

In PRISM, the property of interest is formulated as:

```
// Constants:
const double T; // time instant
const int i; // number of molecules

// Probability of there being i Na molecules at time T?
P=? [ true U[T,T] na=i ]
```

Sibilla Population Model Implementation

The corresponding implementation in Sibilla's population modelling language (LPM) is shown below:

```
/* Na + Cl <--> Na+ + Cl- Population Model */

/* Parameters - reaction rates */
param e1rate = 100.0;    /* Forward reaction rate: Na + Cl -> Na+ + Cl-
 */
param e2rate = 10.0;     /* Backward reaction rate: Na+ + Cl- -> Na +
Cl */
param i = 1;

/* Constants - initial molecule counts */
param N1 = 10;           /* Total number of Na/Na+ molecules */
param N2 = 10;           /* Total number of Cl/Cl- molecules */

/* Species definitions */
species Na;              /* Sodium atoms */
species Na_plus;         /* Sodium ions */
species Cl;              /* Chlorine atoms */
species Cl_minus;        /* Chloride ions */

/* Forward reaction: Na + Cl -> Na+ + Cl- */
rule forward_reaction {
  Na|Cl -[ #Na * #Cl * e1rate ]-> Na_plus|Cl_minus
}
```

```
/* Backward reaction: Na+ + Cl- -> Na + Cl */
  rule backward_reaction {
Na_plus|Cl_minus -[ #Na_plus * #Cl_minus * e2rate ]-> Na|Cl
}

/* Initial system configuration */
system initial = Na<N1>|Cl<N2>|Na_plus<0>|Cl_minus<0>;
predicate na_equals_i = #Na == i;
```

Listing 5.2: *Sodium-Chlorine Ionization Model in Sibilla*

This implementation captures the essential dynamics of the chemical system while maintaining consistency with the PRISM model formulation.

Experimental Setup

The validation study employs a comprehensive comparison between PRISM's analytical solutions and Sibilla's statistical simulations across multiple methodological approaches.

PRISM Analytical Analysis Using PRISM's CTMC capabilities, we computed exact transient probability distributions $P(\text{Na} = i, t)$ for $i \in \{0, 1, \ldots, 10\}$ at 30 equidistant time points over the interval $[0, 0.006]$. The analytical solution provides ground truth probabilities for each possible state at each time point.

Sibilla Statistical Simulation Two different simulation approaches were employed to assess various aspects of statistical accuracy:

- *Approach 1: Individual Trajectory Analysis*: individual trajectories were collected using single-replica simulations (replica = 1), repeated 1000 times. For each time point, we constructed empirical probability distributions by counting the frequency of each possible Na count across all simulation runs. This approach provides the most direct comparison with PRISM's analytical probability distributions, as it generates empirical probabilities for each discrete state without distributional assumptions. However, this methodology is not standard practice in Sibilla, where running multiple single-replica simulations is computationally inefficient and operationally inconvenient. This approach was implemented specifically for validation purposes to provide the most accurate comparison with analytical results.
- *Approach 2: Multi-Replica Statistical Summaries*: multi-replica simulations (replica = 1000) were used to collect mean and standard deviation at each time point, providing summary statistics for the population dynamics. This approach represents Sibilla's standard operational mode, where statistical summaries (mean, standard deviation, and optionally descriptive statistics including quartiles) are computed across multiple replicas at each time point. To enable comparison with PRISM's probability distributions, we converted these summary statistics into approximate probability distributions using the mean $\mu(t)$ and standard deviation $\sigma(t)$, assuming a normal distribution with continuity correction:

$$P_{\text{normal}}(\text{Na} = i, t) = \Phi\left(\frac{i + 0.5 - \mu(t)}{\sigma(t)}\right) - \Phi\left(\frac{i - 0.5 - \mu(t)}{\sigma(t)}\right) \tag{5.5}$$

where Φ is the standard normal cumulative distribution function. The continuity correction (± 0.5) accounts for the discrete nature of molecular counts when approximating with a continuous distribution. This approach validates Sibilla's standard output format and demonstrates how summary statistics can be transformed into probability estimates for comparison with analytical solutions.

Figure 5.4 shows the evolution of the mean number of sodium molecules and its standard deviation over time from Sibilla's multi-replica simulations. The mean decreases from the initial value of 10 molecules toward the equilibrium value around 2.5 molecules, while the standard deviation initially increases as the system explores intermediate states before stabilizing at the equilibrium variance.

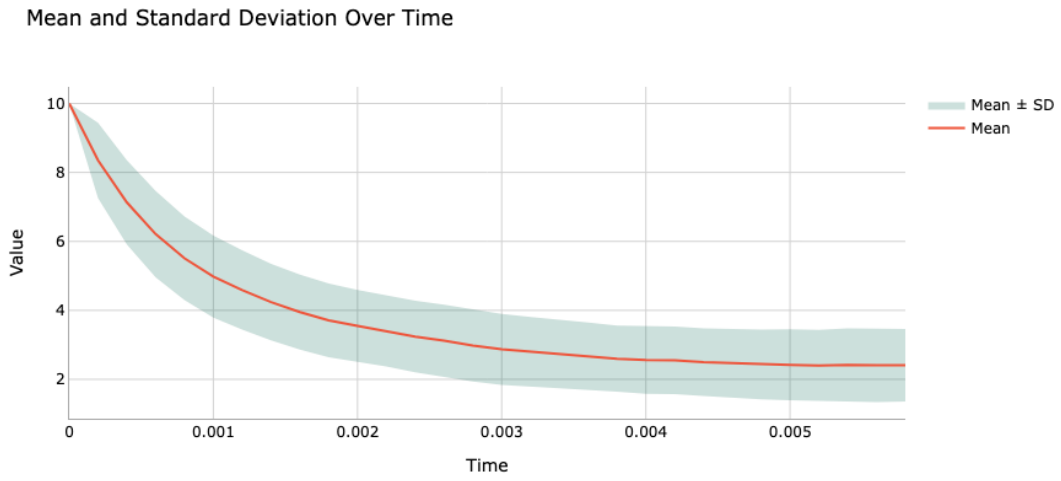
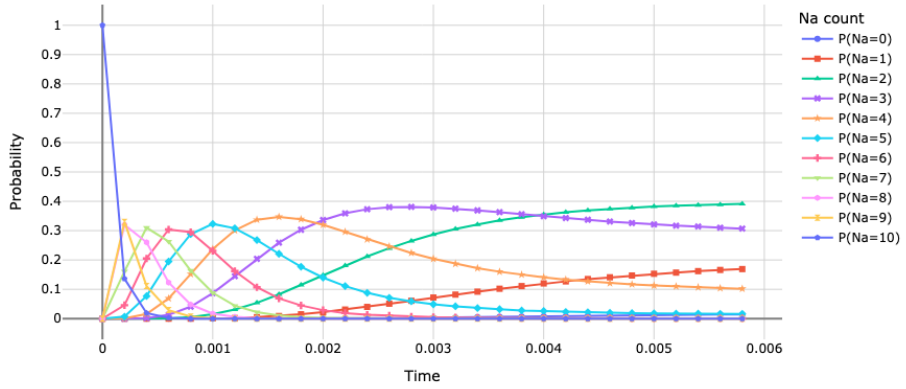


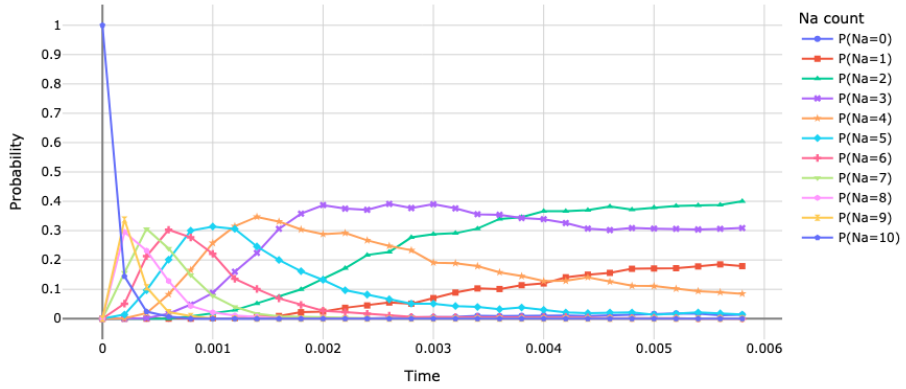
Figure 5.4: *Evolution of mean number of sodium molecules and standard deviation over time from Sibilla's multi-replica simulations (Approach 2). The red line shows the mean decreasing from the initial value of 10 molecules toward the equilibrium value around 2.5 molecules, following the expected trajectory of the chemical reaction system. The shaded area represents one standard deviation around the mean, illustrating the variability in molecular counts across simulation replicas. The standard deviation initially increases as the system explores intermediate states during the transition phase, then stabilizes at the equilibrium variance.*

Results and Validation

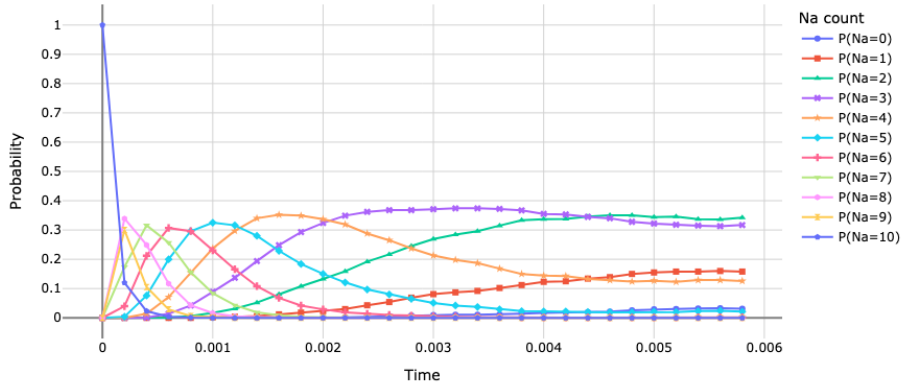
Figure 5.5 show the transient probability distributions $P(\text{Na} = i, t)$ obtained from PRISM (analytical), Sibilla Approach 1 (empirical trajectories), and Sibilla Approach 2 (normal approximation), respectively. All three approaches capture the fundamental system dynamics: the initial deterministic state ($P(\text{Na} = 10, t = 0) = 1.0$), the rapid transition phase where probability mass spreads across intermediate states, and the gradual evolution toward equilibrium distribution.



(a)



(b)



(c)

Figure 5.5: *Transient probability distributions $P(Na = i, t)$ for the sodium-chlorine ionization reaction obtained using different analytical and simulation approaches. (a) PRISM analytical results providing exact probability distributions computed via continuous-time Markov chain analysis. (b) Sibilla Approach 1 results using empirical probability distributions constructed from 1000 individual trajectory simulations. (c) Sibilla Approach 2 results using normal distribution approximations derived from multi-replica simulation statistics (mean and standard deviation).*

The visual comparison reveals excellent qualitative agreement between all approaches. The characteristic probability curves show consistent temporal evolution patterns, with the system transitioning from the initial pure state through intermediate distributions to approaching equilibrium by $t = 0.006$.

Quantitative Error Analysis To assess the accuracy of Sibilla's simulation approaches, we computed three distance metrics comparing the probability distributions against PRISM's analytical solutions:

- **L2 (Euclidean) Distance**: Measures the geometric distance between two probability distributions in Euclidean space, treating each distribution as a vector and computing the straight-line distance between them.

$$d_{L2}(P, Q) = \sqrt{\sum_{i=0}^{10} (P_i - Q_i)^2}$$

- **Total Variation Distance**: Quantifies the maximum difference between two probability distributions, representing the largest possible difference in probability that the distributions can assign to the same event.

$$d_{TV}(P, Q) = \frac{1}{2} \sum_{i=0}^{10} |P_i - Q_i|$$

- **Kullback-Leibler Divergence**: Measures the information-theoretic distance between distributions, quantifying how much information is lost when using distribution Q to approximate the true distribution P.

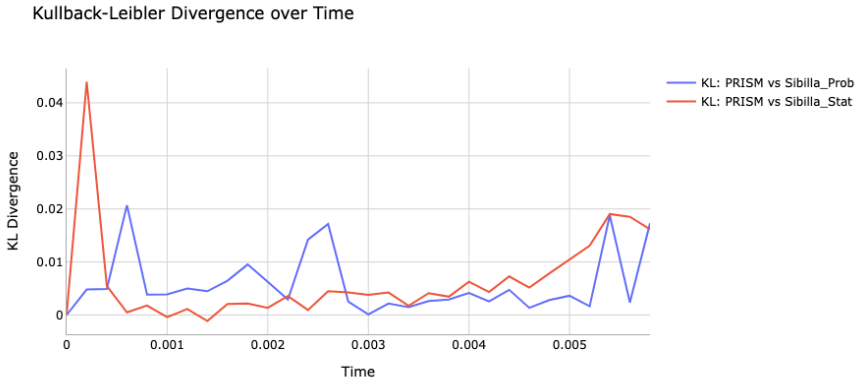
$$d_{KL}(P, Q) = \sum_{i=0}^{10} P_i \log \left(\frac{P_i}{Q_i} \right)$$

The error metrics over time are shown in Figure 5.6. All distance measures remain consistently low throughout the simulation period, with occasional spikes during rapid transition phases where small timing differences become more apparent.

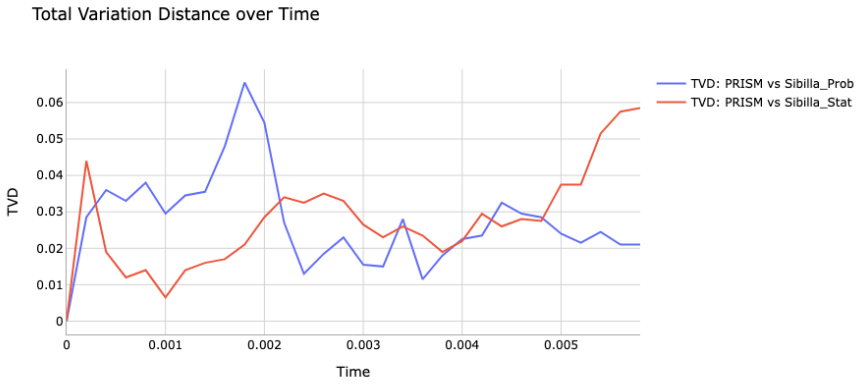
Summary Statistics The average error metrics across all time points demonstrate the high accuracy of both Sibilla approaches:

Comparison	L2 Distance	TV Distance	KL Divergence
PRISM vs Sibilla Approach 1	0.0286	0.0252	0.0078
PRISM vs Sibilla Approach 2	0.0285	0.0265	0.0048

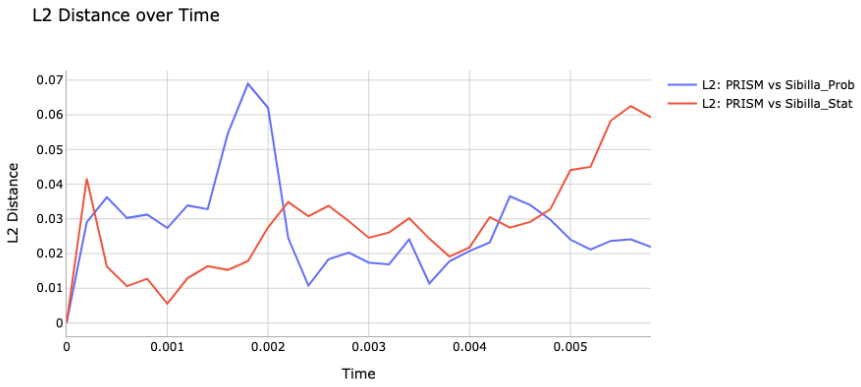
Both simulation approaches achieve remarkably similar accuracy levels, with average L2 distances below 0.03 and total variation distances below 0.027. The KL divergences are particularly low (< 0.008), indicating excellent preservation of the probability distribution structure.



(a)



(b)



(c)

Figure 5.6: *Quantitative error analysis comparing Sibilla's simulation approaches against PRISM's analytical solutions using different distance metrics. (a) Kullback-Leibler divergence showing consistently low values (< 0.045) throughout the simulation period. (b) Total variation distance quantifying the maximum difference between probability distributions, with values remaining below 0.065 and averaging around 0.025. (c) L2 (Euclidean) distance consistently staying below 0.07. All metrics demonstrate excellent agreement between simulation and analytical approaches, with occasional spikes during rapid transition phases where small timing differences become more apparent. The consistently low error values across all time points validate the accuracy of Sibilla's statistical model checking capabilities.*

Comparative Assessment of Approaches **Approach 1 (Empirical Trajectories)** provides slightly better performance in terms of L2 and TV distances, which is expected since it directly estimates empirical probabilities without distributional assumptions. The method captures the discrete nature of the probability distributions exactly.

Approach 2 (Normal Approximation) demonstrates surprisingly robust performance despite the simplifying assumption of normality. The slightly higher TV distance (0.0265 vs 0.0252) reflects minor discrepancies in the tail probabilities, but the overall approximation quality remains excellent. Notably, this approach actually achieves lower KL divergence, suggesting good preservation of the distribution's information content.

Validation Conclusions The validation study confirms that Sibilla's statistical model checking capabilities provide highly accurate approximations to analytical solutions. Both simulation approaches maintain errors well within acceptable bounds for practical applications:

- Average probability errors remain below 3% (L2 distance < 0.03)
- Maximum total variation distances stay below 0.065, indicating good distributional fidelity
- KL divergences remain minimal, confirming preservation of information structure

The strong agreement between Sibilla's standard output format (Approach 2) and analytical solutions validates the framework's capability to provide reliable statistical estimates for complex stochastic systems where analytical methods become intractable. This establishes confidence in Sibilla's application to larger, more complex systems beyond the reach of exact analytical verification.

Scalability and Computational Complexity The validation study also demonstrates the complementary strengths of analytical and simulation-based approaches regarding scalability. Table 5.1 shows the computational characteristics of PRISM's analytical approach for different system sizes (N1, N2 values).

The table compares the scalability of the analytical engine of PRISM by measuring key computational characteristics as the model size increases, with symmetric initial molecule counts ($N_1 = N_2$). The columns represent the following quantities: **N1, N2** indicates the initial number of sodium and chlorine molecules; **States** is the number of distinct global states generated in the model; **Transitions** reports the total number of state transitions in the underlying Markov chain; **MTBDD Nodes** is the number of nodes in the Multi-Terminal Binary Decision Diagram used to compactly represent the transition system; **Construction Time (s)** denotes the time, in seconds, required to build the model; and **Iterations** refers to the number of iterations performed to compute the transient probability distribution at a given time instant.

For small population sizes (N1, N2 $\leqslant$ 100) , both PRISM and Sibilla provide comparable computational performance, with PRISM's analytical approach offering exact solutions in reasonable time. However, as system size increases, the state space grows quadratically ($(N1 + 1) \times (N2 + 1)$ states), leading to exponential increases in computational complexity for analytical methods.

Table 5.1: *Scalability comparison between PRISM and Sibilla approaches*

N1, N2	States	Transitions	MTBDD Nodes	Construction Time (s)	Iterations
10, 10	11	20	247	0.061	11
20, 20	21	40	577	0.065	21
40, 40	41	80	1,317	0.104	41
60, 60	61	120	1,967	0.159	61
80, 80	81	160	2,957	0.280	81
100, 100	101	200	3,695	0.351	101
250, 250	251	500	10,231	2.742	251

In contrast, Sibilla's simulation-based approach scales linearly with system size, as the computational cost depends primarily on the number of simulation replicas rather than the state space size. This fundamental difference makes Sibilla particularly valuable for large-scale systems where analytical verification becomes computationally intractable, while maintaining the accuracy demonstrated in this validation study.

Detailed probability distributions for all time points and approaches, including the complete numerical data used in this validation study, can be found in Appendix C.

5.5 Synthesis

Synthesis in the context of stochastic modelling is determining parameter configurations that meet specific objectives regarding the satisfaction or violation of given properties. This section elaborates on how the synthesis module integrates with other components of Sibilla and how Sibilla's structure facilitates practical synthesis tasks.

The Synthesis module heavily relies on the Signal Temporal Logic monitoring capabilities provided by Sibilla's analysis components. STL monitoring allows for qualitative and quantitative evaluation of temporal properties over system trajectories. In the context of synthesis, these capabilities are essential for defining objective functions, specifying constraints on system behaviour, and evaluating the performance of synthesised designs.

The Synthesis module builds upon Sibilla's simulation capabilities and measure sampling techniques. The simulation engine efficiently samples the parameter space, generating trajectories for different parameter configurations. Measure sampling techniques provide a way to collect relevant statistics across multiple simulation runs, which is crucial for assessing the performance of synthesised designs in stochastic settings.

The architecture of Sibilla, particularly its approach to model definition and analysis, significantly facilitates the implementation of synthesis tasks. All models in Sibilla are designed with inherent support for parameterisation. This design choice allows for the development of generic synthesis algorithms that can be applied to a wide range of models, regardless of the specific modelling formalism used (LPM, LIO, YODA, SLAM). The explicit representation of parameters simplifies exploring different system configurations, which is at the core of synthesis tasks.

Synthesis in the context of stochastic modelling is determining parameter configurations that meet specific objectives regarding the satisfaction or violation of given properties. This section elaborates on the theoretical foundations of the synthesis approaches implemented in this work, highlighting the versatility of the synthesis capa-

bilities provided by our framework. Specifically, we discuss the Optimal Feasibility and Variant Model Adjustment approaches, the parametric nature of models within the Sibyl instrument, and the implementation of synthesis tasks and strategies.

5.5.1 Theoretical Overview of Synthesis Approaches

We present two main approaches for synthesis in stochastic models: Optimal Feasibility and Variant Model Adjustment. Both approaches leverage a surrogate-based methodology to navigate the parameter space efficiently.

Optimal Feasibility

Optimal feasibility aims to find an instantiation of parameters that optimises the probability or robustness of satisfying a given property. We begin by defining the optimal feasibility problem in terms of probability:

Definition 55 (Optimal Feasibility - Probability)
Given a stochastic model $\mathcal{M}$, a property φ, and a set of constraints C, the optimal feasibility problem for probability is defined as:

$$\begin{aligned} \Theta^* = \arg\min_{\Theta \in \mathcal{H}} \quad & P\left(\varphi(\Theta) \mid \mathcal{M}_\Theta\right) \\ \textit{s.t.} \quad & c_i(\Theta) \quad \forall c_i \in C \end{aligned}$$

where Θ^* is the optimal parameter configuration, $\mathcal{H}$ is the parameter space, $P(\varphi(\Theta) \mid \mathcal{M}_\Theta)$ is the probability of satisfying property φ given model $\mathcal{M}$ with parameters Θ, and C is a set of constraint functions.

While the previous definition focused on probability, we can also formulate the optimal feasibility problem in terms of robustness:

Definition 56 (Optimal Feasibility - Robustness)
Similarly, for robustness, the optimal feasibility problem is defined as:

$$\begin{aligned} \Theta^* = \arg\min_{\Theta \in \mathcal{H}} \quad & R\left(\varphi(\Theta) \mid \mathcal{M}_\Theta\right) \\ \textit{s.t.} \quad & c_i(\Theta) \quad \forall c_i \in C \end{aligned}$$

where $R(\varphi(\Theta) \mid \mathcal{M}_\Theta)$ is the robustness of satisfying property φ given model $\mathcal{M}$ with parameters Θ.

Optimal feasibility can be employed for model repair and falsification. In model repair, we seek parameters that maximise the satisfaction probability or robustness of the property, effectively adjusting the model to achieve the desired behaviour. Conversely, we search for configurations that minimise the satisfaction probability or robustness in model falsification, identifying worst-case scenarios.

Variant Model Adjustment

The Variant Model Adjustment approach allows for systematic comparison and adjustment of model variants. We consider a baseline model $\mathcal{M}$ and a variant $\widetilde{\mathcal{M}}$, both verified against the same set of logical-time formulas $\Phi = [\varphi_1, \varphi_2, \ldots, \varphi_n]$. To facilitate this comparison, we first define the concept of an expected robustness vector:

Definition 57 (Expected Robustness Vector)
Given a model $\mathcal{M}$ and a set of formulas Φ, we define the expected robustness vector as:

$$\vec{R}(\Phi \mid \mathcal{M}) = (R(\varphi_i \mid \mathcal{M}) \;\mid\; \forall \varphi_i \in \Phi)$$

where $R(\varphi_i \mid \mathcal{M})$ is the expected robustness of formula φ_i given model $\mathcal{M}$.

To express parametrised formulas, we use $\Phi_\Theta = [\varphi_1(\Theta), \varphi_2(\Theta), \ldots, \varphi_n(\Theta)]$.

With the expected robustness vector defined, we can now formulate three objectives for variant model adjustment. The first objective is preservation, which aims to maintain similar behaviour between the original and variant models:

Definition 58 (Preservation)
Tune Θ over $\widetilde{\mathcal{M}}_\Theta$ to be as similar as possible to $\mathcal{M}$ in satisfying Φ:

$$\Delta^{\simeq}_{\Phi}\left(\mathcal{M}, \widetilde{\mathcal{M}}_\Theta\right) = \min_{\Theta \in \mathcal{H}} \left(d\left(\vec{R}(\Phi \mid \mathcal{M}), \vec{R}\left(\Phi_\Theta \mid \widetilde{\mathcal{M}}_\Theta\right)\right)\right)$$

The second objective focuses on improving the variant model's performance relative to the original:

Definition 59 (Improvement)
Maximise the distance so that $\widetilde{\mathcal{M}}_\Theta$ satisfies Φ better than $\mathcal{M}$:

$$\Delta^{+}_{\Phi}\left(\mathcal{M}, \widetilde{\mathcal{M}}_\Theta\right) = \max_{\Theta \in \mathcal{H}} \left(d^\star\left(\vec{R}\left(\Phi_\Theta \mid \widetilde{\mathcal{M}}_\Theta\right), \vec{R}(\Phi \mid \mathcal{M})\right)\right)$$

Finally, we consider the objective of intentional divergence, which aims to worsen the variant model's performance:

Definition 60 (Intentional Divergence)
Maximise the distance so that $\widetilde{\mathcal{M}}_\Theta$ satisfies Φ worse than $\mathcal{M}$:

$$\Delta^{-}_{\Phi}\left(\mathcal{M}, \widetilde{\mathcal{M}}_\Theta\right) = \max_{\Theta \in \mathcal{H}} \left(d^\star\left(\vec{R}(\Phi \mid \mathcal{M}), \vec{R}\left(\Phi_\Theta \mid \widetilde{\mathcal{M}}_\Theta\right)\right)\right)$$

Where d is the *Minkowski distance* , a generalised form of the geometrical distance measure distance between two vectors, and it's defined as

$$d\left(\vec{x}, \vec{y}\right) = \sqrt[p]{\sum_{i=1}^{n} \left(|x_i - y_i|\right)^p}$$

Minkowski distance can be used with p being 1 or 2, which correspond to the *Manhattan distance* and the *Euclidean distance*, respectively. In the limiting case of $p \to +\infty$, we obtain the *Chebyshev distance*. The Minkowski distance formula, as it is defined in the case of maximisation, does not allow us to decide whether we are maximising worse or better, so the formula must be adapted as follows:

$$d^{\star}\left(\vec{x}, \vec{y}\right) = \sqrt[p]{\sum_{i=1}^{n} \left|\max\left(0, x_i - y_i\right)\right|^p}$$

With this simple reformulation it is possible to maximise the improvement by writing $d^{\star}\left(\vec{R}\left(\Phi_{\Theta} \mid \widetilde{\mathcal{M}}_{\Theta}\right), \vec{R}\left(\Phi \mid \mathcal{M}\right)\right)$ and $d^{\star}\left(\vec{R}\left(\Phi \mid \mathcal{M}\right), \vec{R}\left(\Phi_{\Theta} \mid \widetilde{\mathcal{M}}_{\Theta}\right)\right)$ to maximise the worsening. It should be noted that by varying the parameters within the formulas, one can explore different temporal aspects of the model behaviour. This approach enables us to investigate specific time intervals during which the variant model mirrors the behaviour of the original (preservation), performs superiorly (improvement), or exhibits deteriorated behaviour (intentional divergence).

5.6 Framework Implementation and Design Patterns

Implementing the Sibilla framework represents a significant contribution to the simulation and analysis of CAS. This section presents a detailed examination of the framework's architecture, design patterns, and implementation choices, enabling its powerful capabilities while maintaining flexibility and extensibility for future enhancements.

5.6.1 Simulation Module Implementation

The simulation module is a key part of the Sibilla framework and could be considered the cornerstone of the entire complex framework. This section comprehensively analyses the implementation details, architectural decisions, and design patterns employed throughout the system.

Core Architecture Overview

The simulation module is built around a central orchestrator class called:
`SimulationEnvironment` serves as the primary entry point and coordinator for all simulation operations. This class exemplifies the Facade design pattern by providing a simplified interface that shields clients from the underlying complexity of the simulation subsystem. The environment manages multiple aspects of simulation execution, including state management, execution strategies, monitoring, and data collection.

The simulation architecture in Sibilla is built upon several interconnected components that work together to enable flexible and powerful simulation capabilities:

- **SimulationEnvironment**: The facade and primary coordinator of the simulation system that manages the simulation lifecycle, configuration settings, and data col-

lection. It provides the high-level API for executing simulations and orchestrates interactions between other components.

- **ModelDefinition**: The factory interface responsible for creating Model instances and managing model parameters. It handles the lazy initialisation of model components and maintains the separation between model definition and execution concerns.
- **Model**: The interface representing the actual system being simulated. It defines the core behaviours and state transitions characterising the system under study, providing methods for state progression and measurement collection.
- **SimulationManager**: The component responsible for executing simulation tasks. It controls how simulation units are processed and coordinated through its AbstractSimulationManager base class and specialised implementations like SequentialSimulationManager.
- **SimulatorCursor**: The interface that provides the core abstraction for stepping through a simulation. It manages simulation progression, state transitions, and timing, supporting discrete and continuous time models.
- **SimulationUnit**: The encapsulation of a single simulation run, containing the initial state supplier, sampling mechanisms, and stopping conditions. It packages all necessary components for executing one iteration of a simulation.
- **SamplingHandler**: The component responsible for collecting and processing simulation data. It implements different statistical sampling strategies and manages data collection during simulation execution.
- **State**: The base interface defining the contract for simulation states, with specialised implementations for different modelling formalisms and requirements like immutability.
- **SimulationMonitor**: The interface providing monitoring and control capabilities, including progress tracking, cancellation support, and performance monitoring.
- **TrajectoryCollector**: The component that captures and stores complete simulation trajectories enables post-simulation analysis and visualisation.
- **SimulationStatus**: an enumeration that provides a clean mechanism for managing simulation lifecycle states while ensuring thread-safety and consistency in concurrent execution scenarios

Design Pattern Implementation

Factory Pattern The architecture uses the Factory Method pattern through the `SimulationManagerFactory` interface. This pattern provides a flexible mechanism for instantiating appropriate simulation managers without coupling the core simulation logic to specific implementation details. The factory pattern enables runtime selection of different execution strategies while maintaining a consistent interface for simulation creation.

The factory implementation allows the system to switch transparently between sequential and parallel execution models, with each factory implementation providing specialised manager instances optimised for different execution contexts.

Strategy Pattern The `SimulationManager` interface defines the contract for different execution strategies, allowing the system to accommodate various execution models. This pattern enables the seamless integration of different simulation approaches, including sequential execution, parallel processing, and potentially distributed computation.

Each concrete strategy implementation encapsulates its specific execution logic while adhering to the common interface, facilitating easy extension and modification of simulation behaviours.

Observer Pattern The monitoring subsystem implements the Observer pattern through the `SimulationMonitor` interface. This pattern enables loose coupling between the simulation execution logic and monitoring components, allowing for the flexible integration of different monitoring strategies. The pattern implementation supports synchronous and asynchronous monitoring approaches, with built-in support for cancellation and progress tracking.

Monitors can register property change listeners to receive notifications about simulation events, enabling reactive monitoring and control mechanisms.

Template Method Pattern The `AbstractSimulationManager` class implements the Template Method pattern, providing a skeletal implementation of the simulation execution process while allowing subclasses to customise specific aspects of the execution flow. This pattern ensures consistency in the simulation lifecycle while enabling specialisation for different execution contexts.

The template implementation defines the core simulation workflow, including initialisation, execution, and cleanup phases, while delegating specific implementation details to concrete subclasses. This approach maintains a consistent execution model while supporting diverse implementation requirements.

5.6.2 Analysis Module

The analysis module in Sibilla represents a sophisticated component designed to examine and verify properties of collective adaptive systems. This module serves as the framework's analytical backbone, providing temporal logic verification, reachability analysis, and statistical model checking capabilities. Integrating various analysis techniques into a cohesive system enables deep insights into system behaviours and properties.

Core Architecture Overview

The analysis module architecture is built on a sophisticated layered design that combines multiple specialised components working in concert. At its foundation lies the ModuleEngine class, which is the central coordinator for all analysis operations. This engine doesn't simply execute analyses - it orchestrates a complex interplay between different analysis components, managing state transitions, coordinating with the simulation subsystem, and processing analysis results.

The architecture employs a hierarchical structure centered around several key components:

The SibillaModule interface defines the core contract for all analysis capabilities, establishing a consistent API for different types of analyses. This interface is complemented by AbstractSibillaModule, which provides a robust foundation of standard functionality that specific modules extend. This layered approach allows specialised modules like LIOModelModule, PopulationModelModule, and YodaModelModule to focus on their unique requirements while inheriting standard analysis capabilities.

An exceptionally sophisticated aspect of the architecture is its monitoring subsystem. Built around the QualitativeMonitor and QuantitativeMonitor interfaces, this subsystem enables both boolean and quantitative verification of temporal properties. The monitoring system employs a streaming architecture for processing temporal properties, allowing efficient evaluation of complex properties over long simulation traces.

The state management system is another crucial architectural component. Rather than maintaining static state representations, it implements a sophisticated state tracking mechanism that supports both deterministic and stochastic state transitions. This flexibility is essential for analysing systems with complex probabilistic behaviours.

At the highest level, the SibillaRuntime class is the master coordinator, managing module selection and execution of analysis commands. This class implements a comprehensive command processing system that routes analysis requests to appropriate modules while maintaining overall consistency.

Design Pattern used

Factory Pattern The analysis module employs the Factory pattern extensively through several key implementations. The primary factory, StlMonitorFactory, serves as a sophisticated monitor creation system that manages the instantiation of qualitative and quantitative monitors. This factory maintains three critical registries: qualitative monitor factories, quantitative monitor factories, and parameter mapping functions.

This factory implementation's handling of parameterised analyses is particularly noteworthy. Rather than creating static monitor instances, it generates templates that

can be instantiated with different parameter values. This approach enables the flexible reuse of analysis specifications while maintaining strict type safety and parameter validation.

Visitor Pattern The module implements a comprehensive visitor-based approach for formula evaluation through several specialised evaluator classes. The StlQuantitativeMonitorEvaluator and StlQualitativeMonitorEvaluator work in tandem with the StlMonitorExpressionEvaluator to process temporal logic formulas. This visitor implementation goes beyond simple traversal—it handles complex semantic analysis and monitor construction while maintaining a clean separation between formula structure and evaluation logic.

The visitor pattern proves particularly powerful in handling the recursive nature of temporal logic formulas. Visitors can traverse the formula structure while building appropriate monitors for each subformula, maintaining crucial semantic relationships and timing constraints.

Template Method Pattern The AbstractSibillaModule implements a sophisticated template method pattern that defines the skeleton of analysis algorithms while allowing specific modules to override certain steps. This pattern ensures consistency across different analysis types while enabling specialised behaviour where needed. The template implementation carefully manages the analysis lifecycle, including initialisation, execution, and cleanup phases.

Composite Pattern The temporal logic formula representation employs the composite pattern to build complex formulas from simpler components. This pattern allows the creation of sophisticated analysis specifications while maintaining a uniform interface for evaluation. Each temporal operator (globally, eventually, until) and boolean operator (or not) is implemented as a separate monitor class that can be composed into more complex monitors.

5.6.3 Model Module

The model module represents a fundamental cornerstone of the Sibilla framework, providing the foundation for specifying, generating, and executing simulations of complex systems. This section comprehensively analyses its implementation details, architectural decisions, and operational mechanisms.

Core Architecture Overview

The model module architecture in Sibilla is built around several interconnected components that transform high-level system specifications into executable simulations.

At the heart of the architecture lies the `ModelDefinition` interface, which serves as the primary entry point and coordinator for all model operations. This interface is a sophisticated factory that bridges the gap between specification and execution through three fundamental responsibilities: model instantiation, configuration management, and state generation. Generic typing ensures type safety by binding model definitions to their corresponding state types.

The `ModelDefinition<S extends State>` coordinates the creation of concrete model instances and manages the lifecycle of model parameters. It implements lazy initialisation of model components and maintains a clear separation between definition and execution concerns, providing methods for creating model instances, managing configurations, and generating initial states.

The generation pipeline processes model specifications through three main stages. First, ANTLR4-based parsing handles the initial transformation of textual specifications through dedicated lexers and parsers for each model type. These parsers generate grammar-specific parse trees while performing syntax validation and error reporting. The second stage involves model generation, where visitor pattern implementations traverse the parse trees to perform semantic validation, resolve references, construct components, and verify type constraints. Finally, the configuration management stage handles parameter evaluation, initial state generation, and the registration of measures and predicates.

The executable form of a model specification is represented by the `Model` interface. This interface defines the core behaviours and state transitions that characterise the system under study, including methods for creating simulation cursors, defining and computing measures, managing state transitions, collecting simulation data, and handling predicates and properties.

State management employs a sophisticated hierarchy built around the `State` interface. The base interface establishes minimal requirements, while specialised interfaces like `ImmutableState` and `IndexedState` provide additional guarantees and capabilities. `ImmutableState` ensures state immutability for parallel execution scenarios, while `IndexedState` enables efficient access to individual elements within collective states. This hierarchical approach allows different modelling paradigms to represent the state most appropriately while maintaining consistent interfaces.

The analysis framework provides comprehensive capabilities through an integrated system of measures, predicates, sampling mechanisms, and trajectory analysis tools. Measures compute quantitative properties of states, while predicates evaluate Boolean conditions. Together, they enable sophisticated analysis of system behaviour during simulation.

Design Patterns Used

The model module implementation leverages several key design patterns to achieve flexibility, maintainability, and extensibility.

Factory Pattern The Factory pattern manifests primarily through the `ModelDefinition` interface and its implementations. This pattern encapsulates model creation logic while supporting different model types and flexibility of runtime configuration. The factory implementation enables dynamic model type selection and parameter-based configuration while supporting specialised optimisation strategies and component lifecycle management. This approach ensures type-safe model instantiation while enabling lazy initialisation of components.

Template Method Pattern Abstract base classes like `AbstractModelDefinition` implement the Template Method pattern to define skeletal processing algorithms while allowing customisation of specific steps. This pattern ensures consistent behaviour across different model types while enabling specialised implementations to modify particular aspects of the workflow. Template methods standardise model generation workflows, state management operations, configuration processing, and resource management, maintaining architectural integrity throughout the system.

Visitor Pattern The model generation process extensively uses the Visitor pattern through ANTLR4 generated visitors and custom syntax tree processors. This pattern enables the separation of algorithms from structure, allowing flexible processing strategies and extensible validation rules. Semantic analysers and component generators leverage this pattern to maintain a clean separation of concerns while supporting sophisticated model construction and validation processes.

5.6.4 Synthesis Module

A key part of the Sibilla framework is the synthesis module, which provides sophisticated parameter synthesis capabilities through surrogate-based optimisation techniques. This section comprehensively analyses the implementation details, architectural decisions, and design patterns employed throughout the system.

Core Architecture Overview

The synthesis module is built around a central orchestrator class called `Synthesizer`, which serves as the primary entry point and coordinator for all synthesis operations. This class exemplifies both the Facade and Builder design patterns, providing a simplified interface that shields clients from the surrogate-based optimisation subsystem's underlying complexity while enabling the flexible synthesis configuration construction. The synthesizer manages multiple aspects of the synthesis process, including surrogate model training, optimisation strategy execution, sampling coordination, and result collection.

The synthesis architecture in Sibilla is built upon several interconnected components that work together to enable flexible and powerful synthesis capabilities:

- **Synthesizer**: The synthesis system's central facade and primary coordinator that manages the synthesis lifecycle, configuration settings, and result collection. It provides the high-level API for executing synthesis tasks and orchestrates interactions between other components.
- **SynthesisTask**: The abstract base class defining the core structure for synthesis operations. Its concrete implementations, like OptimalFeasibilityTask and ComparisonTask encapsulate different types of synthesis objectives and strategies.
- **SynthesisStrategy**: The component responsible for configuring the synthesis approach, including search space definition, sampling strategy selection, surrogate model choice, and optimisation algorithm settings.
- **SurrogateModel**: The interface representing the machine learning models used for function approximation. It provides different approaches to surrogate modelling through implementations like RandomForestModel, GradientTreeBoostModel, and RBFNetworkModel.
- **OptimizationTask**: The component executing optimisation algorithms. It controls how the parameter space is explored through implementations like PSOTask, MADSTask, and SimulatedAnnealingTask.
- **SamplingTask**: The interface that provides the core abstraction for generating sample points in the parameter space. It supports different sampling strategies through implementations like LatinHyperCubeSampling, RandomSampling, and FullFactorialSampling.
- **DataSet**: The component responsible for managing training data, including splitting into training and test sets, data transformation, and efficient storage and retrieval of samples.

- **HyperRectangle**: The class defining the search space for optimisation, managing parameter bounds and constraints through a collection of intervals.
- **SynthesisRecord**: The comprehensive record-keeping component that captures all aspects of a synthesis operation, including configuration, results, and performance metrics.
- **InfillStrategy**: The component responsible for adapting the search space based on optimisation progress, implementing intelligent exploration-exploitation balance.

Design Patterns Used

Builder Pattern The `Synthesizer` class implements the Builder pattern to manage the construction of synthesis configurations. This pattern is crucial given the numerous parameters and optional components involved in setting up a synthesis process. The implementation provides a fluent interface for configuring the optimisation algorithm, surrogate model, sampling strategy, and various operational parameters. This pattern ensures that synthesis configurations can be constructed step by step, with each component adequately initialised and validated.

Factory Pattern The architecture implements a comprehensive factory-based creation system throughout all significant components. This pattern manifests in several key areas:

The `SynthesisTaskFactory` serves as the primary factory for creating synthesis tasks, enabling runtime selection of different synthesis approaches while maintaining a consistent interface for task creation. The factory implementation allows transparent switching between optimal feasibility and comparison tasks.

The surrogate modelling system employs the `SurrogateFactory` interface, with specialised implementations for each modelling approach:

- `RandomForestFactory` for creating random forest models
- `GradientTreeBoostFactory` for gradient boosting models
- `RBFNetworkFactory` for radial basis function networks

The optimisation component uses `OptimizationAlgorithm` as its factory interface, with implementations including:

- `PSOAlgorithm` for particle swarm optimisation
- `SimulatedAnnealingAlgorithm` for simulated annealing
- `LTMADSAlgorithm` and `OrthoMADSAlgorithm` for MADS variants

The sampling subsystem is built around the `SamplingFactory` interface, with implementations such as:

- `LatinHyperCubeSamplingFactory` for LHS sampling
- `RandomSamplingFactory` for random sampling
- `FullFactorialSamplingFactory` for full factorial sampling

Strategy Pattern The Strategy pattern manifests in several key areas of the implementation, most notably in the surrogate modelling and optimisation components. The `SurrogateModel` interface defines the contract for different surrogate modelling approaches, allowing the system to accommodate various modelling techniques without modifying the client code.

Each concrete strategy implementation encapsulates its specific modelling or optimisation logic while adhering to the common interface. The strategy pattern enables easy extension of the framework with new surrogate models or optimisation algorithms.

Registry Pattern and Singleton The synthesis module employs a Registry pattern implementation through specialised registry classes:

- `SurrogateModelRegistry` manages available surrogate model implementations
- `OptimizationAlgorithmRegistry` coordinates optimisation algorithm availability
- `SamplingStrategyRegistry` handles sampling strategy registration

Each registry implements the Singleton pattern and provides centralised management of available implementations while supporting runtime registration of new components. This pattern enables flexible configuration and extension of synthesis capabilities while maintaining consistent access to registered implementations.

Template Method Pattern The `AbstractSurrogateModel` class implements the Template Method pattern, providing a skeletal implementation of the surrogate modelling process. This pattern defines the core workflow for surrogate model training and prediction while allowing specific implementations to customise key aspects such as:

- Model initialisation and configuration
- Training data preprocessing
- Model fitting procedures
- Prediction mechanisms

This approach ensures consistency in the surrogate modelling lifecycle while enabling specialisation for different modelling approaches. The template implementation maintains a unified structure while supporting diverse implementation requirements across various types of surrogate models.

The pattern is also evident in the `AbstractMADSTask` class, which provides a template for MADS optimisation implementations while allowing specialisation in concrete MADS variants.

Composite Pattern The sampling subsystem utilises the Composite pattern in handling complex sampling strategies, particularly in implementing multidimensional sampling approaches. This pattern enables the composition of sampling strategies for different parameters while maintaining a consistent interface for sample generation.

5.6.5 Synthesis Specification

Our framework employs a structured YAML-based configuration system that consists of two primary components: the Synthesis Strategy and the Synthesis Task. This modular approach allows for the flexible definition of optimisation parameters and problem-specific requirements.

Configuration Structure Overview

The framework's configuration system employs a carefully structured YAML specification that divides the synthesis process into two fundamental sections: Synthesis Strategy and Synthesis Task. This architectural decision reflects the separation of concerns principle, where the optimisation methodology is decoupled from the problem-specific requirements, enabling greater flexibility and reusability of components.

The Synthesis Strategy section serves as the optimisation engine of the framework, providing a comprehensive set of components that guide the search process. This section maintains consistency across different synthesis tasks, establishing a unified approach to optimisation while allowing for the customisation of individual components. It encompasses the search space definition, which bounds the exploration area; the sampling strategy, which determines initial point distribution; the surrogate model configuration, which aids in efficient exploration; the optimisation algorithm settings, which guide the search process; and the infill criteria, which control the iterative refinement of solutions.

Conversely, the Synthesis Task section adapts to the specific type of analysis being performed. This adaptability is crucial as it allows the framework to address two distinct classes of problems: optimal feasibility and comparative analysis. Each task requires different specifications and evaluation criteria, yet they both leverage the same underlying optimisation infrastructure defined in the Synthesis Strategy section.

This architectural design achieves several key objectives: First, it promotes modularity by allowing components to be modified or replaced without affecting the overall structure. Second, it enables the framework to handle diverse synthesis tasks while maintaining a consistent optimisation approach. Third, it provides a clear separation between the mechanics of optimisation and the specifics of the problem being solved. Finally, it facilitates extensibility, allowing new task types to be added while leveraging the existing optimisation infrastructure.

Synthesis Strategy Specification

The Synthesis Strategy section (`synthesisStrategy`) defines the optimisation and search parameters, comprising five key components:

1. **Search Space** (*Required*): Defines the parameter bounds and constraints for optimisation:
    - Parameter name, lower bound, and upper bound for each variable
    - Optional constraint definitions for the search space
2. **Sampling**: Configures the initial sampling strategy:
    - Sampling method (e.g., Latin Hypercube Sampling, Random)

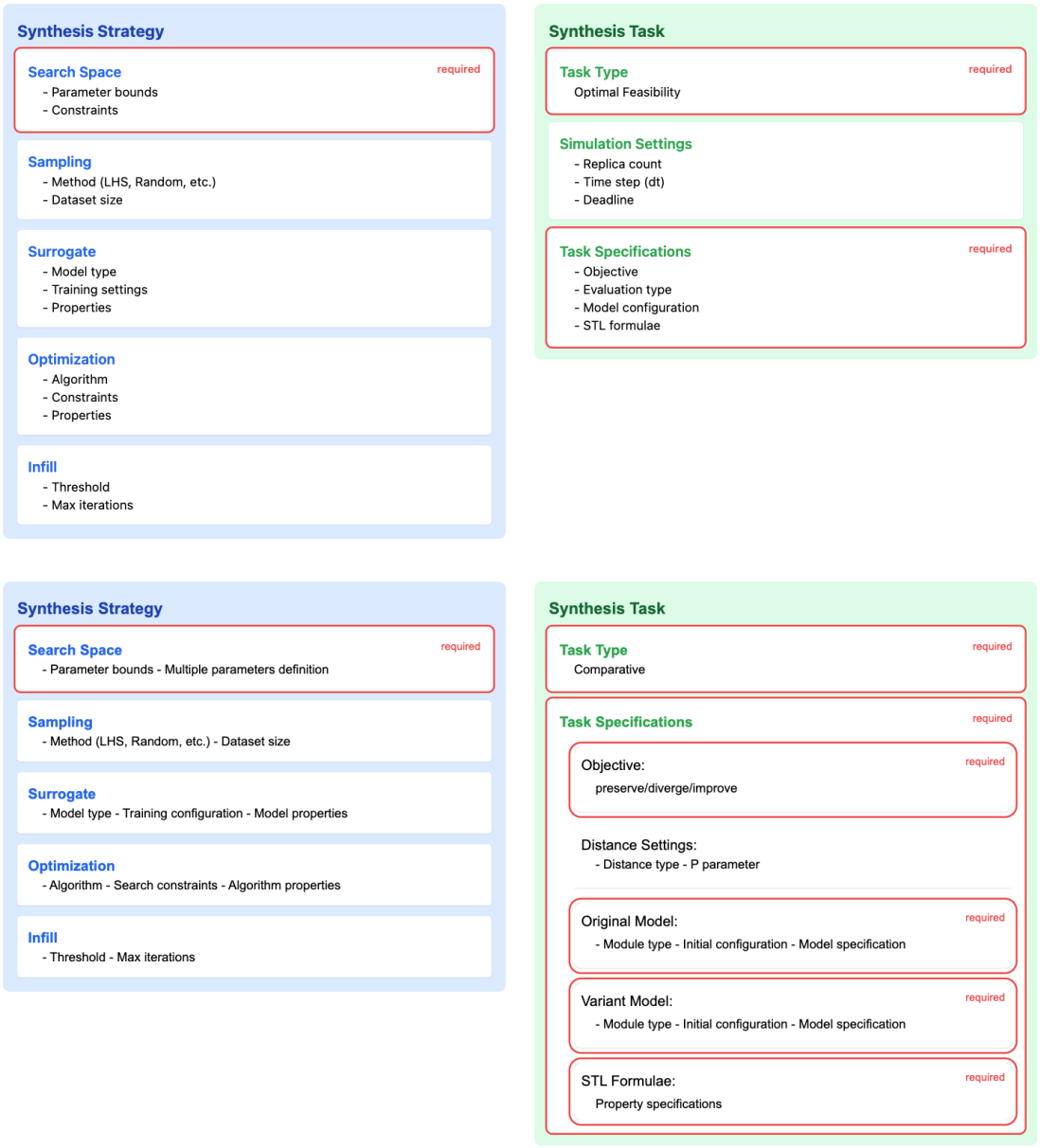


Figure 5.7: *Synthesis Configuration structures: The diagram illustrates our framework's two primary configuration patterns. The Optimal Feasibility structure (top), the Variant Model Adjustment structure (bottom) extends the framework to handle comparative analysis between model variants. The unified structure of both patterns demonstrates the framework's ability to accommodate different synthesis goals while maintaining a consistent optimisation approach through the shared synthesis strategy components.*

- Dataset size for initial exploration

3. **Surrogate**: Specifies the surrogate model configuration:
   - Model type and training settings
   - Model-specific properties (e.g., number of trees, maximum depth)
4. **Optimisation**: Defines the optimisation algorithm settings:
   - Algorithm selection and properties
   - Constraint definitions
   - Algorithm-specific parameters
5. **Infill**: Controls the iterative optimisation process:
   - Threshold values for convergence
   - Maximum iteration limits

Synthesis Task Specification

The Synthesis Task section (`synthesisTask`) defines the problem-specific requirements and varies based on the type of analysis:

Optimal Feasibility Configuration

For optimal feasibility tasks:

- **Task Type**: Specified as `optimalFeasibility`
- **Simulation Settings**: Includes replica count, time step, and deadline
- **Task Specifications**: Defines objective function, evaluation type, model configuration, and property specifications

Comparative Analysis Configuration

For model comparison tasks:

- **Task Type**: Specified as `comparative`
- **Task Specifications**: Includes:
  - Objective (`preserve`, `diverge`, or `improve`)
  - Distance metrics and parameters
  - Original and variant model configurations
  - Property specifications

5.6.6 Synthesis Output Format

The synthesis framework provides a systematic way to report its results through a structured output format that can adapt to different levels of detail based on user needs. This output system is designed to present information concisely or comprehensively, controlled by a verbosity flag.

At its core, the basic output focuses on the essential outcomes of the synthesis process. It always presents three critical pieces of information representing the optimisation's fundamental results. First, it shows **the optimal coordinates** discovered during the synthesis process, presenting the specific values found for each parameter that produced the best results. Second, it reports the **objective function value** achieved at these optimal coordinates, representing the actual performance or quality of the solution. Third, it displays the **surrogate function's predicted value** at these exact coordinates, which helps validate how well the surrogate model approximates the true objective function at the solution point.

When more detailed information is needed, the verbose output expands upon this basic format by including comprehensive details about the entire synthesis process. This extended output begins with the same core results but continues with several additional categories of information. The algorithm configuration section details the choices made for the optimisation process, including which optimisation algorithm was selected, what type of surrogate model was employed, and which sampling strategy was used to explore the parameter space.

The verbose output also includes important optimisation parameters that shaped the synthesis process. These parameters indicate whether the process utilised a surrogate model, whether the problem was formulated as a minimisation or maximisation task, how many samples were used to build the surrogate model, and what proportion of the data was allocated for training. This information is crucial for understanding how the synthesis was conducted and reproducing the results.

Furthermore, the verbose output provides detailed information about the search space configuration, showing the boundaries within which the optimisation occurred. It includes any specific properties or settings used for the optimisation algorithms, such as iteration counts or population sizes. The output concludes with **performance metrics** that evaluate the quality of the surrogate model, presenting both in-sample metrics that show how well the model fits the training data and out-of-sample metrics that indicate how well it generalises to new points.

This dual reporting system serves different needs within the synthesis workflow. The basic output provides quick access to essential results for rapid verification or integration with other tools. In contrast, the verbose output supports detailed analysis, debugging, and validation of the entire synthesis process.

5.6.7 Default Synthesis Configuration

The synthesis framework implements a sophisticated configuration system for robust optimisation performance across diverse scenarios. When no specific configuration is provided, the framework employs three default components: Particle Swarm Optimisation (PSO) as the optimisation algorithm, Random Forest as the surrogate model, and Latin Hypercube Sampling (LHS) for parameter space exploration. Each component is initialised with carefully calibrated default settings that balance exploration efficiency with computational overhead.

These default configurations can be customised through a comprehensive set of identifiers, each mapping to specific parameters within their respective components. The following sections detail these mappings and their default values, providing a complete reference for framework customisation.

Optimisation Algorithm Parameters

Mesh Adaptive Direct Search (MADS) The MADS algorithm operates through a dynamic mesh structure that adapts to the local topology of the objective function. The mesh size parameter ($\delta_{mesh} = 1.0$) and poll size parameter ($\delta_{poll} = 1.0$) establish the initial granularity of the search. A mesh refinement parameter ($\tau = 0.25$) governs the rate at which the mesh adapts, while the convergence criterion ($\epsilon = 1.0E - 50$) determines when the algorithm terminates. Each iteration examines 20 search points, with a maximum of 500 iterations ensuring thorough exploration while maintaining computational efficiency.

- `delta_mesh` ↔ Initial mesh size (Default: 1.0)
- `delta_poll` ↔ Initial poll size (Default: 1.0)
- `tau` ↔ Mesh refinement rate (Default: 0.25)
- `epsilon` ↔ Convergence threshold (Default: 1.0E-50)
- `iteration` ↔ Maximum iterations (Default: 500)
- `search_points` ↔ Points evaluated per iteration (Default: 20)

Particle Swarm Optimisation (PSO) The PSO algorithm employs a carefully balanced set of parameters to guide particle movement through the search space. An inertia weight ($\omega = 0.72984$) maintains momentum, while identical self-confidence and swarm confidence coefficients ($c_1 = c_2 = 1.49617$) create equilibrium between individual and collective behaviour. A swarm population of 100 particles explores the space over 100 iterations, providing comprehensive coverage while avoiding excessive computational overhead. The following identifiers can be used to modify default settings:

- `inertia` ↔ Inertia weight (Default: 0.72984)
- `self_confidence` ↔ Cognitive component coefficient (Default: 1.49617)
- `swarm_confidence` ↔ Social component coefficient (Default: 1.49617)
- `particles_number` ↔ Swarm population size (Default: 100)
- `iteration` ↔ Maximum iterations (Default: 100)

Simulated Annealing The simulated annealing process begins at a high temperature ($T_0 = 1000.0$) to promote extensive exploration, gradually cooling at $\alpha = 0.003$ over 1000 iterations. The linear cooling schedule ensures a steady transition from exploration to exploitation phases, maintaining a balanced search strategy throughout the optimisation process. The following identifiers can be used to modify default settings:

- `initial_temperature` ↔ Starting temperature (Default: 1000.0)
- `cooling_rate` ↔ Temperature reduction rate (Default: 0.003)
- `max_iterations` ↔ Maximum iterations (Default: 1000)
- `cooling_schedule` ↔ Temperature reduction pattern (Default: "linear")

Surrogate Parameters

Random Forest Architecture The RF surrogate model constructs an ensemble of 500 decision trees, each with a maximum depth of 200 levels. Feature selection at each split considers one-third of available features, rounded down but never less than one. The node population is regulated by setting the maximum nodes to one-fifth of the sample size, with a minimum node size of 5 samples ensuring statistical relevance. The model utilises a full sampling of the training data to maximise information usage. The following identifiers can be used to modify default settings:

- `trees` ↔ Number of trees in ensemble (Default: 500)
- `max_depth` ↔ Maximum tree depth (Default: 200)
- `mtry` ↔ Features per split (Default: max(features/3, 1))
- `max_nodes` ↔ Maximum leaf nodes (Default: samples/5)
- `node_size` ↔ Minimum samples per node (Default: 5)
- `rate` ↔ Subsampling ratio (Default: 1.0)

RBF Network Structure The Radial Basis Function network adapts its complexity to the problem dimension by employing ten neurons per input feature. This scaling ensures sufficient model capacity while preventing overfitting. The network operates on raw data values, avoiding normalisation to preserve the natural scale relationships within the data. The following identifiers can be used to modify the default settings:

Radial Basis Function Network (RBF)

- `neurons` ↔ Number of neurons (Default: features × 10)
- `normalised` ↔ Input normalisation flag (Default: false)

Gradient Tree Boost Framework The Gradient Tree Boost implementation employs the Least Absolute Deviation loss function to perform robust regression. The model consists of 500 trees, each limited to 20 levels and six nodes, creating a balanced ensemble structure. A conservative learning rate of 0.05 ensures stable convergence, while a 70% subsampling ratio for each boosting iteration introduces beneficial randomisation. The following identifiers can be used to modify default settings:

- `loss` ↔ Loss function type (Default: "LeastAbsoluteDeviation")
- `trees` ↔ Number of trees (Default: 500)
- `max_depth` ↔ Maximum tree depth (Default: 20)
- `max_nodes` ↔ Maximum leaf nodes (Default: 6)
- `node_size` ↔ Minimum samples per node (Default: 5)
- `shrinkage` ↔ Learning rate (Default: 0.05)
- `sample_rate` ↔ Subsampling ratio (Default: 0.7)

Data Management Parameters

The framework initiates with 100 samples to establish a foundational understanding of the objective function landscape. This data is partitioned with 80% allocated to model training, ensuring robust surrogate model development, while the remaining 20% serves for validation purposes, enabling reliable assessment of model performance.

5.6.8 Libraries Used

The project's backend uses different libraries that streamline various development aspects, including configuration management, statistical computations, machine learning, and data manipulation. This section provides an overview of the key libraries used and their specific roles and features.

SnakeYAML SnakeYAML is a Java-based library designed for parsing and emitting YAML (YAML Ain't Markup Language) configurations. It simplifies configuration management, allowing for the seamless loading of model definitions, synthesis tasks, and strategies from YAML files. In Section 5.5, this capability is used by the `SynthesisEvaluator` class to parse task specifications, which are crucial for defining optimisation goals and constraints. More details about SnakeYAML can be found on its Maven Repository page [2].

Smile Smile (Statistical Machine Intelligence and Learning Engine) is a comprehensive library for statistical analysis, machine learning, and advanced data processing. It provides various algorithms, including clustering, classification, regression, and natural language processing tools. Smile also features utilities for visualisation and performance evaluation, making it a versatile choice for implementing machine learning workflows in the back end. It is extensively used in Section 5.5 during surrogate modelling, where models like Random Forests, Gradient Tree Boost, and Radial Basis Function Networks are implemented to approximate the behaviour of the underlying stochastic models. Further details can be found on the Smile website [3] or its Maven Repository page [4].

Tablesaw Tablesaw is a Java library designed for data manipulation and analysis. It provides an intuitive API for managing structured data, which is particularly valuable for handling simulation results and creating training datasets during the sampling phase of the synthesis workflow (Section 5.5). Tablesaw simplifies operations like filtering, aggregation, and statistical summarisation, making preparing data for surrogate modelling and optimisation easier. More information about Tablesaw is available on its GitHub repository [5] or Maven page [6].

Apache Commons Apache Commons is a widely-used collection of Java libraries that provide reusable components for various tasks. The `Apache Commons Math` library is used extensively across the entire tool for various numerical operations. The Apache Commons Math library, in particular, ensures precision and reliability in mathematical calculations, enhancing the system's robustness. More information can be accessed on the Apache Commons website [7].

By integrating these libraries, the back end leverages well-tested and efficient tools to handle diverse requirements, reducing development time and ensuring high perfor-

[2] `https://mvnrepository.com/artifact/org.yaml/snakeyaml`
[3] `https://haifengl.github.io/`
[4] `https://mvnrepository.com/artifact/com.github.haifengl/smile-core`
[5] `https://github.com/jtablesaw/tablesaw`
[6] `https://mvnrepository.com/artifact/tech.tablesaw/tablesaw-core`
[7] `https://commons.apache.org/`

mance. Each library has been selected based on unique capabilities, contributing to a cohesive and scalable architecture.

5.7 Sibilla extendability

Sibilla is a modular and extensible framework designed to facilitate complex systems' modelling, simulation, analysis, and synthesis. Its architecture emphasises flexibility, allowing users to adapt and expand its functionalities to accommodate new requirements. This section provides a comprehensive guide to extending Sibilla, focusing on the main components of the backend.

Developers can introduce new components while maintaining compatibility with the framework's core principles.

5.7.1 Add a new model

Adding a new model to Sibilla follows a structured approach that ensures proper integration with the framework's core components. The process involves six key steps, each building upon the previous to create a fully functional model implementation.

The first step involves creating a new class that implements the State interface. This class represents the fundamental state structure of your model.

1. Define State

Create a new class implementing the State interface to represent your model's state.

```
public class NewModelState implements State {
    // State methods
}
```

The Model implementation defines your system's behavioural logic. Extending AbstractModel, you inherit common functionality while implementing specific transition rules and measure definitions. This class serves as the core engine for your model's simulation behaviour.

2. Implement Model

Extend AbstractModel to define model behaviour and state transitions.

```
public class NewModel extends AbstractModel<NewModelState> {
    // Define measures
    // Implement state transitions
}
```

The ModelDefinition class handles the creation and configuration of model instances. This class bridges the gap between user-provided configurations and the internal model representation, ensuring proper initialisation of all components.

3. Create Model Definition

Extend AbstractModelDefinition to handle model creation and configuration.

```
public class NewModelDefinition
    extends AbstractModelDefinition<NewModelState> {
    @Override
    public Model<NewModelState> createModel() {
        // Model creation logic
    }
}
```

The Module class integrates your model into Sibilla's runtime environment. This implementation defines how your model interacts with the broader framework, including command handling and resource management.

4. Build Module

Create a module extending AbstractSibillaModule to integrate with Sibilla.

```
public class NewModelModule
    extends AbstractSibillaModule {
    @Override
    public String getModuleName() {
        return "new_model";
    }
}
```

The Parser component handles the translation of model specifications from external representation to internal data structures. This implementation should include robust error handling and validation mechanisms. In Sibilla, each model type has its domain-specific language formalism. The framework extensively uses ANTLR4 (ANother Tool for Language Recognition) to implement these language parsers.

5. Create Parser

Implement parsing logic for your model's configuration format.

```
public class NewModelParser {
    public NewModelDefinition parse(String input) {
        // Parsing logic
    }
}
```

The final step involves registering your module with Sibilla's core framework and implementing any necessary configuration handlers. This ensures your model is integrated and accessible through Sibilla's interfaces.

6. Register & Configure

Register your module with Sibilla and implement any custom commands.

```
public interface SibillaModule {
    List<SibillaModule> MODULES = List.of(
        // ...
        new NewModelModule()
    );
}
```

This structured approach allows new models to be seamlessly integrated into the Sibilla framework while maintaining consistency with its architectural principles. Each step builds upon the previous ones, creating a cohesive implementation that fully leverages Sibilla's simulation and analysis capabilities. The modular design ensures that new models can be added without affecting existing implementations. At the same time, the strong typing system helps prevent runtime errors and ensures type safety throughout the modelling process.

Optional: Custom SimulatorCursor Implementation

For models requiring specialized simulation control logic, implementing a custom SimulatorCursor provides fine-grained control over the simulation execution process. This optional step is particularly useful when the default simulation mechanisms don't fully accommodate your model's specific requirements or when you need custom state transition handling.

The SimulatorCursor interface defines the contract for simulation execution control, managing the progression of simulation steps and state transitions. A custom implementation allows you to tailor the simulation behaviour to match your model's specific characteristics and requirements.

Optional: Implement Custom Cursor

Create a custom SimulatorCursor implementation for specialized simulation control.

```
public class NewModelSimulationCursor
        implements SimulatorCursor<NewModelState> {
    // Custom simulation control logic
    // State management
    // Step execution methods
}
```

The custom cursor implementation provides several key benefits:

- **Specialized State Transitions**: Custom logic for complex state transition mechanisms that may not be adequately handled by default implementations (for example, SLAM models with agent-based interactions, message handling, and discrete event processing at specific time intervals).
- **Performance Optimization**: Tailored execution strategies for models with specific performance requirements or constraints (such as models requiring batch processing of state updates or specialized memory management for large-scale simulations).
- **Custom Termination Conditions**: Implementation of model-specific termination criteria beyond standard terminal state detection. The default cursor terminates when no more transitions are possible, but custom cursors can implement additional termination criteria such as convergence conditions, resource depletion, or complex multi-agent consensus requirements.
- **Enhanced Control Flow**: Fine-grained control over simulation progression, including custom step handling and state management (particularly useful for models with non-uniform time steps or event-driven dynamics that differ from the standard continuous-time progression used by default cursors).

When implementing a custom cursor, ensure that your Model class returns instances of your custom cursor implementation rather than using the default simulation mechanisms. This integration allows your model to leverage the specialized simulation control while maintaining compatibility with Sibilla's broader simulation infrastructure.

Custom cursor implementations are particularly valuable for models with complex temporal dynamics, specialized stochastic processes, or unique state management requirements that benefit from direct control over the simulation execution process.

5.7.2 Add a new Analysis

Adding new analysis capabilities to Sibilla follows a structured approach that leverages the framework's modular architecture. The process integrates with existing simulation and monitoring components while maintaining consistency with established patterns.

The implementation approach can vary based on the complexity of the analysis - more straightforward analyses like reachability can be implemented directly in the module engine. In contrast, more complex analyses with multiple subtypes may benefit from a more structured interface-based approach.

The first step is optional and depends on the analysis's complexity. This step can be skipped for simple analyses like reachability or first passage time calculations. However, defining a dedicated interface provides better structure and extensibility for more complex analyses with multiple variants or implementations.

1. Optional Interface Definition

Determining a dedicated interface can provide better structure and extensibility for complex analyses that may have multiple implementations or variations. However, more straightforward analyses like reachability or first passage time can be implemented directly without this abstraction layer.

```
// Optional - for complex analysis types only
public interface NewAnalysis<S extends State> {

  // passing the model and the simulation environment.
    AnalysisResult analyze( ... ) {
      ...
    }

}
```

After optionally defining the interface, the next crucial step is integration with the ModuleEngine. This component serves as the core executor for all analyses in Sibilla. The implementation can be direct, as seen in more straightforward analyses like reachability, or through a dedicated analysis class for more complex cases. The ModuleEngine provides access to the **model**, **simulation environment**, and other necessary components for executing the analysis.

2. Integrate with ModuleEngine

Integration with the ModuleEngine is the core requirement for any analysis. This can be done directly (as with reachability) or through a dedicated analysis class.

```
public class ModuleEngine<S extends State> {
    // Direct analysis implementation example (like reachability)
    public double performSimpleAnalysis( ... ) {
        ...
        return computeResult();
    }

    // Or using dedicated analysis class for complex cases
    public AnalysisResult performComplexAnalysis( ... ) {
        NewAnalysis<S> analysis = new NewAnalysisImpl<>( ... );
        return analysis.analyze( ... );
    }
}
```

The module system integration is a critical step that makes the analysis available across all Sibilla's components. This involves extending the `SibillaModule` interface with the new analysis method and providing a default implementation in `AbstractSibillaModule`. This architecture allows individual model modules to override the implementation if they need model-specific behaviour while maintaining a consistent interface across the framework.

3. Module System Integration

First, add the analysis method to the SibillaModule interface to make it available across all modules:

```
public interface SibillaModule {
    // Add the new analysis method signature
    AnalysisResult newAnalysis( ... );
}
```

Then, implement the common functionality in AbstractSibillaModule:

```
public abstract class AbstractSibillaModule implements
    SibillaModule {

    @Override
    public AnalysisResult newAnalysis( ... ){
        return checkForLoadedDefinition().performNewAnalysis( ...
     );
    }
}
```

Model-specific modules can override this implementation if needed:

```
public class PopulationModelModule extends AbstractSibillaModule
    {

    @Override
    public AnalysisResult newAnalysis( ... ) {
        // Population-specific analysis logic
        ...
    }
}
```

The final step involves making the analysis accessible through Sibilla's main interface by extending the Sibilla Runtime class. This makes the analysis available to users through programmatic and interactive interfaces, ensuring proper validation of prerequisites (like loaded modules and simulation parameters) before execution.

4. Add Runtime Support

Extend the SibillaRuntime class to expose the analysis functionality.

```
public class SibillaRuntime implements CommandHandler {
    public AnalysisResult newAnalysis(
        AnalysisConfiguration config)
        throws CommandExecutionException {

        checkLoadedModule();
        checkDeadline();

        return currentModule.performAnalysis( ... );
    }
}
```

This structured approach ensures proper integration with Sibilla's existing architecture while maintaining flexibility in implementation. The implementation can be direct within the ModuleEngine for simpler analyses like reachability. The interface-based approach provides better structure and extensibility for more complex analyses with multiple variants or implementations. The framework's design accommodates both approaches while ensuring consistent access through the SibillaRuntime interface.

5.7.3 Add a new Synthesis Task

Adding new synthesis capabilities to Sibilla follows a structured approach similar to adding new analyses. The synthesis framework is built around tasks and strategies, where tasks define what to optimise and strategies define how to perform the optimisation. Currently, Sibilla supports two main synthesis tasks: Optimal Feasibility and Comparison Task, but the framework is designed to be extensible.

The first step in adding a new synthesis task is extending the abstract SynthesisTask class. This base class provides common functionality for all synthesis tasks, including parameter handling and integration with Sibilla's runtime environment.

1. Extend SynthesisTask

Create a new class extending the abstract SynthesisTask class:

```
public class NewSynthesisTask extends SynthesisTask {
    private final TaskParameters parameters;

    public NewSynthesisTask(
        // Parse task-specific parameters
    }

    @Override
    public SynthesisRecord execute(SibillaRuntime runtime)
        // Implementation of synthesis logic
    }
}
```

After creating the new synthesis task class must be registered with the framework's task factory system. This involves adding an entry to the TaskName enumeration and updating the SynthesisTaskFactory to handle the creation of the new task type. This registration step makes the new task type available through Sibilla's configuration system.

2. Register Task Type

Add your task type to the TaskName enumeration and update the SynthesisTaskFactory:

```
public enum TaskName {
    OPTIMAL_FEASIBILITY,
    COMPARATIVE,
    NEW_TASK;  // Add new task type

    public static TaskName getName(String name) {
        return valueOf(name.toUpperCase());
    }
}

public class SynthesisTaskFactory {
    public static SynthesisTask createTask(
        TaskName taskName,
        Map<String,Object> taskMap,
        SynthesisStrategy strategy) {

        return switch (taskName) {
            case OPTIMAL_FEASIBILITY ->
                new OptimalFeasibilityTask(taskMap, strategy);
            case COMPARATIVE ->
                new ComparisonTask(taskMap, strategy);
            case NEW_TASK ->
                new NewSynthesisTask(taskMap, strategy);
        };
    }
}
```

The core of any synthesis task lies in its execution method implementation. This is where the task's specific optimisation logic is defined, including setting up the model, creating the objective function, and configuring the synthesiser. The implementation should follow the pattern established by existing tasks like OptimalFeasibilityTask and ComparisonTask while adding any task-specific functionality.

3. Implement Task Logic

Define the core synthesis logic in the execute method:

```
@Override
public SynthesisRecord execute(SibillaRuntime runtime)
    throws CommandExecutionException {
    // Configure model parameters
    setupModel(runtime);

    // Define objective function
    ToDoubleFunction<Map<String,Double>> objective =
        createObjectiveFunction(runtime);

    // Create and configure synthesizer
    Synthesizer synthesizer = generateSynthesizer( ... );

    // Run synthesis
    synthesizer.searchOptimalSolution();
    return synthesizer.getLastSynthesisRecord();
}
```

This structured approach ensures that new synthesis tasks integrate seamlessly with Sibilla's existing framework.

5.7.4 Add a new Simulation Component

Extending Sibilla's simulation capabilities involves integrating new components that enhance or modify the simulation behaviour. This process follows a structured approach to maintain consistency with the framework's architecture while allowing for customisation of simulation execution, sampling, monitoring, and state management.

Sibilla's simulation capabilities can be extended through various entry points in the framework's architecture. While the simulation manager system provides one avenue for extension, particularly for specialised execution requirements, developers can extend the sampling system, monitoring capabilities, or state management according to their specific needs. The framework's modular design ensures that these extensions can be implemented independently or in combination.

These managers control how simulation tasks are executed and coordinated. By extending the AbstractSimulationManager class, developers can implement specialised execution strategies that cater to specific performance requirements or execution environments. This foundation class provides essential functionality while allowing customisation of the core execution logic.

1. Create Manager Extension

Extend AbstractSimulationManager to implement custom execution strategies.

```
public class CustomSimulationManager<S extends State>
        extends AbstractSimulationManager<S> {

    public CustomSimulationManager(
            RandomGenerator random,
            SimulationMonitor monitor) {
        super(random, monitor);
    }

    @Override
    protected void handleTask(SimulationTask<S> simulationTask) {
        // Custom task execution logic
    }

}
```

The sampling system provides an extension point for customising how data is collected and processed during simulations. By extending the StatisticSampling class, developers can implement specialised data collection strategies. This allows for custom statistical analysis, unique data processing approaches, or integration with external analysis tools. The sampling extension provides complete control over how simulation data is gathered and processed while maintaining compatibility with Sibilla's data management framework.

2. Implement Custom Sampling

Create a new sampling implementation by extending StatisticSampling.

```
public class CustomSampling<S extends State>
        extends StatisticSampling<S> {

    private final CustomStatistics[] data;

    public CustomSampling(int samples, double dt,
            Measure<? super S> measure) {
        super(measure, dt);
        this.data = new CustomStatistics[samples];
        init();
    }

}
```

Monitoring capabilities can be extended through custom implementations of the SimulationMonitor interface. This enables specialised tracking and control of simulation execution, allowing developers to implement custom progress reporting, performance monitoring, or integration with external monitoring systems. The monitor implementation can provide detailed insights into simulation progress and control simulation execution based on custom criteria.

3. Create Monitor Extension

Implement a custom simulation monitor for specialised tracking needs.

```
public class CustomSimulationMonitor implements SimulationMonitor
      {
    private final PropertyChangeSupport changes =
            new PropertyChangeSupport(this);
    private boolean cancelled = false;

    @Override
    public void startIteration(int i) {
        // Custom iteration start logic
    }

    @Override
    public void endIteration(int i) {
        // Custom iteration end logic
    }

    @Override
    public void endSimulation() {
        // Custom simulation end handling
    }

    @Override
    public boolean isCancelled() {
        return cancelled;
    }

    @Override
    public void registerPropertyChangeListener(
            PropertyChangeListener l) {
        changes.addPropertyChangeListener(l);
    }
}
```

The final step involves integrating these custom components with Sibilla's runtime environment. This integration ensures that custom implementations work seamlessly with the framework's existing functionality. The registration process makes custom components available through Sibilla's standard interfaces while maintaining the framework's type safety and error-handling capabilities. This step bridges the gap between custom implementations and the framework's core functionality.

4. Register Components

Integrate new components with Sibilla's runtime environment.

```
public class SimulationEnvironment {

    // Use custom components in simulation
    public <S extends State> void simulate(
            CustomSimulationMonitor monitor,
            RandomGenerator random,
            Model<S> model,
            Function<RandomGenerator,S> initialStateSupplier,
            CustomSampling<S> sampling,
            long iterations,
            double deadline)
            throws InterruptedException {

        SimulationManager<S> manager =
            new CustomSimulationManager<>(random, monitor);
        // Simulation execution logic
    }
}
```

The extension points provide flexibility while maintaining consistency with Sibilla's architectural principles. Each component can be customised independently, allowing targeted enhancements to specific aspects of the simulation process while preserving compatibility with other framework features.

CHAPTER 6

Case Studies

"The most exciting phrase to hear in science, which heralds new discoveries, is not 'Eureka!' but 'That's funny..."

Isaac Asimov

Sibilla is designed to facilitate adaptable modelling through parameterisation and monitoring capabilities. This design philosophy enables flexible model configuration and comprehensive analysis of system properties through measured indicators.

The models described in Section 5.2, from the fundamental LPM to the advanced SLAM, are all designed to be parameterizable and monitorable. This capability allows for model adaptation without core implementation modifications, systematic exploration of different configurations, and comprehensive sensitivity analyses.

6.1 Parameterised Population Models

We focus on population models for the subsequent case studies, which exemplify Sibilla's parameterisation capabilities. We extend the population model definition from Section 5.2.1 to incorporate parameterisation as follows:

Definition 61 (Parametrised Population Model)
A parameterised population model is a tuple $\mathcal{M} = (\mathbf{S}, \mathbf{X}, \mathcal{D}, x_0, \mathcal{T}, \mathcal{H})$ *where:*

- $\mathbf{S} = \{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_i, \ldots, \boldsymbol{s}_n\}$ *is the set of species (i.e. the different agent states);*
- $\mathbf{X} = (X_1, \ldots, X_i, \ldots, X_n)$ *is a vector of population variables, counting the number of entities of each kind, and taking values in* $\mathcal{D} \subseteq \mathbb{N}^n$*;*
- $\mathcal{D} = \bigtimes_{i=1}^{n} (\mathcal{D}_i)$ *(countable) state space;*
- $x_0 \in \mathcal{D}$ *is the initial state of the model;*
- $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_m\}$ *is a set of transition rules, describing how the system state can change, each rule is a tuple* $\tau = (\ell, \mathbf{u}, \boldsymbol{g}, \lambda)$
  - ℓ *is the label of the rule;*
  - $\mathbf{u}$*(sed), is a vector encoding how many agents are consumed in the reaction;*
  - $\mathbf{g}$*(enereted), is a vector encoding how many agents are produced in the reaction;*
  - $\lambda = \lambda(\mathbf{X}, \Theta)$ *is the rate function (*$\mathcal{D} \to \mathbb{R}_{\geqslant 0}$*), associating to each reaction the rate of an exponential distribution, depending on the global state of the model.*

  Each reaction can be seen as:

$$\tau \quad : \quad \mathbf{u}_1 \boldsymbol{s}_1 + \ldots + \mathbf{u}_n \boldsymbol{s}_n \xrightarrow{\lambda(\mathbf{X}, \Theta)} \mathbf{g}_1 \boldsymbol{s}_1 + \ldots + \mathbf{g}_n \boldsymbol{s}_n$$

- $\mathcal{H}$ *is a hyperrectangle defining the parameter space (as defined in Definition 34) where the parameter vector* Θ *takes values.*

6.2 Models Descriptions

6.2.1 The SIR Epidemiological Model

The Susceptible-Infected-Recovered (SIR) model is a fundamental mathematical framework in epidemiology, first introduced by Kermack and McKendrick in 1927. This model captures the essential dynamics of disease spread through a population by partitioning individuals into three distinct compartments based on their disease status. We formalise this model within the Population Modelling Framework as follows.

Let $\mathcal{M}_{\text{SIR}} = (\mathbf{S}, \mathbf{X}, \mathcal{D}, x_0, \mathcal{T})$ represent our SIR population model. The model's structure emerges from the biological progression of the disease through the population:

- The species set $\mathbf{S} = \{S, I, R\}$ represents the three possible states of individuals:
  - S (Susceptible): individuals who can contract the disease

  - I (Infected): individuals who have contracted the disease and can transmit it
  - R (Recovered): individuals who have recovered and gained immunity
- The population state $\mathbf{X} = (X_S, X_I, X_R)$ tracks the number of individuals in each compartment at any given time, where:
  - X_S counts susceptible individuals
  - X_I counts infected individuals
  - X_R counts recovered individuals

The state space $\mathcal{D} = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ reflects the discrete nature of population counts. In contrast, the initial state $x_0 = (90, 10, 0)$ represents a population predominantly susceptible with a small number of initial infections.

The dynamics of the system are governed by two fundamental transition rules $\mathcal{T} = \{\tau_{\text{infection}}, \tau_{\text{recovery}}\}$:

1. The infection process $\tau_{\text{infection}} = (\text{infection}, \mathbf{u}_1, \mathbf{g}_1, \lambda_1)$:
   - Consumes one susceptible and one infected: $\mathbf{u}_1 = (1, 1, 0)$
   - Produces two infected: $\mathbf{g}_1 = (0, 2, 0)$
   - Occurs at rate $\lambda_1(\mathbf{X}) = k_i \cdot X_S \cdot \frac{X_I}{X_S + X_I + X_R}$
   - Where $k_i = 0.05$ is the infection rate constant
2. The recovery process $\tau_{\text{recovery}} = (\text{recovery}, \mathbf{u}_2, \mathbf{g}_2, \lambda_2)$:
   - Consumes one infected: $\mathbf{u}_2 = (0, 1, 0)$
   - Produces one recovered: $\mathbf{g}_2 = (0, 0, 1)$
   - Occurs at rate $\lambda_2(\mathbf{X}) = k_r \cdot X_I$
   - Where $k_r = 0.05$ is the recovery rate constant

These transitions can be expressed in chemical reaction notation as:

$$S + I \xrightarrow{k_i \cdot X_S \cdot \frac{X_I}{X_S + X_I + X_R}} 2I$$
$$I \xrightarrow{k_r \cdot X_I} R$$

The infection rate function incorporates the law of mass action through the term $X_S \cdot \frac{X_I}{X_S + X_I + X_R}$, representing the probability of contact between susceptible and infected individuals. The recovery process follows a more straightforward linear rate, assuming a constant likelihood of recovery for each infected individual.

This formalisation provides a rigorous mathematical foundation for studying disease spread dynamics, enabling analytical analysis and computational simulation of epidemic processes. The model's simplicity belies its power in capturing fundamental epidemiological phenomena, including the epidemic threshold, peak infection time, and final size of the epidemic.

6.2.2 The Repressilator: A Synthetic Genetic Oscillator

The Repressilator, first engineered by Elowitz and Leibler in 2000, represents a milestone in synthetic biology as one of the first synthetic genetic circuits to demonstrate periodic oscillations. This system consists of three genes arranged in a cyclic negative feedback loop, where each gene represses the transcription of its successor. We formalise this synthetic circuit within the Population Modelling Framework as follows.

Let $\mathcal{M}_{\text{Rep}} = (\mathbf{S}, \mathbf{X}, \mathcal{D}, x_0, \mathcal{T})$ represent our Repressilator model. The model incorporates both transcription and translation processes, with proteins acting as transcriptional repressors:

- The species set $\mathbf{S} = \{X, Y, Z, \text{PX}, \text{PY}, \text{PZ}, \text{VOID}\}$ represents:
  - X, Y, Z: mRNA transcripts for each gene (LacI, TetR, and CI respectively)
  - PX, PY, PZ: corresponding proteins
  - VOID: auxiliary species for reaction modelling
- The population state $\mathbf{X} = (X_X, X_Y, X_Z, X_{\text{PX}}, X_{\text{PY}}, X_{\text{PZ}}, X_{\text{VOID}})$ tracks:
  - mRNA molecule counts (X_X, X_Y, X_Z)
  - Protein molecule counts ($X_{\text{PX}}, X_{\text{PY}}, X_{\text{PZ}}$)
  - Auxiliary species count (X_{VOID})

The state space $\mathcal{D} = \mathbb{N}^7$ reflects the discrete nature of molecular counts, with initial state $x_0 = (0, 20, 0, 0, 0, 0, 1)$ representing starting conditions with only Y mRNA present.

The system dynamics are governed by four categories of transition rules $\mathcal{T}$:

1. Transcription processes $\{\tau_{\text{tr_X}}, \tau_{\text{tr_Y}}, \tau_{\text{tr_Z}}\}$:

$$\text{VOID} \xrightarrow{\lambda_{\text{tr}}(i)} \text{VOID} + i \qquad \text{for } i \in \{X, Y, Z\}$$

$$\lambda_{\text{tr}}(i) = a_{0_\text{tr}} + \frac{a_{\text{tr}} \cdot K_M^n}{K_M^n + {X_{\text{P}_r}}^n} \qquad \text{where P}_r \text{ is repressor protein}$$

2. Translation processes $\{\tau_{\text{tl_X}}, \tau_{\text{tl_Y}}, \tau_{\text{tl_Z}}\}$:

$$\text{VOID} \xrightarrow{k_{\text{tl}} \cdot X_i} \text{VOID} + \text{P}i \qquad \text{for } i \in \{X, Y, Z\}$$

3. mRNA degradation $\{\tau_{\text{deg_mX}}, \tau_{\text{deg_mY}}, \tau_{\text{deg_mZ}}\}$:

$$i + \text{VOID} \xrightarrow{k_{\text{d_mRNA}} \cdot X_i} \text{VOID} \qquad \text{for } i \in \{X, Y, Z\}$$

4. Protein degradation $\{\tau_{\text{deg_pX}}, \tau_{\text{deg_pY}}, \tau_{\text{deg_pZ}}\}$:

$$\text{P}i + \text{VOID} \xrightarrow{k_{\text{d_prot}} \cdot X_{\text{P}i}} \text{VOID} \qquad \text{for } i \in \{X, Y, Z\}$$

The rate constants and parameters are biologically motivated:

- $n = 2$: Hill coefficient representing cooperative binding

- $K_M = 40$: Michaelis constant for repression
- $\tau_{\text{mRNA}} = 2$: mRNA half-life
- $\tau_{\text{prot}} = 10$: protein half-life
- $ps_a = 0.5$: promoter strength (repressed)
- $ps_0 = 0.0005$: promoter strength (unrepressed)

Derived constants include:

$$
\begin{aligned}
k_{\text{d_mRNA}} &= \ln(2)/\tau_{\text{prot}} && \text{(mRNA decay rate)} \\
k_{\text{d_prot}} &= \ln(2)/\tau_{\text{mRNA}} && \text{(protein decay rate)} \\
k_{\text{tl}} &= \text{eff}/t_{\text{ave}} && \text{(translation rate)} \\
a_{\text{tr}} &= (ps_a - ps_0) \cdot 60 && \text{(transcription rate)} \\
a_{0_\text{tr}} &= ps_0 \cdot 60 && \text{(basal transcription rate)}
\end{aligned}
$$

This formalisation captures the essential features of the Repressilator: the cyclic repression pattern, the separation of transcription and translation processes, and the nonlinear repression kinetics through Hill functions. The model's oscillatory behaviour emerges from the interplay between protein-mediated repression and the natural degradation of mRNA and proteins, creating a delayed negative feedback loop characteristic of biological oscillators.

Including both transcriptional and translational processes and realistic degradation kinetics makes this model particularly valuable for understanding the design principles of synthetic genetic circuits and the requirements for sustained oscillations in biological systems.

Specifications: in Appendix D Section D.2, Listing D.3 provides the complete specification for the Repressilator model parameter synthesis. The specification employs Latin Hypercube Sampling with 300 samples and includes detailed configurations for transcription, translation, and protein degradation processes. The model parameters and reaction rules are carefully chosen to capture the complex dynamics of the synthetic genetic oscillator, with multiple simulation replicas ensuring robust evaluation of the system's behaviour.

6.2.3 Three-State Protocol with Malicious Agents

The Three-State Protocol (TSP) represents a fundamental model for studying consensus formation in distributed systems under adversarial conditions. We extend the classical three-state model by incorporating malicious agents that can disrupt the consensus process. This extension provides insights into system resilience against Byzantine behaviour. We formalise this model within the Population Modelling Framework as follows.

Let $\mathcal{M}_{\text{TSP}} = (\mathbf{S}, \mathbf{X}, \mathcal{D}, x_0, \mathcal{T})$ represent our TSP consensus model with malicious agents:

- The species set $\mathbf{S} = \{\text{RED}, \text{BLUE}, \text{UNC}, \text{MAL}\}$ represents:

  - RED, BLUE: Agents committed to opposing consensus values
  - UNC: Uncommitted agents that can be persuaded
  - MAL: Malicious agents attempting to disrupt consensus

- The population state $\mathbf{X} = (X_{\text{RED}}, X_{\text{BLUE}}, X_{\text{UNC}}, X_{\text{MAL}})$ tracks the number of agents in each state

The state space $\mathcal{D} = \mathbb{N}^4$ reflects the discrete nature of agent counts, with initial state $x_0 = (10, 10, 75, 5)$ representing a population predominantly uncommitted with equal opposing factions and a minority of malicious agents.

The system dynamics are governed by five transition rules $\mathcal{T} = \{\tau_{\text{UB}}, \tau_{\text{UR}}, \tau_{\text{UM}}, \tau_{\text{BU}}, \tau_{\text{RU}}\}$, where the presence of malicious agents conditionally modulates each transition:

1. Uncommitted to Committed transitions $\{\tau_{\text{UB}}, \tau_{\text{UR}}\}$:

$$\text{UNC} \xrightarrow{\lambda_{\text{UC}}(c)} c \qquad \text{for } c \in \{\text{BLUE}, \text{RED}\}$$
$$\lambda_{\text{UC}}(c) = \mathbb{1}_{M<0.5} \cdot \rho \cdot X_{\text{UNC}} \cdot \frac{X_c}{X_{\text{total}}}$$

   Where $\rho = 1.0$ is the persuasion rate and $\mathbb{1}_{M<0.5}$ is an indicator function that equals 1 when the fraction of malicious agents is less than 0.5.

2. Malicious recruitment τ_{UM}:

$$\text{UNC} \xrightarrow{\lambda_{\text{UM}}} \text{MAL}$$
$$\lambda_{\text{UM}} = \rho \cdot X_{\text{UNC}} \cdot \frac{X_{\text{MAL}}}{X_{\text{total}}}$$

3. Committed to Uncommitted transitions $\{\tau_{\text{BU}}, \tau_{\text{RU}}\}$:

$$c \xrightarrow{\lambda_{\text{CU}}(c,o)} \text{UNC} \qquad \text{for } (c, o) \in \{(\text{BLUE}, \text{RED}), (\text{RED}, \text{BLUE})\}$$
$$\lambda_{\text{CU}}(c, o) = \mathbb{1}_{M<0.5} \cdot \rho \cdot X_c \cdot \frac{X_o}{X_{\text{total}}}$$

The model includes four key measures that track the system state:

$$R = \frac{X_{\text{RED}}}{X_{\text{total}}} \qquad \text{(Red faction proportion)}$$
$$B = \frac{X_{\text{BLUE}}}{X_{\text{total}}} \qquad \text{(Blue faction proportion)}$$
$$U = \frac{X_{\text{UNC}}}{X_{\text{total}}} \qquad \text{(Uncommitted proportion)}$$
$$M = \frac{X_{\text{MAL}}}{X_{\text{total}}} \qquad \text{(Malicious proportion)}$$

This formalisation captures several crucial aspects of consensus formation under adversarial conditions:

- **Threshold behaviour:** The system exhibits a critical threshold at 50% malicious agents, beyond which normal consensus formation is completely suppressed.
- **Competition Dynamics:** The model incorporates direct competition between committed agents and the recruitment of uncommitted agents.
- **Malicious Influence:** Malicious agents can recruit uncommitted agents and indirectly affect the transition rates between other states.
- **Population Conservation:** The total population remains constant, with agents transitioning between states according to local interaction rules.

This model provides a foundation for analysing consensus protocols' resilience against Byzantine behaviour, offering insights into the critical thresholds of malicious agent proportions and their effects on consensus formation dynamics. Incorporating uncommitted agents adds layer of complexity, making the model particularly relevant for studying opinion dynamics in social networks and distributed systems where uncertainty and malicious behaviour coexist.

6.2.4 Spatial Three-State Protocol with Ring Topology

We present a spatial extension of the TSP where agents are distributed across a ring of compartments, introducing local interaction dynamics and spatial heterogeneity into the consensus formation process. This spatial variant enables the study of consensus emergence under local constraints and regional variations in agent distributions.

Let $\mathcal{M}_{\text{STSP}} = (\mathbf{S}, \mathbf{X}, \mathcal{D}, x_0, \mathcal{T})$ represent our spatial TSP model:

- The species set $\mathbf{S} = \{\text{RED}[i], \text{BLUE}[i], \text{UNC}[i], \text{MAL}[i] \mid i \in [0, K-1]\}$ where:
  - $K = 5$ is the number of compartments in the ring
  - Each species is replicated across all compartments
  - $[i]$ denotes the compartment index
- The population state $\mathbf{X} = (X_{\text{RED},i}, X_{\text{BLUE},i}, X_{\text{UNC},i}, X_{\text{MAL},i})_{i=0}^{K-1}$ tracks:
  - Local population counts for each species in each compartment
  - Total population is conserved within each compartment

The state space $\mathcal{D} = (\mathbb{N}^4)^K$ reflects the compartmentalised nature of the system. The initial state x_0 is defined by:

$$
\begin{aligned}
X_{\text{RED},i}(0) &= r_i \cdot N_{\text{RB}}/K && \text{where } r_i \text{ is red fraction in compartment } i \\
X_{\text{BLUE},i}(0) &= (1 - r_i) \cdot N_{\text{RB}}/K && \text{for } i \in [0, K-1] \\
X_{\text{UNC},i}(0) &= (1 - m) \cdot N_{\text{UM}}/K && \text{for all compartments} \\
X_{\text{MAL},i}(0) &= m \cdot N_{\text{UM}}/K && \text{for all compartments}
\end{aligned}
$$

where:

- $N_{\text{RB}} = 20$: Total committed (Red and Blue) agents
- $N_{\text{UM}} = 80$: Total uncommitted and malicious agents

- $r_i = 0.5$: Initial red fraction in each compartment
- $m = 0.0$: Initial malicious fraction

The system dynamics are governed by five types of transition rules $\mathcal{T}$, each operating within the spatial context:

1. Local uncommitted to committed transitions $\{\tau_{\text{UB},i}, \tau_{\text{UR},i}\}_{i=0}^{K-1}$:

$$\text{UNC}[i] \xrightarrow{\lambda_{\text{UC}}(c,i)} c[i] \qquad \text{for } c \in \{\text{BLUE}, \text{RED}\}$$

$$\lambda_{\text{UC}}(c,i) = \mathbb{1}_{M_i<0.5} \cdot \rho \cdot X_{\text{UNC},i} \cdot \frac{\sum_{j\in\mathcal{N}(i)} X_{c,j}}{3 \cdot X_{\text{total},i}}$$

   where:

   - $\rho = 3.0$ is the persuasion rate
   - $\mathcal{N}(i) = \{i, (i+1) \mod K, (i-1+K) \mod K\}$ is the neighborhood of compartment i
   - M_i is the fraction of malicious agents in compartment i

2. Local malicious recruitment $\{\tau_{\text{UM},i}\}_{i=0}^{K-1}$:

$$\text{UNC}[i] \xrightarrow{\lambda_{\text{UM}}(i)} \text{MAL}[i]$$

$$\lambda_{\text{UM}}(i) = \rho \cdot X_{\text{UNC},i} \cdot \frac{X_{\text{MAL},i}}{X_{\text{total},i}}$$

3. Local committed to uncommitted transitions $\{\tau_{\text{BU},i}, \tau_{\text{RU},i}\}_{i=0}^{K-1}$:

$$c[i] \xrightarrow{\lambda_{\text{CU}}(c,o,i)} \text{UNC}[i] \qquad \text{for } (c,o) \in \{(\text{BLUE}, \text{RED}), (\text{RED}, \text{BLUE})\}$$

$$\lambda_{\text{CU}}(c,o,i) = \mathbb{1}_{M_i<0.5} \cdot \rho \cdot X_{c,i} \cdot \frac{\sum_{j\in\mathcal{N}(i)} X_{o,j}}{3 \cdot X_{\text{total},i}}$$

The model includes four global measures that aggregate across compartments:

$$R = \sum_{i=0}^{K-1} \frac{X_{\text{RED},i}}{X_{\text{total},i}} \qquad \text{(Total Red proportion)}$$

$$B = \sum_{i=0}^{K-1} \frac{X_{\text{BLUE},i}}{X_{\text{total},i}} \qquad \text{(Total Blue proportion)}$$

$$U = \sum_{i=0}^{K-1} \frac{X_{\text{UNC},i}}{X_{\text{total},i}} \qquad \text{(Total Uncommitted proportion)}$$

$$M = \sum_{i=0}^{K-1} \frac{X_{\text{MAL},i}}{X_{\text{total},i}} \qquad \text{(Total Malicious proportion)}$$

This spatial formalisation introduces several key features:

- **Local Interactions:** Agents are influenced by their local and adjacent compartments, creating neighborhood effects.
- **Spatial Heterogeneity:** The ring topology allows for studying spatial patterns and wave-like propagation of opinions.
- **Compartment-Specific Thresholds:** Malicious agent effects are evaluated independently in each compartment.
- **Conservation Laws:** Total population is conserved globally and within each compartment.

This model provides a framework for studying how spatial structure affects consensus formation, particularly:

- The role of local versus global interactions in opinion dynamics
- The emergence and propagation of regional consensus patterns
- The impact of spatially distributed malicious agents on consensus formation
- The relationship between local thresholds and global system behaviour

6.3 Optimal Feasibility

6.3.1 SIR Model Case Study

Problem Statement: This case study demonstrates the application of parameter synthesis to model and temporal logic formula parameters in an SIR (Susceptible-Infected-Recovered) model. We consider two optimisation scenarios within this framework.

In the first scenario, we aim to find optimal values for the infection rate (k_i^*) and recovery rate (k_r^*),with an initial population of 90 susceptible, 10 infected, and zero recovered individuals, that maximise the probability of satisfying a specific temporal pattern of infection. The property we consider is expressed by the following STL formula:

$$\varphi_{\text{SIR_1}} = \Diamond_{[100,120]} \left(\mathrm{I} = 0\right) \wedge \Box_{[0,100]} \left(\mathrm{I} > 0\right) \tag{6.1}$$

This formula states that within a specific time frame (100 to 120), the population remains entirely free from infectious individuals, while during the earlier period (0 to 100), infectious individuals continuously exist.

So the problem can be formalised as in Equation 6.2

$$(k_i^\star, k_r^\star) = \underset{\substack{(k_i,k_r)\in \\ [0.005,0.3]\times[0.005,0.2]}}{\arg\max} \; P\left(\varphi_{\text{SIR_1}} \mid \mathcal{M}_{(k_i,k_r)}\right) \tag{6.2}$$

In the second scenario, fixed the infection rate at $k_i = 0.2$ and the recovery rate at $k_r = 0.05$ and starting by the same initial conditions, our objective is to identify a time interval $[a, a + b]$ during which the proportion of infected individuals remains consistently above 30%. This is expressed using the following STL formula:

$$\varphi_{\text{SIR_2}}(a, b) = \Box_{[a,a+b]}(\mathbf{I}_{(\%)} > 0.3) \tag{6.3}$$

where $a \in [0, 50]$ and $b \in [0, 10]$ are the parameters to be synthesised. The optimisation task aims to maximise the quantitative robustness of this formula.

We can formulate this optimisation problem as shown in Equation 6.4:

$$(a^\star, b^\star) = \underset{\substack{(a,b)\in \\ [0,50]\times[0.001,50]}}{\arg\max} \; R\left(\varphi_{\text{SIR_2}}(a, b) \mid \mathcal{M}\right) \tag{6.4}$$

Results: These are mean values that summarise the tendencies for each scenario:

1. **Scenario 1:**
   - Mean values: $k_i^\star \approx 0.2271$, $k_r^\star \approx 0.0582$, surrogate evaluation ≈ 0.3120, real evaluation ≈ 0.3265, computation time ≈ 8114.4 ms.
   - These parameters suggest that with a probability of approximately 32.65%, the population will experience a continuous presence of infectious individuals for the first 100 time units, followed by a complete absence of infectious individuals between 100 and 120 time units.
2. **Scenario 2:**
   - Mean values: $a^\star \approx 17.0373$, $b^\star \approx 1.9685$, surrogate evaluation ≈ 0.1791, real evaluation ≈ 0.1828, computation time ≈ 11210.7 ms.
   - These parameters indicate that with a probability of approximately 18.28%, more than 30% of the population remains consistently infected in the time intervals a and b.

Figure 6.2 illustrates the results of our parameter synthesis for the SIR model, showcasing both the optimal parameter values and the system's behaviour under these conditions. The dataset for both scenarios is composed of 1000 samples, and 95% is used for training. The surrogate model demonstrates strong learning capabilities in both scenarios. Due to the inherent simplicity of the SIR model, which tends to stabilise relatively quickly, the surrogate performs synthesis very efficiently. The tool offers versatility by accommodating a diverse range of surrogate models and optimisation algorithms, enabling a comprehensive exploration of various combinations for complex optimisation problems. In those cases of study, multiple combinations were tested. Figure 6.1 shows that the results of both case studies reveal a robust performance across all combinations tested. Although minor variations in performance and computation time were observed in both studies, the overall similarity in outcomes underscores the tool's ability to converge on optimal solutions across different methodological approaches, including Simulated Annealing (SA), Particle Swarm Optimizaiton (PSO) and variants of Mesh Adaptive Direct Search (MADS) paired with surrogate models such as Radial Basis Function (RBF) networks, Random Forests (RF), and Gradient Tree Boosting (GTB).

Specifications: in Appendix D Section D.1, Listings D.1 and D.2, we present the complete specifications for the first and second case studies, respectively. Both specifications utilise Latin Hypercube Sampling with 100 samples for data generation. The placeholder values ID_SURROGATE and ID_OPTIMIZATION represent the various

combinations of surrogate models and optimisation algorithms tested in our experiments, as detailed in Figure 6.1. The first case study focuses on parameter synthesis for infection and recovery rates, while the second investigates temporal parameters with fixed kinetic rates.

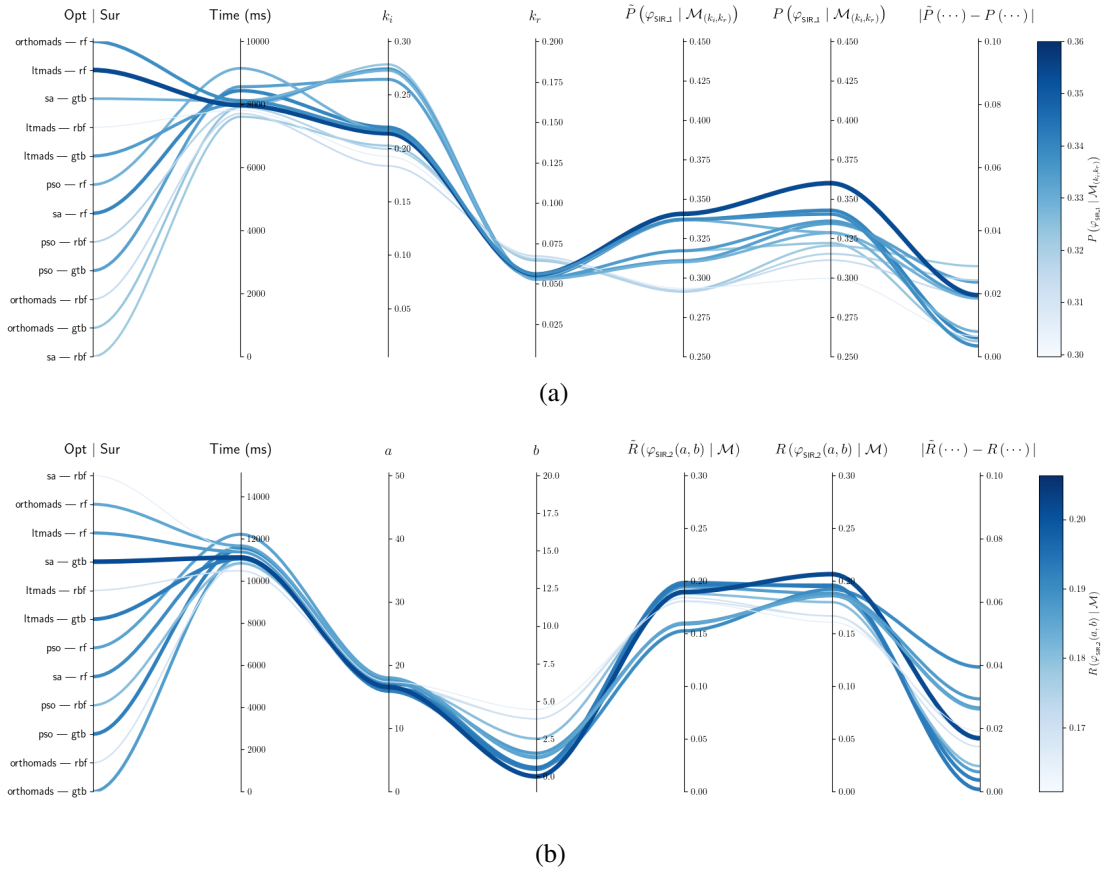


Figure 6.1: *Parallel coordinates plot of optimisation results for two different scenarios. (a) First case study with parameters k_i and k_r. (b) The second case study has parameters a and b. Both plots show the performance of various optimisation algorithms paired with different surrogate models, illustrating the consistency of results across different approaches. The color intensity and line thickness represent the real evaluation scores, with darker and thicker lines indicating better performance.*

6.4 Considerations on Training Set Size impact on performance

The SIR model case studies offer fascinating insights into the relationship between training set size, surrogate accuracy, and optimisation effectiveness. Figure 6.3 shows performance comparisons across different combinations of optimisation algorithms and surrogate models, using 100 samples training sets.

While larger training sets produce more accurate surrogates (as evidenced by the lower error metrics in the rightmost columns), smaller training sets can still effectively guide optimisation toward optimal solutions. This phenomenon occurs because the surrogate, even with reduced accuracy, successfully captures the general topology of the

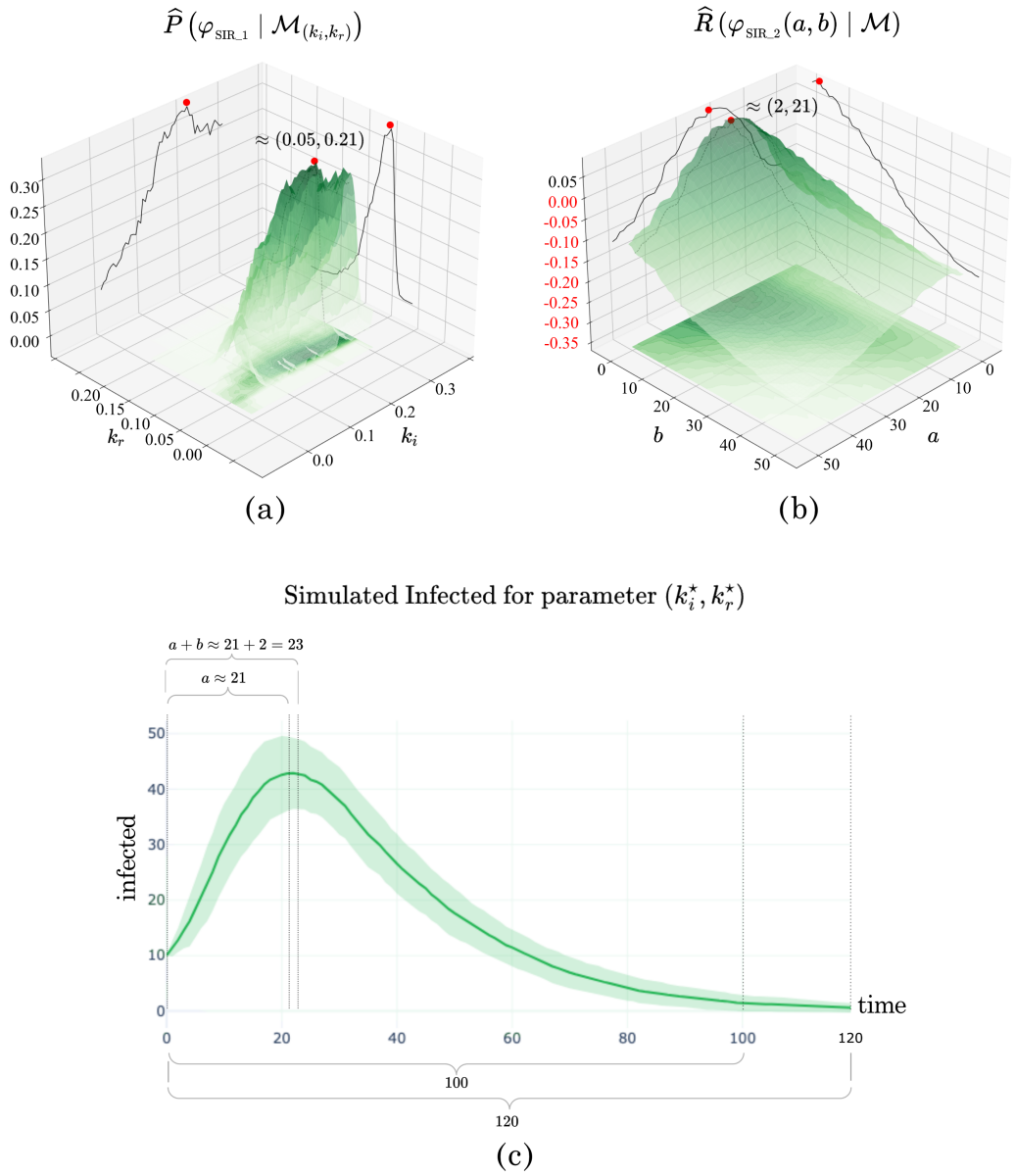
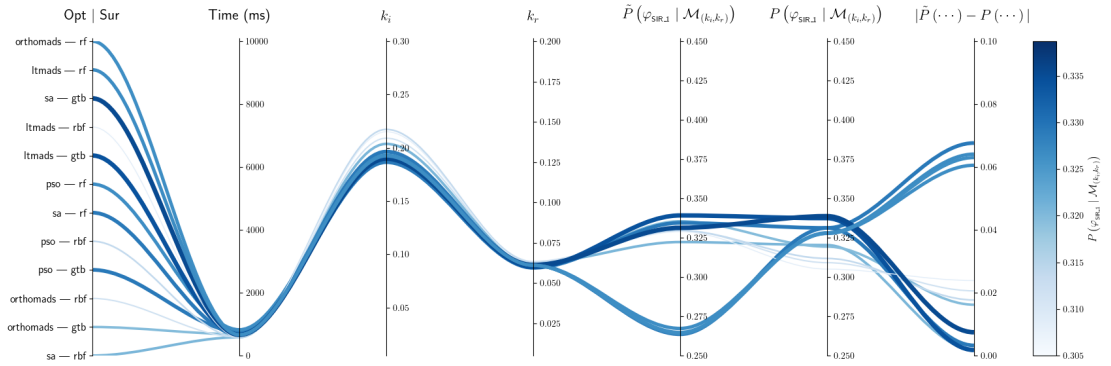
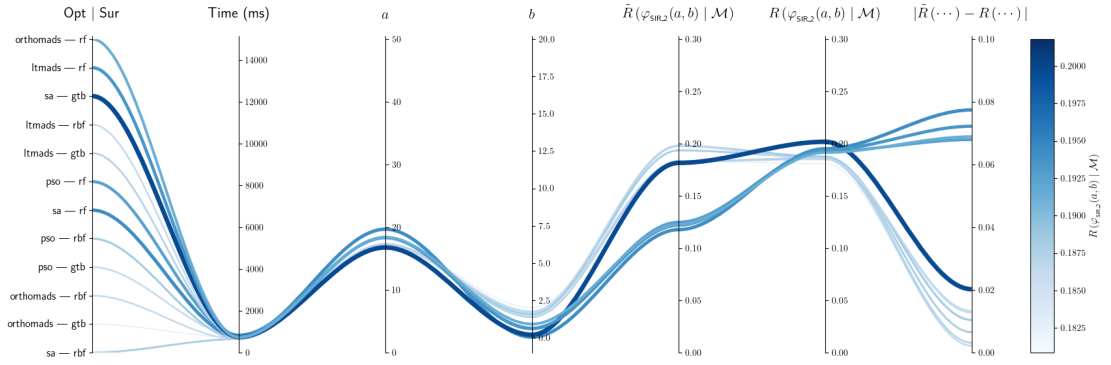


Figure 6.2: *An instance of parameter synthesis demonstrating optimal feasibility for the SIR model under two scenarios: (a) A 3D plot showing the probability of satisfying the formula φ_{SIR_1} as k_i and k_r vary, with optimal values ($k_i \approx 0.05$, $k_r \approx 0.21$) emphasised. (b) Another 3D plot illustrating the robustness $R(\varphi_{\text{SIR}_2}(a,b) \mid \mathcal{M})$ as a and b change, with the optimal values ($a \approx 21$, $b \approx 2$) highlighted. (c) A time simulation of the infected population using the optimal k_i and k_r from the first scenario.*



(a)



(b)

Figure 6.3: *Parallel coordinates plot of optimisation results for the same scenario in Figure6.1 but using a trainingset composed by 100 sample*

objective function, particularly its critical points. For the SIR model scenarios, optimisation runs with 100-sample training sets achieved approximately the exact solutions as those found using 1000 samples, while reducing computation time from approximately 8000ms to 750ms per run.

6.4.1 Repressillator case of study

Problem statement: This study investigates the behaviour of the repressilator model, a synthetic genetic regulatory network designed to produce oscillations in gene expression. The repressilator consists of three genes, each encoding a protein that cyclically represses the next gene, creating a negative feedback loop. We employ an optimal feasibility methodology to analyse and manipulate the repressilator's behaviour, particularly focusing on the temporal dynamics of the protein PZ.

We are considering the following property :

$$\begin{aligned}\varphi_{\text{Repr}}\left(k_{\max},k_{\min},t,t_w,t_\xi\right) \;=\; & \Diamond_{[0\,,\,t\,]}\big(\,\Box_{\left[t_\xi\,,\,t_w+t_\xi\right]}\left(\text{PZ} > k_{\min} \wedge \text{PZ} < k_{\max}\right) \\ & \wedge\; \Diamond_{\left[0,\,t_\xi\right]}\left(\text{PZ} < k_{\min}\right) \\ & \wedge\; \Diamond_{\left[t_w+t_\xi\,,\,t_w+2\cdot t_\xi\right]}\left(\text{PZ} < k_{\min}\right)\,\big)\end{aligned} \tag{6.5}$$

Then, we can formulate this optimisation problem as shown in Equation 6.6:

$$\left(\tau^\star_{\text{mRNA}}, \tau^\star_{\text{prot}}\right) \;=\; \underset{\substack{(\tau_{\text{mRNA}},\tau_{\text{prot}})\in \\ [0,10]\times[0,10]}}{\arg\max}\; P\left(\varphi_{\text{Repr}}\left(800,500,100,10,20\right) \mid \mathcal{M}_{(\tau_{\text{mRNA}},\tau_{\text{prot}})}\right) \tag{6.6}$$

The parameters subject to change in this study are the half-life of mRNA (τ_{mRNA}) and the half-life of proteins (τ_{prot}). These parameters significantly influence the dynamics of gene expression and protein oscillations. Understanding and controlling these oscillations can have several biological implications like synchronising cellular processes, for instance, periodic fluctuations in repressor proteins can coordinate the timing of gene expression, ensuring that different cellular events occur in a harmonised manner or enhancing the robustness of gene regulatory networks, by maintaining protein levels within specific bounds, the cell can avoid detrimental effects of overexpression or underexpression of specific proteins.

Results: The optimisation process, conducted over several experiments, yielded the following mean optimal parameters for the repressilator model: mRNA half-life ($\tau^\star_{\text{mRNA}}$): 5.91 ± 1.04 (mean $\pm$ standard deviation) and Protein half-life ($\tau_{\text{prot}\star}$): 3.20 ± 0.80, with a satisfaction probability of $\approx 34\%$. These optimal parameters maximise the probability of satisfying the STL formula (Equation 6.5) to approximately 0.34. The results of an instance of our optimisation of the repressilator model are visualised in Figure 6.4, which demonstrates the oscillatory behaviour achieved with the optimal parameters. To perform the synthesis, the algorithm used was PSO, with a Random Forest as the surrogate model. The dataset consisted of 300 samples, with 95% used for training, and the process took an average of 3 minutes and 30 seconds

to complete. The repressilator model showcases our surrogate's ability to handle more complex systems. Unlike the SIR model, the repressilator exhibits ongoing oscillatory behaviour, presenting a more significant challenge for analysis. Our surrogate model tackles this challenge effectively. Despite the complexity of the system and the limited sample size, it captures the essential dynamics of the repressilator.

Specifications : in Listings D.1 and D.2, we present the complete specifications for the first and second case studies, respectively. Both specifications utilise Latin Hypercube Sampling with 100 samples for data generation. The placeholder values ID_SURROGATE and ID_OPTIMIZATION represent the various combinations of surrogate models and optimisation algorithms tested in our experiments, as detailed in Figure 6.1. The first case study focuses on parameter synthesis for infection and recovery rates, while the second investigates temporal parameters with fixed kinetic rates.

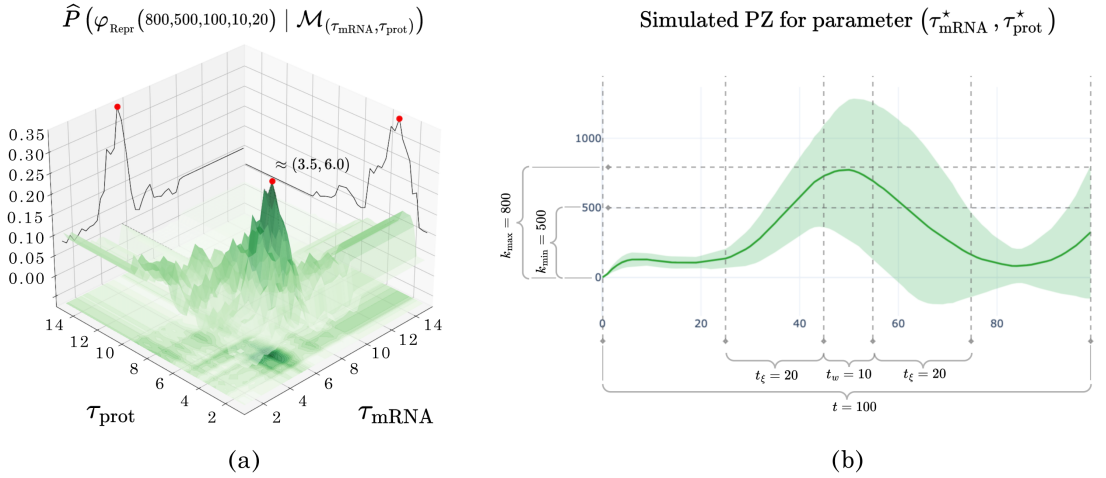


Figure 6.4: *Optimisation results for the repressilator model. (a) 3D plot of the probability $P(\varphi_{Repr}(800, 500, 100, 10, 20) \mid \mathcal{M}(\tau_{mRNA}, \tau_{prot}))$ that the STL formula is satisfied as τ_{mRNA} and τ_{prot} vary. The optimal values found ($\tau^{\star}_{mRNA} \approx 6.0$, $\tau^{\star}_{prot} \approx 3.5$) are highlighted, yielding a maximum satisfaction probability of approximately 0.34. (b) Simulated protein PZ concentration over time using the optimal $\tau^{\star}_{mRNA}$ and $\tau^{\star}_{prot}$ values. The plot demonstrates the oscillatory behaviour of the repressilator, with annotations showing the key time intervals ($t_\xi = 20$, $t_w = 10$, $t_\epsilon = 20$) and protein concentration thresholds ($k_{min} = 500$, $k_{max} = 800$) as specified in the STL formula.*

6.5 Variant model adjustment cases

6.5.1 Three-State Protocol (TSP) consensus variants

Having demonstrated the capabilities of our optimal feasibility technique, we now turn our attention to the core contribution of this work: the Variant Model Adjustment method. To illustrate the utility of this method, we present a comprehensive case study focused on consensus protocols in distributed systems. This example demonstrates how our technique can be applied to preserve, improve, or intentionally diverge the behaviour of model variants while adhering to specified properties.

Consensus case study

This case study examines a TSP consensus mechanism, comparing an original model to a spatially-structured variant. The TSP model represents a consensus process between entities in three main states: Red, Blue, and Uncertain. Additionally, entities can transition to a Malicious (Mal) state, potentially disrupting the consensus process.

In the original model, entities move freely within a shared space, representing a fully connected network. A significant extension of this classic protocol is the introduction of malicious entities. These Malicious entities play a critical role in system stability and consensus formation. If the percentage of Malicious entities exceeds 50%, the entire system halts, preventing further state transitions. The variant model introduces a spatial structure with five interconnected nodes. A key difference is that in this model, if Malicious entities exceed 50% in a single node, only that node's operations are halted, rather than the entire system. This allows for more nuanced dynamics and potential recovery scenarios.

The optimisation process focuses on adjusting the parameter vector Θ, which consists of two key sets of parameters:

- The initial percentage of reds $\text{RED}_{\%}[i]$ to the sum of reds and blues in the node i, so that :

$$\text{RED}[i] = \text{RED}_{\%}[i] \cdot \frac{\sum_{i=0}^{K-1} (\text{RED}[i] + \text{BLUE}[i])}{K}$$

  where $\text{RED}[i]$ is the quantity of RED in each i and K is the number of nodes. Consequently, the numbers of $\text{BLUE}_{\%}[i]$ will be :

$$\text{BLUE}[i] = (1.0 - \text{RED}_{\%}[i]) \cdot \frac{\sum_{i=0}^{K-1} (\text{RED}[i] + \text{BLUE}[i])}{K}$$

- The initial percentage of malicious to the sum of malicious and uncertian

$$\text{MAL}[i] = \text{MAL}_{\%} \cdot \frac{\sum_{i=0}^{K-1} (\text{MAL}[i] + \text{UNC}[i])}{K}$$

  where $\text{MAL}[i]$ is the quantity of MAL in each i. Consequently, the numbers of $\text{UNC}_{\%}[i]$ will be:

$$\text{UNC}[i] = (1.0 - \text{MAL}_{\%}) \cdot \frac{\sum_{i=0}^{K-1} (\text{MAL}[i] + \text{UNC}[i])}{K}$$

Thus, our complete parameter vector is:

$$\Theta = (\text{RED}_{\%}[0], \text{RED}_{\%}[1], \text{RED}_{\%}[2], \text{RED}_{\%}[3], \text{RED}_{\%}[4], \text{MAL}_{\%})$$

The optimisation aims to find the optimal Θ within the defined search space $\mathcal{H}$, where:

$$\mathcal{H} = \{(\text{RED}_{\%}[0], \text{RED}_{\%}[1], \text{RED}_{\%}[2], \text{RED}_{\%}[3], \text{RED}_{\%}[4], \text{MAL}_{\%}) \in \mathbb{R}^6 \mid$$

$$0.1 \leqslant \text{RED}_{\%}[i] \leqslant 1.0 \text{ for } i \in \{0, 1, 2, 3, 4\}, \text{ and } 0.0 \leqslant \text{MAL}_{\%} \leqslant 1.0\}$$

Subject to the following constraint:

$$1.5 \leqslant \sum_{i=0}^{4} \text{RED}_{\%}[i] \leqslant 3.5$$

The key properties being evaluated in the comparison are:

$$\varphi_1(t) = \Diamond_{[0,t]} \left(\text{Red}_{(\%)} \geqslant 1.0\right) \tag{6.7}$$

$$\varphi_2(t) = \left(\text{Red}_{(\%)} > 3 \cdot \text{Blue}_{(\%)}\right) \wedge \Diamond_{[0,t]} \left(\text{Blue}_{(\%)} \leqslant 0.0\right) \tag{6.8}$$

$$\varphi_3(t) = \Box_{[0,t]} \left(\text{Red}_{(\%)} \geqslant 0.5\right) \tag{6.9}$$

$$\varphi_4(t) = \Diamond_{\left[0,\frac{t}{4}\right]} \left(\text{Mal}_{(\%)} \geqslant 0.05\right) \leftrightarrow \Box_{\left[\frac{3t}{4},t\right]} \left(\text{Mal}_{(\%)} \geqslant 0.25\right) \tag{6.10}$$

These formulas evaluate various aspects of entity populations over time. Formula 6.7 assesses whether RED entities achieve complete dominance. Formula 6.8 examines a scenario where RED entities significantly outnumber BLUE entities, eventually leading to BLUE's disappearance. Formula 6.9 ensures a consistent majority presence of RED entities throughout the observed period. Formula 6.10 explores the relationship between the early appearance and later proliferation of malicious entities.

Results Analysis

Given the original model $\mathcal{M}$, the variant model $\widetilde{\mathcal{M}}_\Theta$ with tunable parameters Θ and the set of formulae $\Phi = [\varphi_1, \varphi_2, \varphi_3, \varphi_4]$ represents our properties of interest. We observed that:

$$\vec{R}(\Phi \mid \mathcal{M}) \approx (-0.20, -0.42, -0.67, 0.15)$$

This vector is our baseline for comparing the variant model's performance across different scenarios.

Preservation The results show relatively small values for $\Delta^{\approx}_{\Phi}(\mathcal{M}, \widetilde{\mathcal{M}}_\Theta)$, indicating successful preservation of the original model's characteristics.

- $\Delta^{\approx}_{\Phi}(\mathcal{M}, \widetilde{\mathcal{M}}_\Theta) \approx 0.08$
- $\vec{R}(\Phi \mid \widetilde{\mathcal{M}}_\Theta) \approx (-0.19, -0.44, -0.65, 0.12)$
- Entity distribution: RED [2,3,1,2,2], MAL $\approx 8\%$

This distribution closely mimics $\mathcal{M}$, with RED entities comprising about 20% and MAL entities around 5-10% of the total population.

Intentional Divergence: The values of $\Delta^{-}_{\Phi}(\mathcal{M}, \widetilde{\mathcal{M}}_\Theta)$ are significantly larger, indicating successful divergence from $\mathcal{M}$'s behaviour.

- $\Delta^{-}_{\Phi}(\mathcal{M}, \widetilde{\mathcal{M}}_\Theta) \approx 0.44$
- $\vec{R}(\Phi \mid \widetilde{\mathcal{M}}_\Theta) \approx (-0.37, -0.49, -0.97, -0.13)$

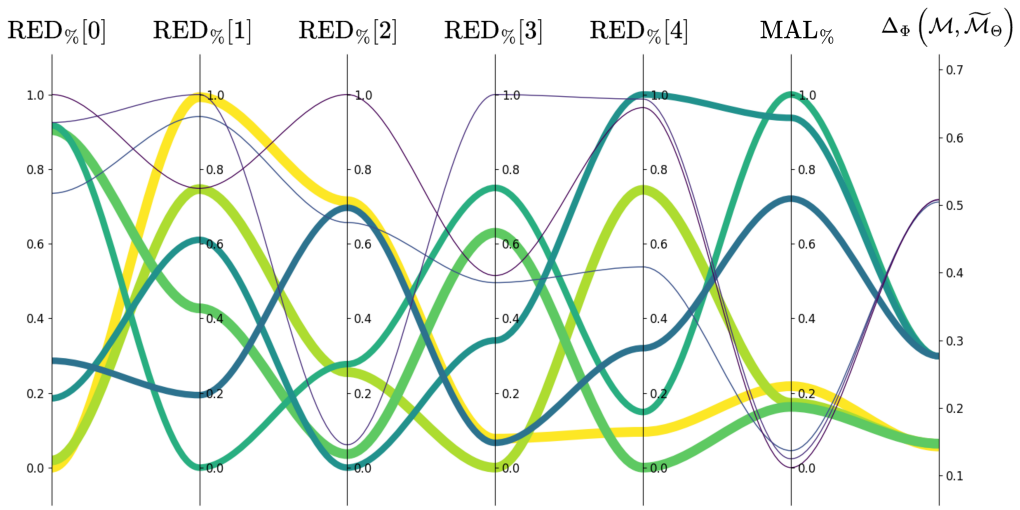


Figure 6.5: *Parallel coordinates plot for the **preservation scenario**. The plot shows consistent ∼5% malicious entities across compartments, varying red entity proportions (20-60%) maintaining a global ∼50% RED-BLUE ratio, and low 'result' values indicating successful preservation of original model characteristics.*

- Entity distribution: RED [1,2,1,1,1], MAL $\approx 95\%$

This distribution shows a dramatic shift towards MAL entities, severely impacting system stability and consensus formation.

Improvement: The improvement results show the largest values for $\Delta^{+}_{\Phi}(\mathcal{M}, \widetilde{\mathcal{M}}_{\Theta})$, suggesting substantial enhancement in satisfying Θ.

- $\Delta^{+}_{\Phi}(\mathcal{M}, \widetilde{\mathcal{M}}_{\Theta}) \approx 0.72$
- $\vec{R}(\Phi \mid \widetilde{\mathcal{M}}_{\Theta}) \approx (0.00, -0.36, 0.00, 0.01)$
- Entity distribution: RED [3,2,3,3,2], MAL $\approx 4\%$

This distribution demonstrates a significant increase in RED entities and a reduction in MAL entities, promoting system stability and consensus.

Figures 6.5, 6.6, and 6.7 present parallel coordinate plots for preservation, divergence, and improvement scenarios, respectively. These plots include representative samples demonstrating how various parameter configurations affect the outcomes.

Synthesis Details

We employed surrogate optimisation techniques for each scenario (preservation, intentional divergence, and improvement) to explore the parameter space efficiently. The details of this process are as follows:

Dataset and Training:

- Total dataset size: 500 samples

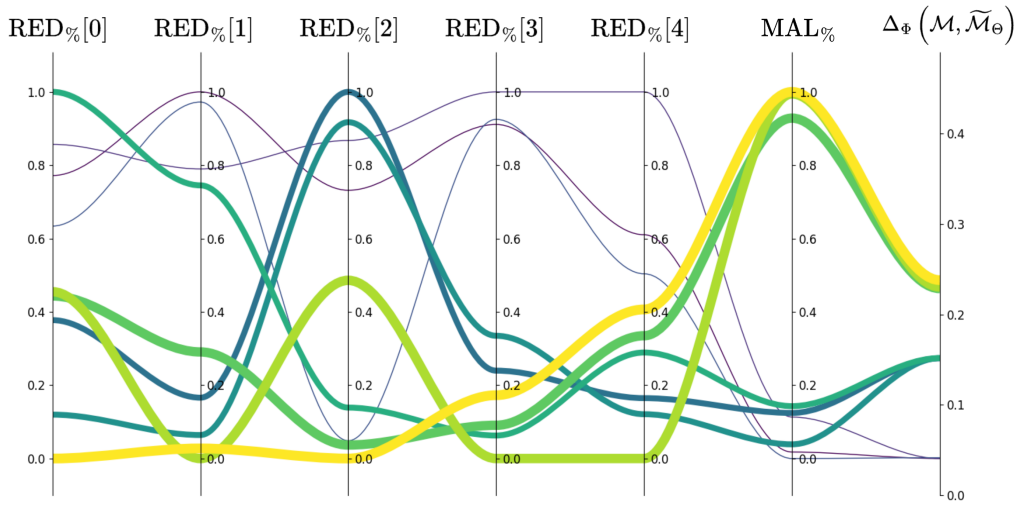


Figure 6.6: *Parallel coordinates plot for the **divergence scenario**. The plot shows dramatically increased malicious entity percentages (∼80%) across compartments, severely reduced red entity proportions (mostly 20-30%), and high 'result' values indicating successful divergence from original model characteristics.*

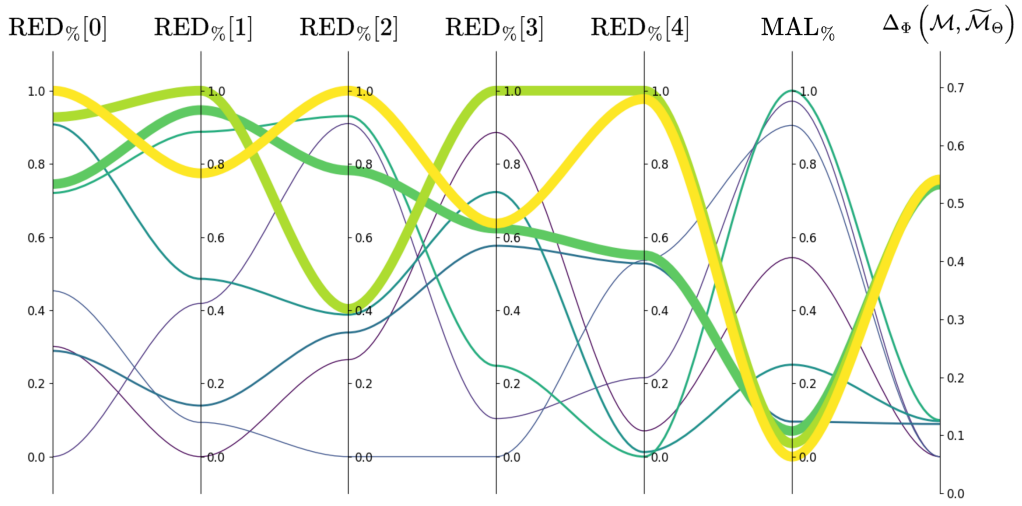


Figure 6.7: *Parallel coordinates plot for the **improvement scenario**. The plot shows significantly increased red entity proportions (60-100%) across most compartments, elimination of malicious entities, and moderately high 'result' values indicating successful enhancement of desired system properties. This configuration optimises system performance by maximising beneficial entities (RED:BLUE ratio of 14:6) and eliminating disruptive elements (MAL = 0) while maintaining a high number of uncertain entities (UNC = 80).*

- Training set size: 475 samples (95% of the total dataset)
- Test set size: 25 samples (5% of the total dataset)

We used Random Forest Regression as our surrogate model and PSO to find the optimal parameter configurations.

Performance Metrics: For each scenario, we evaluated the performance of the surrogate model using the following metrics:

Metric	Preservation	Intentional Divergence	Improvement
In-sample MSE	1.50×10^{-3}	6.27×10^{-4}	1.60×10^{-3}
In-sample RMSE	3.88×10^{-2}	2.50×10^{-2}	4.01×10^{-2}
In-sample R^2	8.50×10^{-1}	9.44×10^{-1}	9.04×10^{-1}
Out-of-sample MSE	3.90×10^{-3}	1.40×10^{-3}	1.70×10^{-3}
Out-of-sample RMSE	6.22×10^{-2}	3.68×10^{-2}	4.07×10^{-2}
Out-of-sample R^2	6.68×10^{-1}	8.57×10^{-1}	7.37×10^{-1}

In-sample performance shows low MSE and RMSE across all scenarios, with Intentional Divergence having the lowest errors (MSE: 6.27e-4, RMSE: 0.0250), indicating the best fit to the training data. Out-of-sample metrics highlight the model's generalisation ability, where Intentional Divergence also maintains the lowest errors (MSE: 0.0014, RMSE: 0.0368), while Preservation has the highest (MSE: 0.0039, RMSE: 0.0622).

R^2 values are high both in-sample and out-of-sample, with Intentional Divergence again showing the best performance (In-sample: 0.9436, Out-of-sample: 0.8568). The decrease in out-of-sample R^2 values indicates some overfitting, particularly in the Preservation scenario (0.8497 vs 0.6675).

Overall, intentional Diversity achieves the best balance of accuracy and generalisation, followed by improvement, with preservation showing slightly lower yet still good performance.

Surrogate optimisation significantly reduced the computational time required for parameter synthesis. Each case was executed in approximately 66.67 seconds (1 minute and 6.67 seconds).

Specifications: in Appendix D Section D.3, Listing D.4 presents the comprehensive specification for the Three-State Protocol comparative analysis. The specification details the original consensus model and its spatial variant, implementing a ring topology with five interconnected nodes. The configuration includes Particle Swarm Optimisation for handling parameter constraints and defines multiple STL formulae for evaluating consensus properties. The sampling strategy utilises 500 samples to thoroughly explore the six-dimensional parameter space, ensuring robust comparative analysis between model variants.

Figure 6.8 provides a comparative view of the average distribution of entities across all three scenarios: preservation, divergence, and improvement.

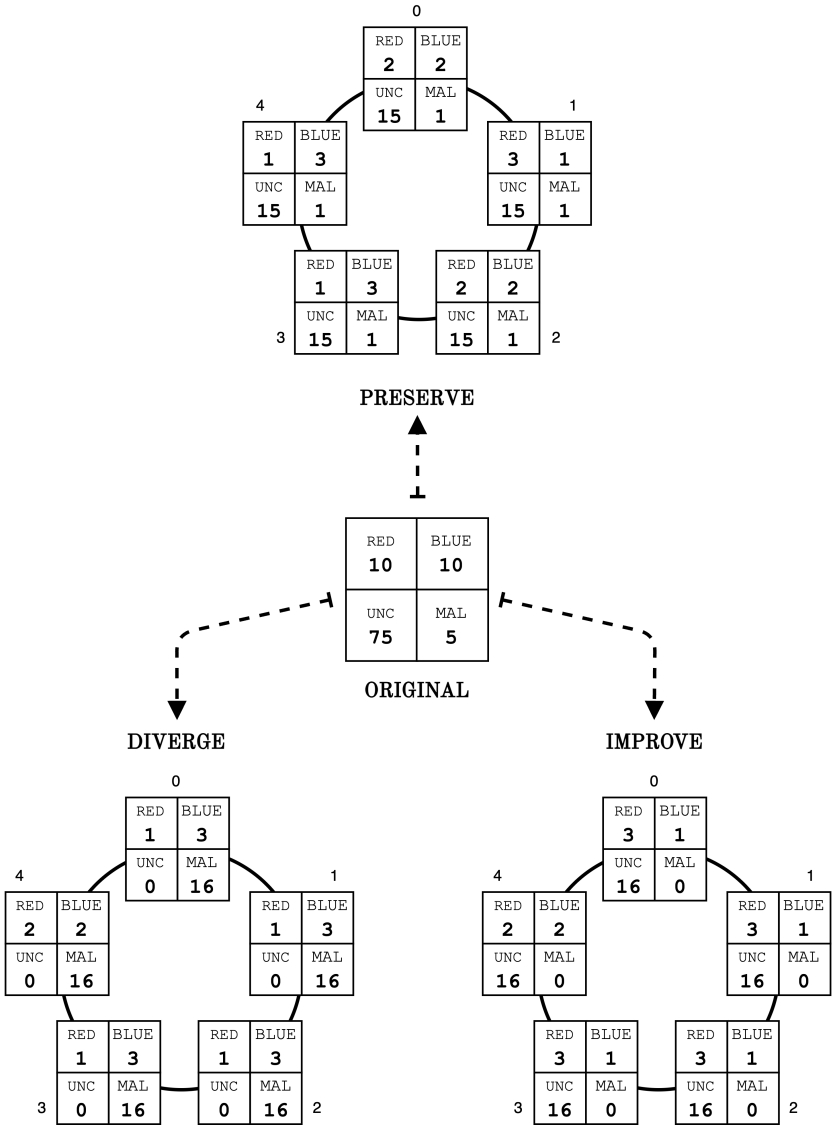


Figure 6.8: *Average distribution of entities in each compartment for preservation, divergence, and improvement scenarios*

Robustness Values

Model	φ_1	φ_2	φ_3	φ_4	Distance
Original	-0.20	-0.42	-0.67	0.15	-
Preserve	-0.19	-0.44	-0.65	0.12	0.08
Diverge	-0.37	-0.49	-0.97	-0.13	0.44
Improve	0.00	-0.36	0.00	0.01	0.72

Property Comparison

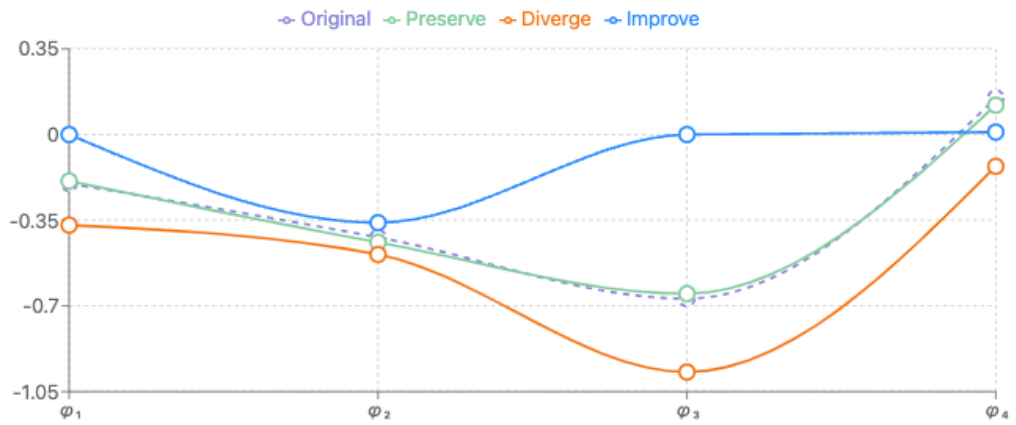


Figure 6.9: *Comparative analysis of robustness values across strategies. Top: Table showing numerical values for each property (φ_1, φ_2, φ_3 and φ_4) and distance from original model. Bottom: Line plot visualising property satisfaction patterns, revealing Preservation's close alignment with Original (dashed), Divergence's deterioration, and Improvement's enhanced performance*



Figure 6.10: *Distribution of entities across network configurations. Top: Original model with global RED:BLUE balance and low malicious presence. Bottom: Variant model showing node-level distributions for each strategy. The bars in RED:BLUE show the proportional presence of competing consensus-forming entities in each node, while MAL:UNC shows the aggregate distribution of Malicious and Uncertain entities for the entire system.*

CHAPTER 7

Conclusion

"Farewell, my brave Hobbits. My work is now finished. Here at last, on the shores of the sea... comes the end of our Fellowship. I will not say do not weep, for not all tears are an evil."

The Lord of the Rings: The Return of the King (2003)

THIS thesis has addressed the critical challenge of developing effective tools and methodologies for system analysis and optimisation in the realm of Collective Adaptive Systems (CAS). Through significant enhancements to the Sibilla framework, we have focused on *parameter synthesis*, *system verification*, and *comparative analysis* capabilities.

The journey from theoretical foundations to practical implementation has revealed our approach's strengths and limitations. Our case studies, ranging from biological systems to consensus protocols, have demonstrated the framework's *versatility* while highlighting areas for future development. The synthesis module, in particular, represents a significant step forward in making sophisticated optimisation techniques accessible to researchers and practitioners working with CAS.

As we reflect on this work's contributions and contemplate future directions, several themes emerge: the importance of balancing *computational efficiency* with *analytical rigor*, the value of *flexible and extensible architectures*, and the ongoing need for tools that bridge theoretical understanding with practical application. These themes guide our discussion of achievements, limitations, and future opportunities in the following sections.

This conclusion chapter synthesises these achievements while critically examining their implications. We begin with a detailed analysis of our research contributions, highlighting how they advance both theoretical understanding and practical capabilities in CAS analysis. We then thoroughly evaluate our framework's strengths and lim-

itations, drawing insights from our case studies and implementation experience. The chapter examines the broader impact of our work and its implications for future research in this field. Finally, we explore promising directions for future development, identifying opportunities to enhance further the capabilities we have developed.

7.1 Research Summary and Contributions

This thesis has presented advancement in the analysis and optimisation of CAS through developing and enhancing the Sibilla framework. Our research has addressed fundamental challenges in parameter synthesis, system verification, and comparative analysis of complex stochastic systems. This work combines theoretical foundations with practical applications, providing researchers and designers with robust tools.

The primary contribution of this work lies in developing a synthesis module that extends Sibilla's capabilities. This module introduces novel approaches to parameter optimisation and system behaviour analysis, enabling more efficient and accurate exploration of complex parameter spaces. Integrating surrogate-based optimisation techniques with statistical model checking has proven effective in handling the computational challenges inherent in CAS analysis.

7.2 Critical Evaluation

This Section critically evaluates the research conducted in this thesis, highlighting its key strengths and achievements and addressing its limitations and challenges. The discussion aims to provide a balanced perspective, acknowledging the significant advancements achieved while identifying areas for future improvement.

7.2.1 Strengths and Achievements

The research presented in this thesis demonstrates several significant strengths and achievements, particularly in terms of framework **extensibility**, **versatility**, and practical **effectiveness**. The synthesis module's architecture represents a significant advancement in modular design for stochastic system analysis. Through careful separation of concerns and well-defined interfaces, we have created a highly extensible framework that readily accommodates new optimisation algorithms, surrogate models, and sampling strategies. This architectural flexibility allows users to integrate novel approaches without modifying the core framework, ensuring the system can evolve alongside advancing methodologies in the field.

The framework's versatility manifests across multiple dimensions. At the modelling level, **it supports diverse formalisms**, ranging from population-based approaches to agent-oriented specifications, enabling researchers to choose the most appropriate representation for their specific problems. The synthesis module demonstrates particular strength in handling different optimisation objectives, from simple parameter tuning to complex multi-objective optimisation scenarios. This adaptability extends to the types of properties being analysed, with support for both qualitative and quantitative evaluation metrics.

Our implementation's effectiveness has been thoroughly validated through diverse case studies. The framework has shown remarkable capability in handling varying

complex models, from relatively straightforward SIR models to sophisticated genetic regulatory networks like the Repressilator.

The comparative analysis framework represents another significant achievement, offering a systematic approach to evaluating model variants. This capability proves invaluable in practical system design, where understanding the implications of different modelling choices can significantly impact system performance. The framework's ability to quantify and analyse differences between model variants provides designers with concrete metrics for making informed decisions about system architecture and implementation.

By **supporting multiple approaches** to both optimisation and surrogate modelling, we enable researchers to select the most appropriate techniques for their specific problems. This flexibility and the framework's efficient implementation create a tool for tackling complex parameter synthesis challenges in collective adaptive systems.

Finally, the **framework's accessibility** has been significantly enhanced through its **Python integration** and cloud deployment capabilities. By providing Python bindings, we leverage one of today's most popular and user-friendly programming languages, making the framework accessible to a broader audience of researchers and practitioners. This integration is compelling when combined with cloud-based development environments like Google Colab, which eliminate traditional barriers to entry by removing the need for local installation and configuration. Users can immediately begin experimenting with the framework through a familiar Jupyter notebook interface, accessing computational resources through the cloud while maintaining a consistent development environment. Furthermore, the ability to easily extend the framework with new capabilities while maintaining backward compatibility ensures its long-term value to the research community and industry practitioners.

7.2.2 Limitations and Challenges

Applying our surrogate-based optimisation framework to diverse case studies has revealed several limitations and technical challenges. Most notably, the **generation of training data** emerges as the primary computational bottleneck, particularly for systems exhibiting complex temporal dynamics. The Repressilator case study clearly illustrates this limitation—models featuring continuous oscillatory behaviour or ongoing temporal evolution demand extensive simulation runs to capture their behavioural characteristics adequately. Unlike systems converging to steady states, such as the SIR model, oscillating systems require substantially more extended simulation periods and more comprehensive sampling to accurately represent their full dynamic range. While our surrogate model training and optimisation phases maintain computational efficiency, this initial data collection phase becomes the dominant constraint on overall performance.

Another crucial limitation centers on determining appropriate training set sizes, mainly when dealing with high-dimensional parameter spaces and complex model behaviours. The framework's performance depends significantly on the proper configuration of multiple elements: training set dimensionality, surrogate model parameters, and optimisation algorithm settings. The selection of training set size proves especially critical, as it must balance computational feasibility against approximation accuracy. Though surrogate quality metrics—including mean squared error, R-squared

values, and cross-validation results—can guide this selection, no universal solution exists. Our empirical investigations suggest that approximately 1000 samples typically provide consistent surrogate model accuracy across complex cases. However, our case studies have demonstrated that more straightforward optimisation tasks can achieve effective results with considerably smaller training sets, often around 100 samples.

These limitations underscore the inherent trade-offs in surrogate-based optimisation approaches. While our framework successfully reduces the computational burden of parameter synthesis compared to direct optimisation methods, it introduces new challenges in experimental design and configuration. The framework's effectiveness ultimately depends on carefully considering these limitations and appropriate adjustment of sampling strategies based on specific model characteristics and optimisation objectives.

7.3 Future Research Directions

The development of the Sibilla framework, while representing significant progress in CAS analysis and optimisation, opens several promising avenues for future research and development. These directions emerge from the limitations we've encountered in our current implementation and from emerging opportunities in collective adaptive systems.

7.3.1 New Synthesis Tasks

Our framework's current synthesis capabilities could be extended in several significant directions. One promising area involves the development of multi-objective synthesis tasks that can simultaneously optimise multiple system properties. This extension would enable researchers to explore trade-offs between competing objectives, such as system performance versus resource utilisation or reliability versus adaptation speed.

A possible direction involves integrating first passage time analysis with our synthesis capabilities. For example, we could implement synthesis tasks that optimise system parameters to achieve specific temporal behaviours, such as minimising the time to reach consensus in distributed protocols or optimising the speed of state transitions in biological systems.

This approach would be valuable for systems where the timing of state changes is critical, such as in emergency response protocols or genetic regulatory networks. The framework could also be enhanced to support real-time synthesis tasks, where parameter adjustments occur dynamically in response to changing system conditions or environmental factors.

The framework capabilities would be particularly valuable for adaptive systems that maintain optimal performance across varying operating conditions. For instance, real-time synthesis could continuously adjust traffic control parameters in smart city applications based on current congestion patterns and emerging traffic demands. Such capabilities would be particularly relevant for applications in robotics, autonomous vehicles, and industrial control systems.

7.3.2 Expanding Case Studies

While our current case studies demonstrate the framework's capabilities across several domains, expanding into new application areas would further validate and refine our approach. Smart city infrastructure presents a particularly promising domain, where the interaction between multiple autonomous systems creates complex optimisation challenges. The framework could be applied to problems such as traffic flow optimisation, resource distribution, and emergency response coordination.

Biological system modelling represents another fertile ground for future case studies. The framework's ability to handle stochastic processes makes it well-suited for analysing genetic regulatory networks beyond the Repressilator, potentially contributing to synthetic biology design and optimisation. Similarly, ecological systems modelling could benefit from our approach, particularly in understanding how parameter variations affect population dynamics and ecosystem stability.

Social network analysis presents a third promising area for case studies. The framework could be adapted to study information diffusion, opinion dynamics, and collective decision-making processes. These applications would validate our methods in new contexts and potentially reveal requirements for additional framework capabilities.

7.3.3 Web-Based Interface Development

Perhaps the most immediate and practical direction for future development lies in creating a comprehensive web-based interface for the framework. This interface would significantly lower the barrier to entry for new users while providing powerful visualisation and analysis capabilities. The development would focus on three key aspects:

First, the interface would provide intuitive tools for synthesis specification, allowing users to define optimisation problems through a graphical interface rather than requiring direct YAML configuration. This would include interactive parameter space definition, objective function construction, and constraint specification. The interface could provide templates for common optimisation scenarios while maintaining the flexibility to handle custom configurations.

Second, the system would offer interactive visualisation of optimisation results. Real-time plotting of optimisation trajectories, parameter sensitivity analysis, and comparative visualisation of different solution strategies would enable users to gain deeper insights into their optimisation problems. The visualisation system could leverage modern web technologies to provide responsive, interactive displays of high-dimensional data.

Current prototype implementations of these web-based interfaces have been developed to demonstrate the feasibility and potential of this approach. These prototypes, automatically generated similar to test report outputs in software development environments, showcase both simulation result visualisation and synthesis optimisation dashboards. A detailed description of these prototype interfaces is provided in Appendix E.

The combination of theoretical advances in synthesis capabilities, practical validation through diverse case studies, and improved accessibility through web-based interfaces presents a compelling roadmap for the future development of the Sibilla framework.

7.4 Concluding Remarks

This thesis represents a step forward in our ability to analyse and optimise CAS. While significant challenges remain, the foundations laid here provide a robust platform for future developments in both theoretical understanding and practical applications. The continuing evolution of CAS in critical applications underscores the importance of further research in this direction.

As we look to the future, the potential applications of our framework continue to expand. The increasing complexity of modern technological systems demands more sophisticated analysis and optimisation tools. Our work provides a foundation for meeting these challenges while pointing the way toward future developments in this crucial field.

The journey of understanding and optimising CAS is incomplete. However, the methodologies and tools developed in this thesis represent significant progress toward this goal. They provide researchers and practitioners with new capabilities for tackling the challenges of complex system design and analysis while opening new avenues for future investigation and development.

Bibliography

[1] Alessandro Abate, Iury Bessa, Daniele Cattaruzza, Lucas Chaves, Lucas Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of provably safe digital controllers for continuous plants. *International Conference on Computer Aided Verification*, pages 462–482, 2017.

[2] Mark A Abramson, Charles Audet, John E Dennis Jr, and Sébastien Le Digabel. Orthomads: A deterministic mads instance with orthogonal directions. *SIAM Journal on Optimization*, 20(2):948–966, 2009.

[3] Michael E Akintunde, Alessio Lomuscio, Lalit Maganti, and Edoardo Pirovano. Verification of decision-making systems via neural network extraction. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1766–1774, 2019.

[4] Rajeev Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[5] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[6] Rajeev Alur and David L. Dill. Algorithmic analysis of hybrid systems. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 278–287. IEEE, 1996.

[7] Rajeev Alur, Tomás Feder, and Thomas A Henzinger. The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, 43(1):116–146, 1996.

[8] Rajeev Alur and Thomas A Henzinger. Real-time logics: Complexity and expressiveness. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 390–401. IEEE, 1990.

[9] Rajeev Alur and Thomas A Henzinger. A really temporal logic. *Journal of the ACM (JACM)*, 41(1):181–204, 1994.

[10] Étienne André, Laurent Fribourg, and Jeremy Sproston. Parametric timed automata for power-aware real-time scheduling. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 110–117. IEEE, 2015.

[11] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[12] R. B. Ash and C. Doléans-Dade. *Probability and Measure Theory*. Harcourt/Academic Press, 2000.

[13] Charles Audet and John E Dennis Jr. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on optimization*, 17(1):188–217, 2006.

[14] Charles Audet and Warren Hare. Chapter 8: Mesh adaptive direct search. In *Introduction: tools and challenges in derivative-free and blackbox optimization*, chapter 8, pages 135–148. Springer, 2017.

[15] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Model-checking continuous-time markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[16] Souheib Baarir, Marco Beccuti, Davide Cerotti, Massimiliano De Pierro, Susanna Donatelli, and Giuliana Franceschinis. The greatspn tool: recent enhancements. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):4–9, 2009.

[17] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*, pages 188–191. IEEE, 1997.

[18] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model checking algorithms for continuous-time markov chains. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR)*, pages 301–317. Springer, 2003.

[19] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[20] Paolo Ballarini, Hilal Djafri, Marie Duflot, Serge Haddad, and Nihal Pekergin. Cosmos: A statistical model checker for the hybrid automata stochastic logic. *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, pages 143–148, 2015.

[21] Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari. Cellular automata: From a theoretical parallel computational model to its application to complex systems. *Cellular Automata: Simplicity Behind Complexity*, 21(2):249–288, 2010.

[22] Mehul Bapodra and Reiko Heckel. A stochastic graph transformation approach to modeling cancer cell behavior. In *Graph Transformation, Specifications, and Nets*, pages 26–40. Springer, 2019.

[23] Jiří Barnat, Lubos Brim, Milan Ceska, and Sven Dražan. Biodivine: a framework for parallel analysis of biological models. *2nd International Workshop on Computational Models for Cell Processes*, 2013.

[24] Ezio Bartocci, Luca Bortolussi, Michele Loreti, and Laura Nenzi. Spatial and spatiotemporal monitoring of cyber-physical systems. *Runtime Verification*, pages 1–20, 2017.

[25] Ezio Bartocci, Luca Bortolussi, Laura Nenzi, and Guido Sanguinetti. System design of stochastic models using robustness of temporal properties. *Theoretical Computer Science*, 587:3–25, 2015.

[26] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. *Lectures on Runtime Verification*, pages 135–175, 2018.

[27] Ezio Bartocci and Pietro Lió. Computational modeling, formal analysis, and tools for systems biology. *PLoS computational biology*, 12(1):e1004591, 2016.

[28] David Benque, Sam Bourton, Caitlin Cockerton, Byron Cook, Jasmin Fisher, Samin Ishtiaq, Nir Piterman, Alex Taylor, and Moshe Y Vardi. Bma: visual tool for modeling and analyzing biological networks. *International Conference on Computer Aided Verification*, pages 686–692, 2012.

[29] Jan A Bergstra, Alban Ponse, and Scott A Smolka. Handbook of process algebra. -, 2001.

[30] Ekaba Bisong. *Google Colaboratory*, pages 59–64. Apress, Berkeley, CA, 2019.

[31] Andrea Bobbio, Dario Bruneo, Davide Cerotti, and Marco Gribaudo. Markovian agents: A new quantitative analytical framework for large-scale distributed interacting systems. *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 311–322, 2011.

[32] Luca Bortolussi. Hybrid behavior of markov population models. *Information and Computation*, 247:235–261, 2015.

[33] Luca Bortolussi, Rocco De Nicola, Vashti Galpin, Stephen Gilmore, Jane Hillston, Diego Latella, Michele Loreti, and Mieke Massink. Carma: Collective adaptive resource-sharing markovian agents. *Workshop on Quantitative Analysis of Programming Languages*, pages 16–31, 2015.

[34] Luca Bortolussi and Jane Hillston. Fluid model checking. *ACM Transactions on Computational Logic*, 16(3):19, 2015.

[35] Luca Bortolussi, Jane Hillston, Diego Latella, and Mieke Massink. Continuous approximation of collective system behaviour: A tutorial. *Performance Evaluation*, 70(5):317–349, May 2013.

[36] Luca Bortolussi and Pietro Lió. Computational models, formal analysis, and tools for systems biology. *PLoS Computational Biology*, 12(1), 2016.

[37] Luca Bortolussi and Simone Silvetti. Efficient robust simulation-based verification of probabilistic systems. *Theoretical Computer Science*, 631:17–33, 2016.

[38] Marc Bouissou, Philippe Préville-Ratelle, and Hassane Chraibi. Pycatshoo: Towards a framework for the analysis of hybrid systems. In *International Conference on Computational Methods in Systems Biology*, pages 259–261. Springer, 2013.

[39] Matthieu Boussard, Duc-Tien Bui, Ilaria Cianci, Rémi Douville, Ludovic Jubert, and Laura Ricci. City pulse: Large scale iot stream processing framework for urban informatics. In *IEEE International Conference on Big Data*, pages 4827–4836. IEEE, 2018.

[40] Pierre Boutillier, Mutaamba Maasha, Xing Li, Héctor F Medina-Abarca, Jean Krivine, Jérôme Feret, Ioana Cristescu, Angus G Forbes, and Walter Fontana. The kappa platform for rule-based modeling. *Bioinformatics*, 34(13):i583–i592, 2018.

[41] Stephen Boyd, Persi Diaconis, and Lin Xiao. Fastest mixing markov chain on a path. *The Annals of Applied Probability*, 14(1):1–27, 2004.

[42] Benoit Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In *Quantitative Evaluation of Systems*, pages 160–164, Berlin, Heidelberg, 2013. Springer.

[43] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability and performance analysis of extended aadl models. In *The Computer Journal*, volume 58, pages 2435–2459. Oxford University Press, 2015.

[44] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[45] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[46] Peter Buchholz. A hierarchical view of gcspns and its impact on qualitative and quantitative analysis. *Journal of Parallel and Distributed Computing*, 15(1):13–30, 1994.

[47] Peter Buchholz. Hierarchical markovian models: Symmetries and reduction. *Performance Evaluation*, 22(1):93–110, 1995.

[48] Sara Bufo, Ezio Bartocci, Guido Sanguinetti, Massimo Borelli, Umberto Lucangelo, and Luca Bortolussi. Temporal logic based monitoring of assisted ventilation in intensive care patients. In *International Symposium on Leveraging Applications of Formal Methods*, pages 391–403. Springer, 2014.

[49] Luca Cardelli. Two-domain dna strand displacement. *Mathematical Structures in Computer Science*, 24(2), 2014.

[50] Luca Cardelli, Marta Kwiatkowska, and Luca Laurenti. Biochemical reaction probabilistic model checking. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 577–586. ACM, 2017.

[51] Luca Cardelli, Marta Kwiatkowska, and Max Whitby. Syntax-guided optimal synthesis for chemical reaction networks. *International Conference on Computer Aided Verification*, pages 375–395, 2016.

[52] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer, 2008.

[53] Martin Cenek, Spencer Dahl, Sarah Malone, Lorcan Buchanan, Lucas Aufdencamp, Anushka Challa, and Rachel Smith. Parameter optimization for agent-based models using bayesian inference. In *AAAI Fall Symposium Series*, 2018.

[54] Milan Češka, Frits Dannenberg, Nicola Paoletti, Marta Kwiatkowska, and Luboš Brim. Prophesy: A probabilistic parameter synthesis tool. In *International Conference on Computer Aided Verification*, pages 214–231. Springer, 2014.

[55] Nathalie Chabrier-Rivier, Francois Fages, and Sylvain Soliman. The biochemical abstract machine biocham. *International Conference on Computational Methods in Systems Biology*, pages 172–191, 2004.

[56] Claudine Chaouiya, Aurélien Naldi, and Denis Thieffry. Logical modelling of gene regulatory networks with ginsim. *Methods in molecular biology*, 804:463–479, 2012.

[57] Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop. The complexity of reachability problems for constant-rate multi-mode systems. *Information and Computation*, 220:23–35, 2012.

[58] Xin Chen, Sriram Sankaranarayanan, and Erika Ábrahám. Flow*: An analyzer for non-linear hybrid systems. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, pages 258–263. Springer, 2013.

[59] Vincenzo Ciancia, Diego Latella, Michele Loreti, and Mieke Massink. Specifying and verifying properties of space. In *International Conference on Theoretical Computer Science*, pages 222–235. Springer, 2014.

[60] Vincenzo Ciancia, Diego Latella, Mieke Massink, and Rytis Pakauskas. Spatial and spatial-temporal logics for monitoring collective adaptive systems. In *Leveraging Applications of Formal Methods, Verification and Validation*, pages 130–150. Springer, 2018.

[61] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Parameter synthesis with ic3. In *International Conference on Formal Methods in Computer-Aided Design*, pages 165–168. IEEE, 2015.

[62] Federica Ciocchetta and Jane Hillston. Bio-pepa: A framework for the modeling and analysis of biological systems. *Theoretical Computer Science*, 410(33-34):3065–3084, 2009.

[63] Federica Ciocchetta and Jane Hillston. Bio-pepa: A framework for the modelling and analysis of biological systems. *Theoretical Computer Science*, 410(33-34):3065–3084, 2009.

[64] Federica Ciocchetta and Jane Hillston. Stochastic modelling for collective behaviour of populations. *Electronic Notes in Theoretical Computer Science*, 310:27–45, 2016.

[65] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. *Journal of the ACM*, 50(5):752–794, 2004.

[66] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1982.

[67] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, pages 154–169. Springer, 2000.

[68] Edmund M. Clarke and Paolo Zuliani. Statistical model checking for cyber-physical systems. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 1–12. Springer, 2012.

[69] Anne Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

[70] James B Dabney and Thomas L Harman. Simulation and model-based design using simulink® and stateflow®. *The MathWorks*, 2011.

[71] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling of cellular signalling. In *CONCUR 2007: 18th International Conference on Concurrency Theory*, volume 4703, pages 17–41. Springer, 2007.

[72] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Scalable simulation of cellular signaling networks. In *Asian Symposium on Programming Languages and Systems*, pages 139–157. Springer, 2007.

[73] Daniele D'Aprile, Susanna Donatelli, and Jeremy Sproston. Probabilistic verification for ebusiness. *Electronic Notes in Theoretical Computer Science*, 151(2):135–155, 2018.

[74] Alexandre David, Dehui Du, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for networks of priced timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 80–96, Cham, 2015. Springer.

[75] Alexandre David, Dehui Du, Kim G Larsen, Axel Legay, and Marius Mikučionis. Statistical model checking for stochastic hybrid systems. In *HSB@ ETAPS*, pages 122–136, 2015.

[76] Conrado Daws. Symbolic model checking for probabilistic timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 293–308. Springer, 2004.

[77] Conrado Daws. Symbolic and parametric model checking of discrete-time markov chains. In *Proceedings of the 1st International Conference on Theoretical Aspects of Computing (ICTAC)*, pages 280–294. Springer, 2005.

[78] Luca De Alfaro, Thomas A. Henzinger, and Rupak Majumdar. Quantitative verification: Models, techniques and tools. *Communications of the ACM*, 61(4):62–71, 2018.

[79] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005.

[80] Rocco De Nicola, Diego Latella, Michele Loreti, and Mieke Massink. Formal methods for collective adaptive systems: The road ahead. In *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems*, pages 3–18. Springer, 2014.

[81] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The scel language. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2):1–29, 2014.

[82] Donald L DeAngelis and Simeon Yurek. Pattern-oriented modeling of collective adaptive systems: the role of spatial local patterns. *Ecological Modelling*, 298:35–46, 2015.

[83] David D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, James M. Doyle, William H. Sanders, and Peter G. Webster. The möbius framework and its implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.

[84] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Benzmüller Harold, Joost-Pieter Bruintjes, Joost-Pieter Katoen, and Milan Češka. Parameter synthesis for markov models: Faster than ever. *IEEE International Symposium on Automated Technology for Verification and Analysis*, pages 50–67, 2015.

[85] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Benzmüller Harold, Joost-Pieter Bruintjes, Joost-Pieter Katoen, and Milan Češka. Prophesy: An infrastructure for parameter synthesis of probabilistic models. In *International Conference on Computer Aided Verification*, pages 349–366. Springer, 2015.

[86] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. Computing conditional probabilities: Implementation and evaluation. In *International Symposium on Formal Methods*, pages 349–366. Springer, 2016.

[87] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *Computer Aided Verification*, pages 592–600, Cham, 2017. Springer.

[88] Lorenzo Dematté, Corrado Priami, and Alessandro Romanel. The beta workbench: a computational tool to study the dynamics of biological systems. *Briefings in bioinformatics*, 9(5):437–449, 2008.

[89] Jyotirmoy V Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51:5–30, 2017.

[90] Andreas Deutsch and Sabine Dormann. Cellular automaton modeling of biological pattern formation: Characterization, applications, and analysis. -, 2005.

[91] Alastair F Donaldson and Alice Miller. Symmetry reduction for probabilistic model checking using generic representatives. *International Symposium on Automated Technology for Verification and Analysis*, pages 9–23, 2009.

[92] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *International Conference on Computer Aided Verification*, pages 167–170. Springer, 2010.

[93] Alexandre Donzé, Gilles Clermont, and Christopher James Langmead. Parameter synthesis in nonlinear dynamical systems: Application to systems biology. *Journal of computational biology*, 17(3):325–336, 2010.

[94] Alexandre Donzé, Eric Fanchon, Lucie Martine Gattepaille, Oded Maler, and Philippe Tracqui. Robustness analysis and behavior discrimination in enzymatic reaction networks. *PloS one*, 6(9):e24246, 2011.

[95] Alexandre Donzé, Thomas Ferrere, and Oded Maler. Efficient robust monitoring for stl. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 264–279. Springer, 2013.

[96] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 92–106. Springer, 2010.

[97] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *International Conference on Computer Aided Verification*, pages 167–170. Springer, 2010.

[98] Elena Dubrova and Maxim Teslenko. A sat-based algorithm for finding attractors in synchronous boolean networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(5):1393–1399, 2010.

[99] Pete Edwards, Mike Jackson, Geoffrey Bowker, and Cory Knobel. Computational models in community-based ecological research: A distributed approach to adaptive management. *BioScience*, 59(9):803–813, 2009.

[100] Steven Eker, Mary Knapp, Keith Laderoute, Patrick Lincoln, José Meseguer, and M Kemal Sönmez. Pathway logic: symbolic analysis of biological signaling. *Pacific symposium on biocomputing*, 9:400–412, 2004.

[101] E Allen Emerson and Edmund M Clarke. Model checking and the mu-calculus. In *Logical Foundations of Computer Science*, pages 185–214. Springer, 2003.

[102] E Allen Emerson and Joseph Y Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[103] James R Faeder, Michael L Blinov, and William S Hlavacek. Rule-based modeling of biochemical systems with bionetgen. *Methods in molecular biology*, 500:113–167, 2009.

[104] François Fages, Steven Gay, and Sylvain Soliman. Synthesis of biochemical reaction networks from temporal logic specifications. *Theoretical Computer Science*, 765:33–51, 2018.

[105] Cheng Feng and Jane Hillston. Paloma: A process algebra for located markovian agents. *International Conference on Quantitative Evaluation of Systems*, pages 265–280, 2014.

[106] Cheng Feng, Jane Hillston, Qingwei Lin, and Miao Cui. Mean-field game theory for population dynamics and analytical modeling of critical information dissemination in social networks. *IEEE Transactions on Network Science and Engineering*, 6(4):751–763, 2019.

[107] Ionut Florescu. *Probability and stochastic processes*. John Wiley & Sons, 2014.

[108] Barry L. Fox and Peter W. Glynn. Implementing the randomization method for transient solutions of continuous-time markov chains. *Communications of the ACM*, 31(4):440–451, 1988.

[109] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *International Journal on Software Tools for Technology Transfer*, volume 10, pages 263–279. Springer, 2008.

[110] Vashti Galpin, Luca Bortolussi, and Jane Hillston. Hybrid semantics for hype: A process algebra for flows and continuous change. *Information and Computation*, 236:17–39, 2013.

[111] Vashti Galpin, Vincenzo Ciancia, Diego Latella, and Mieke Massink. Spatial representations and analysis techniques. pages 120–155, 2016.

[112] Marnix JJ Garvels, Jan-Kees Van Ommeren, and Dirk P Kroese. A cross-entropy method for estimating probabilities of rare events. *European Transactions on Telecommunications*, 11(3):325–336, 2000.

[113] Michael A Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The journal of physical chemistry A*, 104(9):1876–1889, 2000.

[114] Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.

[115] Daniel T Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of chemical physics*, 115(4):1716–1733, 2001.

[116] Winfried K. Grassmann. Transient solutions in markovian queueing systems. *Computers & Operations Research*, 4(1):47–53, 1977.

[117] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. Parameter synthesis for probabilistic systems. In *International Workshop on Methods for Modelling Software Systems*, pages 155–162. Springer, 2010.

[118] Ernst Moritz Hahn, Holger Hermanns, and Lijun Zhang. Parametric markov chains: Pctl complexity and fraction-free gaussian elimination. *Electronic Proceedings in Theoretical Computer Science*, 57:13–25, 2011.

[119] Ernst Moritz Hahn, Holger Hermanns, and Lijun Zhang. Parametric markov models: At most one parameter is sufficient. *International Journal on Software Tools for Technology Transfer*, 13(3):291–304, 2011.

[120] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[121] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[122] Leonard A Harris, Justin S Hogg, José-Juan Tapia, John AP Sekar, Sanjana Gupta, Ilya Korsunsky, Arshi Arora, Dipak Barua, Robert P Sheehan, and James R Faeder. Bionetgen 2.2: Advances in rule-based modeling. *Bioinformatics*, 32(21):3366–3368, 2016.

[123] Arnd Hartmanns and Holger Hermanns. The modest toolset: An integrated environment for quantitative modelling and verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 593–598. Springer, 2014.

[124] Boudewijn R. Haverkort. *Performance Evaluation of Computer and Communication Systems*. Wiley, 2001. Chapter on Markovian Models.

[125] Reiko Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006.

[126] M. Heiner, D. Gilbert, and R. Donaldson. *Petri Nets for Systems and Synthetic Biology*, volume 5016, pages 215–264. Springer, 2008.

[127] Monika Heiner, Mostafa Herajy, Fei Liu, Christian Rohr, and Martin Schwarick. Snoopy—a unifying petri net tool. *International Conference on Application and Theory of Petri Nets and Concurrency*, pages 398–407, 2012.

[128] David Henriques, Paolo Zuliani, Ana Gomes, and Marta Kwiatkowska. Statistical model checking for markov decision processes. In *Proceedings of the 9th International Conference on Quantitative Evaluation of Systems (QEST)*, pages 84–93. IEEE, 2012.

[129] Thomas A. Henzinger. The theory of hybrid automata. *Verification of Digital and Hybrid Systems*, pages 265–292, 2000.

[130] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[131] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[132] Jane Hillston and Michele Loreti. Process algebras for collective dynamics: Putting individuals in their environment. *Electronic Notes in Theoretical Computer Science*, 310:83–99, 2016.

[133] Mike Holcombe. A framework for modelling multi-agent systems. In *Proceedings of Conference on AI, Simulation, and Planning in High Autonomy Systems*, pages 40–45. IEEE, 2001.

[134] Florian Hölzl, Axel Rauschmayer, and Martin Wirsing. *Engineering modeling languages: Turning domain knowledge into tools*. Chapman and Hall/CRC, 2008.

[135] Matthias Hölzl, Axel Rauschmayer, and Martin Wirsing. Engineering of software-intensive systems: State of the art and research challenges. *Software-Intensive Systems and New Computing Paradigms*, pages 1–44, 2008.

[136] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. Copasi—a complex pathway simulator. *Bioinformatics*, 22(24):3067–3074, 2006.

[137] Jianghai Hu, John Lygeros, and Shankar Sastry. Towards a theory of stochastic hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 160–173. Springer, 2000.

[138] Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Parameter synthesis in markov models: A gentle survey. *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, pages 407–437, 2022.

[139] A. Jensen. Markoff chains as an aid in the study of markoff processes. *Skandinavisk Aktuarietidskrift*, 36:87–91, 1953.

[140] Susmit Jha, Edmund Clarke, Christopher Langmead, Axel Legay, André Platzer, and Paolo Zuliani. Bayesian model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 341–355. Springer, 2009.

[141] Mathias John, Roland Ewald, and Adelinde M Uhrmacher. Palps: A stochastic process algebra with distributed spatial locations. *Electronic Notes in Theoretical Computer Science*, 261:91–110, 2010.

[142] Mathias John, Cédric Lhoussaine, Joachim Niehren, and Adelinde M Uhrmacher. A spatial extension to the π calculus. In *International Workshop on Concurrent Objects and Beyond*, pages 82–98. Springer, 2008.

[143] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. A complete method for program abstraction. In *2010 IEEE International Conference on Computer Design*, pages 296–301. IEEE, 2010.

[144] Joost-Pieter Katoen, Christel Baier, Holger Hermanns, and Lijun Zhang. The expected time to reach a goal. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS)*, pages 13–22. IEEE, 2010.

[145] Joost-Pieter Katoen, Maneesh Khattri, and Ivan S Zapreev. A markov reward model checker. *International Conference on Quantitative Evaluation of Systems*, pages 243–244, 2007.

[146] Joost-Pieter Katoen, Ivan S Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance Evaluation*, 68(2):90–104, 2011.

[147] Joost-Pieter Katoen, Ivan S Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance Evaluation*, 68(2):90–104, 2012.

[148] Stuart A Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467, 1969.

[149] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. Jupyter notebooks - a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*, pages 87–90. IOS Press, 2016.

[150] Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. dreach: delta-reachability analysis for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 200–205. Springer, 2015.

[151] Jean Krivine and Jérôme Feret. Scalable simulation of cellular signaling networks, a new rule-based implementation of the kappa simulator. *International Conference on Computational Methods in Systems Biology*, pages 1–18, 2018.

[152] Vidyadhar G Kulkarni. *Modeling and analysis of stochastic systems*. Chapman and Hall/CRC, 2016.

[153] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *Proceedings of the 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[154] Marta Kwiatkowska, Gethin Norman, and David Parker. Symmetry reduction for probabilistic model checking. *Formal Methods in System Design*, 29(2):161–196, 2006.

[155] Marta Kwiatkowska, Gethin Norman, and David Parker. Quantitative verification: Models, techniques and tools. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 449–458. ACM, 2007.

[156] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. *International Conference on Computer Aided Verification*, pages 585–591, 2011.

[157] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. *International conference on computer aided verification*, pages 585–591, 2011.

[158] Marta Kwiatkowska, Gethin Norman, and David Parker. Quantitative verification techniques for biological processes. *Algebraic and Numeric Biology*, pages 155–173, 2011.

[159] Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Assume-guarantee verification for probabilistic systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 23–37. Springer, 2010.

[160] Thomas Le Gleize, Rosemary Monahan, and Jane Hillston. Distributed statistical model checking of rare properties using importance splitting observers. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV)*, pages 58–80. Springer, 2021.

[161] Axel Legay, Benoit Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *International Conference on Runtime Verification*, pages 122–135. Springer, 2010.

[162] Daniel Lemire. Streaming maximum-minimum filter using no more than three comparisons per element. *arXiv preprint cs/0610046*, 2006.

[163] Yuri K Lopes, Stefan M Trenkwalder, André B Leal, Tony J Dodd, and Roderich Groß. Supervisory control theory of swarm robotics systems: A case study. *Swarm Intelligence*, 10(1):65–84, 2016.

[164] Michele Loreti and Jane Hillston. Modeling and analysis of collective adaptive systems with carma and its tools. In *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems*, pages 83–119. Springer, 2016.

[165] Michele Loreti and Aniqa Rehman. A logical framework for reasoning about local and global properties of collective systems. In Erika Ábrahám and Marco Paolieri, editors, *Quantitative Evaluation of Systems - 19th International Conference, QEST 2022, Warsaw, Poland, September 12-16, 2022, Proceedings*, volume 13479 of *Lecture Notes in Computer Science*, pages 133–149. Springer, 2022.

[166] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. Mason: A multi-agent simulation environment. In *Simulation: Transactions of the Society for Modeling and Simulation International*, volume 81, pages 517–527, 2005.

[167] Charles M Macal and Michael J North. Tutorial on agent-based modeling and simulation. -, pages 2–15, 2005.

[168] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[169] George Mason and Radu Calinescu. Stochastic programs as decision-making models for gambling problems. In *International Conference on Quantitative Evaluation of Systems*, pages 123–142. Springer, 2018.

[170] Carsten Maus, Stefan Rybacki, and Adelinde M Uhrmacher. Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology*, 5(1):166, 2011.

[171] Robin Milner. Communication and concurrency. -, 1989.

[172] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[173] Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and api gateways in microservices. *arXiv preprint arXiv:1609.05830*, 2016.

[174] Vimal A Muganthan, Andrew Phillips, and Maria G Vigliotti. Bam: Bioambient machine. *Proceedings of ACSD 2008*, pages 45–49, 2008.

[175] Kevin Patrick Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. University of California, Berkeley, 2002.

[176] Christoph Müssel, Martin Hopfensitz, and Hans A Kestler. Boolnet—an r package for generation, reconstruction and analysis of boolean networks. *Bioinformatics*, 26(10):1378–1380, 2010.

[177] Elena Nardini, Andrea Omicini, and Mirko Viroli. Engineering collective adaptive systems by coordination. *Concurrency and Computation: Practice and Experience*, 29(2), 2017.

[178] Laura Nenzi, Luca Bortolussi, Vincenzo Ciancia, Michele Loreti, and Mieke Massink. Mela: Modelling in ecology with location attributes. *International Conference on Quantitative Evaluation of Systems*, pages 1–16, 2018.

[179] Michael J North, Nicholson T Collier, Jonathan Ozik, Eric R Tatara, Charles M Macal, Mark Bragen, and Pamela Sydelko. *Complex Adaptive Systems Modeling with Repast Simphony*. Springer, 2013.

[180] Michael J North, Nicholson T Collier, and Jerry R Vos. Experiences creating three implementations of the repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 16(1):1–25, 2006.

[181] Peter Olofsson. *Probability, Statistics, and Stochastic Processes*. Wiley, 2005.

[182] Peter Olofsson and Mikael Andersson. *Probability, statistics, and stochastic processes*. John Wiley & Sons, 2012.

[183] Andrew Phillips and Luca Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. *International Conference on Computational Methods in Systems Biology*, pages 184–199, 2007.

[184] Diego GS Pivoto, Luiz FF de Almeida, Rodrigo da Rosa Righi, Joel JPC Rodrigues, Alexandre Baratella Lugli, and Antonio M Alberti. Cyber-physical systems architectures for industrial internet of things applications in industry 4.0: A literature review. *Journal of manufacturing systems*, 58:176–192, 2021.

[185] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.

[186] Amir Pnueli. The temporal logic of programs. -, pages 46–57, 1977.

[187] Stephen Prajna and Ali Jadbabaie. Safety verification using barrier certificates. *IFAC Proceedings Volumes*, 37(13):543–548, 2004.

[188] Corrado Priami. Stochastic pi-calculus. *The Computer Journal*, 38(7):578–589, 1995.

[189] PRISM Model Checker Team. Case study: Molecules. `https://www.prismmodelchecker.org/casestudies/molecules.php`, 2025. Accessed: 2025-05-28.

[190] Tim Quatmann, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Parameter synthesis for markov models: Faster than ever. In *International Symposium on Automated Technology for Verification and Analysis*, pages 50–67. Springer, 2016.

[191] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with flame for the gpu. *Briefings in bioinformatics*, 11(3):334–347, 2010.

[192] Aurélien Rizk, Grégory Batt, François Fages, and Sylvain Soliman. On a continuous degree of satisfaction of temporal logic formulae with applications to systems biology. In *International Conference on Computational Methods in Systems Biology*, pages 251–268. Springer, 2008.

[193] Gerardo Rubino and Bruno Tuffin. *Rare Event Simulation Using Monte Carlo Methods*. John Wiley & Sons, 2009.

[194] Susan M Sanchez and TW Lucas. Effective parameter tuning for discrete event simulation models. *Proceedings of the Winter Simulation Conference*, pages 373–383, 2011.

[195] Stefano Sebastio and Andrea Vandin. Distributed statistical analysis of complex systems modeled through a chemical metaphor. In *International Conference on High Performance Computing and Simulation*, pages 416–423. IEEE, 2013.

[196] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box systems. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, pages 202–215. Springer, 2005.

[197] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[198] William J. Stewart. *Numerical Solution of Markov Chains*. Marcel Dekker, 2000.

[199] Maciej H Swat, Gilberto L Thomas, Julio M Belmonte, Abbas Shirinifard, Dimitrij Hmeljak, and James A Glazier. Compucell3d: a multi-model framework for simulation of morphogenesis. *Bioinformatics*, 28(7):1040–1046, 2012.

[200] Kivy Team and other contributors. pyjnius. `https://github.com/kivy/pyjnius`.

[201] René Thomas and Marcelle Kaufman. Multistationarity, the basis of cell differentiation and memory. ii. logical analysis of regulatory networks in terms of feedback circuits. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 11(1):180–195, 2001.

[202] Mirco Tribastone, Adam Duguid, and Stephen Gilmore. Scalable differential analysis of process algebra models. *IEEE Transactions on Software Engineering*, 38(1):205–219, 2012.

[203] Aad van Moorsel and William H Sanders. Computational evaluation of dependability attributes for fast and accurate analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(2):300–314, 1994.

[204] Abraham Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[205] Uri Wilensky. Netlogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 1999.

[206] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

[207] Rajaa Vikhram Yohanandhan, Rajvikram Madurai Elavarasan, Premkumar Manoharan, and Lucian Mihet-Popa. Cyber-physical power system (cpps): A review on modeling, simulation, and analysis with cyber security applications. *IEEE Access*, 8:151019–151064, 2020.

[208] Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. In *Information and Computation*, volume 204, pages 1368–1409. Elsevier, 2006.

[209] Håkan LS Younes and Reid G Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Information and Computation*, 204(9):1368–1409, 2006.

[210] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, 2000.

[211] Paolo Zuliani, André Platzer, and Edmund M. Clarke. Statistical model checking for cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):1–19, 2013.

APPENDIX A

Specification Languages for Model Definition

The effectiveness of a modelling framework is intrinsically linked to the expressiveness and clarity of its specification languages. These languages interface the modeler's conceptual understanding and the framework's computational capabilities, making their design and implementation crucial to the framework's utility. This chapter presents a detailed description of the specification languages implemented in Sibilla, each carefully crafted to support different modelling paradigms while maintaining consistency in their fundamental structure.

At their core, these specification languages share common syntactic elements that provide a familiar foundation across different modelling approaches. These shared elements include basic constructs for expressing mathematical operations, logical conditions, and state transitions. However, each language extends this common base with specialised constructs that capture the unique characteristics of its target modelling paradigm, whether it be population-based systems, interactive objects, agent-based systems, or stochastic processes.

The chapter begins with an overview of the common syntactic elements shared across all specification languages, providing a foundation for understanding the more specialised constructs that follow. We then proceed with detailed presentations of each language.

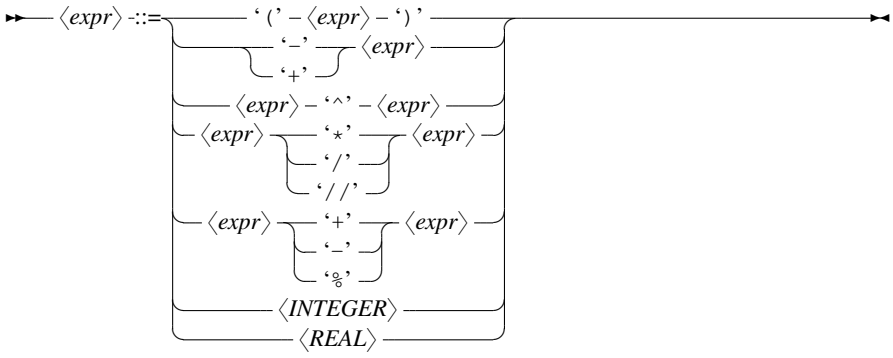
The formal syntax definitions presented here serve as a reference for correct model specification and a guide to the expressive possibilities available within each modelling paradigm. Through careful study of these languages, modelers can better understand how to translate their conceptual models into precise, executable specifications.

A.1 Common syntax

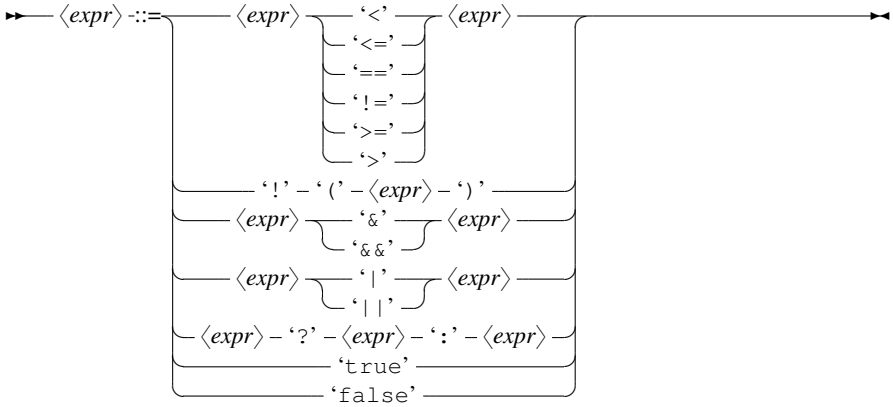
We start by defining some essential components such as numerals, expressions and identifiers and then move on to the components closely related to the LPM specification language.

Expressions In our languages, there is a comprehensive rule for expressions encompassing various types of operations and values. We can conceptually divide these into arithmetic expressions, boolean expressions, reference expressions, function calls, and constants for clarity and easier understanding. However, it's crucial to remember that this division is purely for explanation purposes, and the actual grammar treats all of these as part of a single $< expr >$ rule.

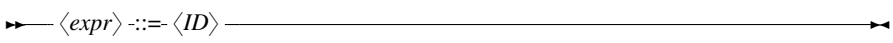
Arithmetic Expressions These involve numerical operations and values.



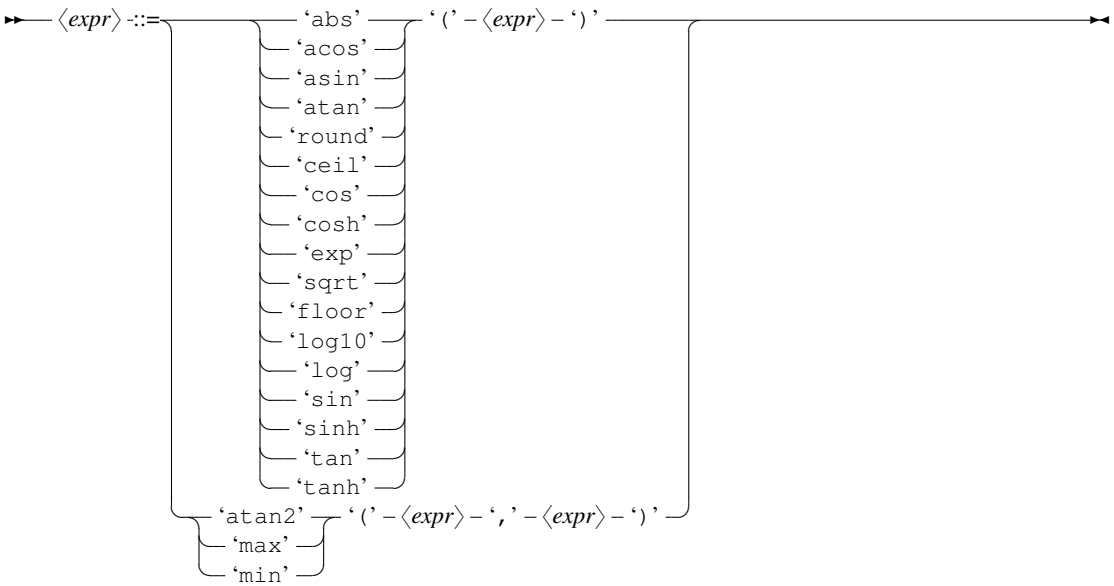
Boolean Expressions These involve logical operations and comparisons.



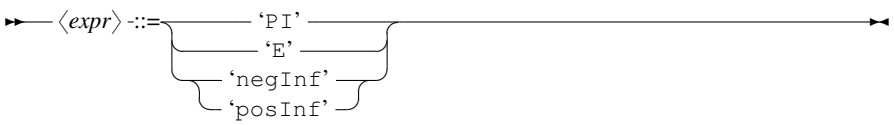
Reference Expressions These involve identifiers that refer to variables or other named entities in the model specification.



Function Calls These represent invocations of mathematical and other functions.



Constants These represent predefined constant values.



Digits and Letters Digits and letters are the basic building blocks for identifiers and numeric literals.



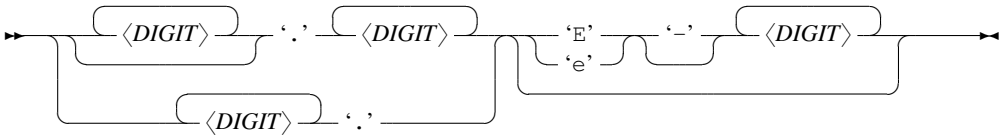


Identifiers Identifiers are names for parameters, constants, and other components (e.g. measures).



Integer Literals Integer literals consist of one or more digits.

Real Number Literals Real number literals can have an optional integer part, a decimal point, an optional fractional part, and an optional exponent.



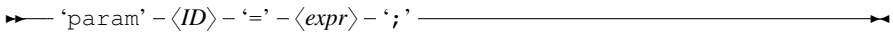
Constants Constants define fixed values in the model that cannot be changed during simulation.



For example:

```
const MAX_POPULATION = 10000;
const GRID_SIZE = 100;
const SIMULATION_TIME = 365;
```

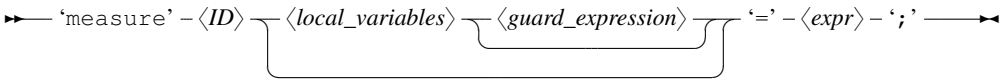
Parameters Parameters are similar to constants but can be adjusted to explore different scenarios.



For example:

```
param infection_rate = 0.3;
param recovery_rate = 0.1;
param birth_rate = 0.05;
```

Measures Measures allow for quantifying and tracking specific aspects of the population over time.

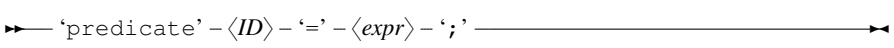


For example, for the population model:

```
measure totalInfected =
            sum(x in [0,GRID_SIZE], y in [0,GRID_SIZE]: #Infected[x,y]);

measure infectionDensity = totalInfected / (GRID_SIZE * GRID_SIZE);
```

Predicates Predicates are Boolean expressions that define specific conditions or properties of the system.



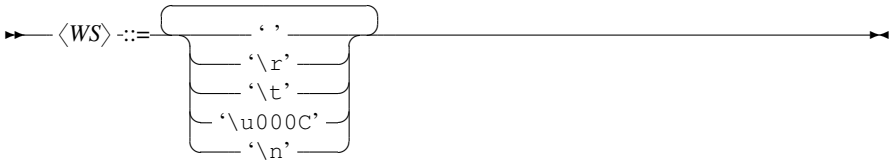
For example:

```
predicate epidemicOver = (totalInfected == 0);
predicate criticalState = (infectionDensity > 0.5)
                          || (reproductionNumber > 2);
```

Comments Comments are used to add notes or explanations in the code that the compiler ignores. This grammar supports C-style block comments.



Whitespace Whitespace includes spaces, tabs, form feeds, and newlines, which are typically ignored by the compiler except to separate tokens.

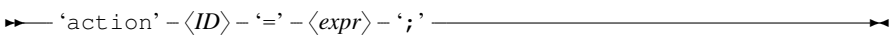


Note: In the actual language implementation, both <COMMENT> and <WS> are directed to a hidden channel, meaning that they are recognised by the lexer but not passed to the parser.

A.2 LIO Model Specification Language

This section formally describes the syntax for each component specific to the LIO (Labelled Input/Output) model specification language, using railroad diagrams.

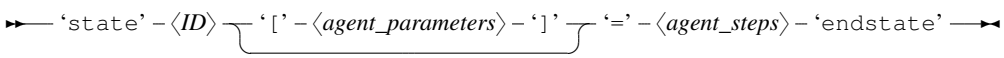
Actions Actions represent events or transitions that can occur in the system.

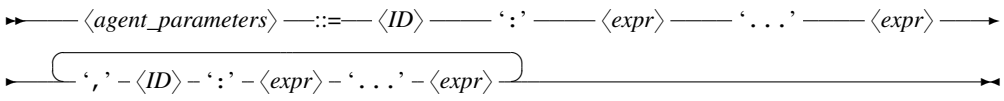


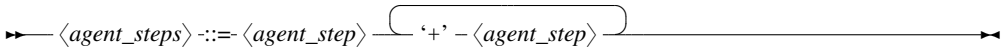
For example:

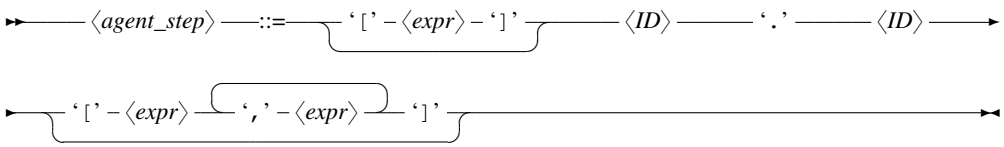
```
action changeInBlue = %R * meetingProbability;
action changeInRed = %B * meetingProbability;
```

States States define the possible configurations of agents in the system.





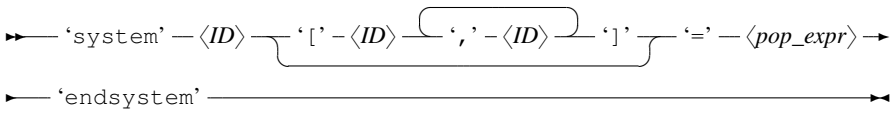




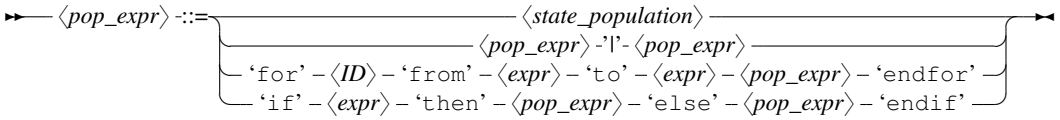
For example:

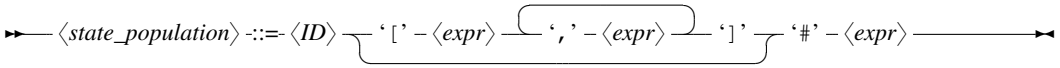
```
state B[i: 0 .. K] =
    [i>0] perceiveB.B[i-1]
    + [i==0] perceiveB.R[0]
    + [i<K] perceiveR.B[i+1]
endstate
```

System The system declaration defines the initial configuration of the model.



To simplify the diagrammatic representation, *population_expression* has been abbreviated to *pop_expr*.

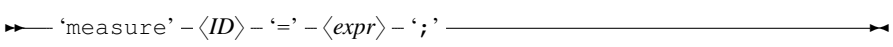




For example:

```
system InitialConfiguration =
    B[K]#100 | R[K]#100
endsystem
```

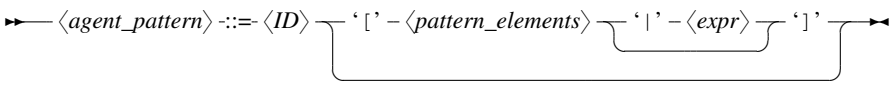
Measures Measures allow for quantifying and tracking specific aspects of the population over time.

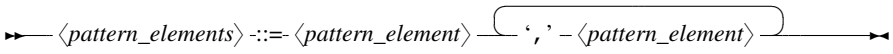


For example:

```
measure fractionOfR = %R[_];
measure fractionOfBWithHighConfidence = %B[i | i>3];
```

Agent Patterns Agent patterns are used in expressions to select specific subsets of the population.







For example:

```
%R[_]                   // All agents in state R
%B[i | i>3]             // Agents in state B with i > 3
```

Note: The LIO language uses '%' to represent the fraction of agents matching a pattern, similar to how it's used in the LPM language.

Example A.2.1 (LIO example)
Let us consider the scenario presented in [165], where we have N agents that can be either *blue* or *red* coloured. The goal is to guarantee that without any centralised control, agents evolve to a *balanced configuration* where the number of *blue agents* is *similar* to the number of *red agents*. In the system, each agent can autonomously change its colour according to its perceptions of the state of the other agents.

```
const p = 0.5
action toRed = p*%B/(%B+%R)
action toBlue = p*%R/(%B+%R)

state R =
    toBlue.B
endstate

state B =
    toRed.R
endstate
```

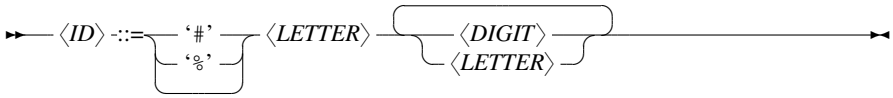
Listing A.1: *LIO esample*

We can assume that each agent in the system can be either in the state B, indicating that the agent is *blue coloured*, or in the state R, when the agent is *red*.
At each step, a blue agent B (resp. a red agent R) interacts with another agent with a given probability p. If this agent is B (resp. R) it changes its colour by performing action toRed (resp. toBlue).

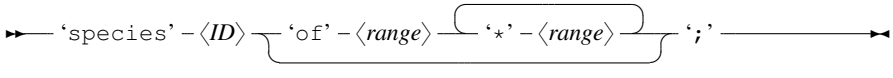
A.3 LPM Model Specification Language

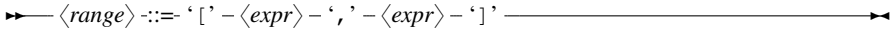
This section uses railroad diagrams (also called syntax diagrams) to provide a formal description of the syntax for each component of the population model specification language.

Identifiers Identifiers are names for parameters, constants, and other components (e.g. measures). In the context of LPM, the symbols '#' and '%' are used explicitly for species. The symbol '#' represents the quantity of a species, while '%' indicates the percentage of a species in the system.



Species Species represent the different types or states of individuals within the population.

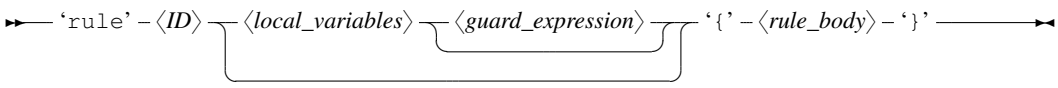




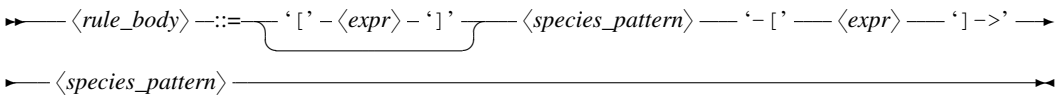
For example:

```
species Susceptible of [0, GRID_SIZE] * [0, GRID_SIZE];
species Infected of [0, GRID_SIZE] * [0, GRID_SIZE];
species Recovered;
```

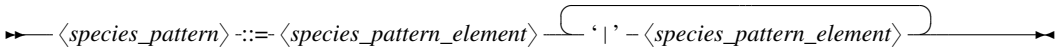
Rules Rules describe how individuals transition between species and interact with each other.



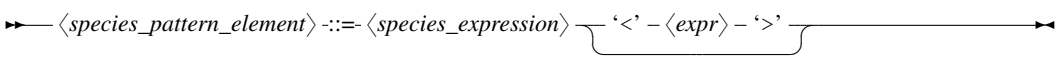
A rule consists of an identifier, optional local variables and guard expression, and a rule body.



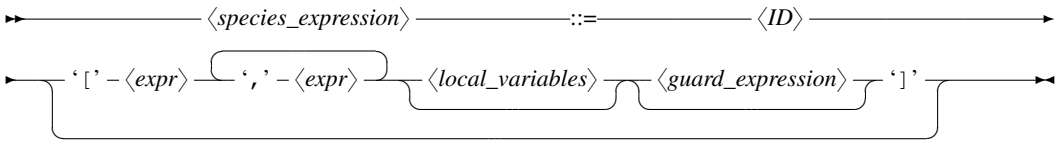
The rule body defines the transformation from one species pattern to another, with an optional global condition and a rate expression.



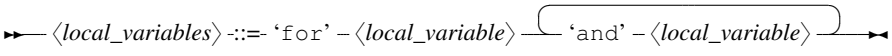
A species pattern comprises one or more species pattern elements separated by 'l'.



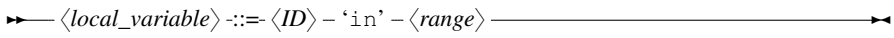
A species pattern element is a species expression, optionally followed by a quantity expression.



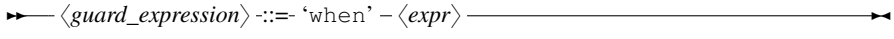
A species expression identifies a species, with optional coordinates, local variables, and a guard expression.



Local variables are defined using the 'for' keyword, with multiple variables separated by 'and'.



A local variable is defined by an identifier and a range of values it can take.



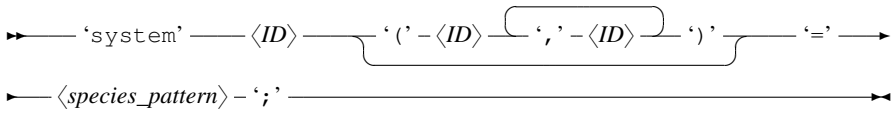
A guard expression is a condition introduced by the 'when' keyword, followed by a Boolean expression. For example:

```
rule triplication {
    A -[ %A * triplication_rate ]-> A|A|A
}

rule infection {
    S|I -[ #S * %I * k_i ]-> I|I
}

rule infection for x in [0, GRID_SIZE] and y in [0, GRID_SIZE] {
    Susceptible[x,y]|Infected[x,y] -[
        infection_rate * #Susceptible[x,y] * %Infected[x,y]
    ]-> Infected[x,y]|Infected[x,y]
}
```

System System configurations define the initial state of the population model.



For example:

```
system initial = A[50,50]<9900> | B[50,50]<100> | C[50,50]<300>;

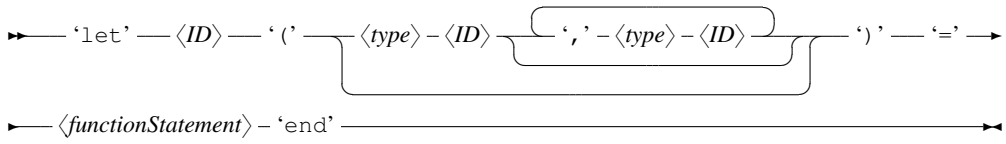
system initial_SIR = S[50,50]<9900> | I[50,50]<100> | R[50,50]<0>;
```

This formal syntax specification provides a precise definition of the structure and components of the population model specification language. It can be used as a reference for implementing parsers, creating development tools, or ensuring consistency in model definitions.

A.4 YODA Model Specification Language

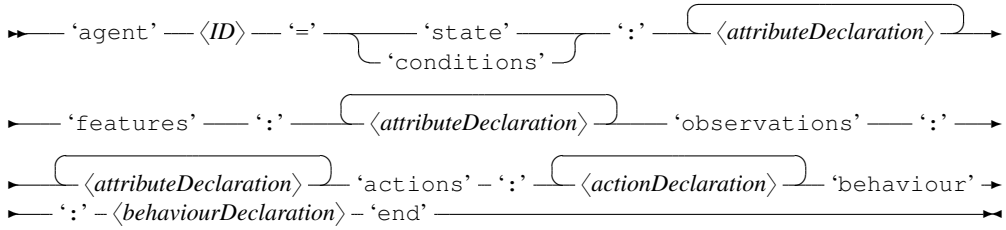
YODA is a domain-specific language designed for modelling and analysing Multi-Agent Systems (MASs). This section outlines the key components and syntax of YODA, organised into main categories.

Functions Functions in YODA allow for modular code and reusable computations. They are declared as follows:

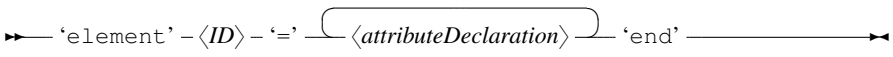


Where <functionStatement> can be a return statement, an if-then-else construct, or a let-in expression.

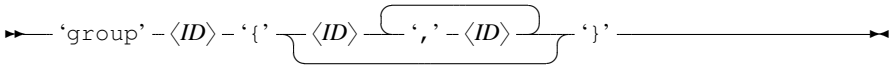
Agents Agents are the core entities in YODA models. They are defined by conditions, features, observations, actions, and behaviours.



Scene Elements Scene elements represent static objects in the environment:



Groups Groups allow for collective referencing of agents:

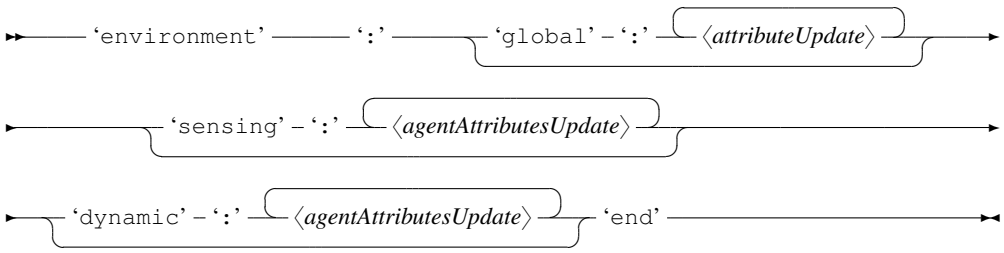


Example of an agent declaration:

```
agent Robot =
    conditions:
        int dirx = 0;
        int diry = 0;
    features:
        int x = 0;
        int y = 0;
    observations:
        bool north = false;
        bool south = false;
        bool east = false;
        bool west = false;
        bool goal = false;
    actions:
        moveNorth [ dirx <- 0; diry <- 1; ]
        moveSouth [ dirx <- 0; diry <- -1; ]
        moveEast [ dirx <- 1; diry <- 0; ]
        moveWest [ dirx <- -1; diry <- 0; ]
        stop [ dirx <- 0; diry <- 0; ]
    behaviour:
```

```
        when goal -> [ stop: 1 ]
        orwhen !north -> [ moveNorth: 1 ]
        orwhen !east -> [ moveEast: 1 ]
        orwhen !west -> [ moveWest: 1 ]
        otherwise [ stop: 1 ]
end
```

Environment The environment in YODA regulates the interactions between agents and their surroundings:

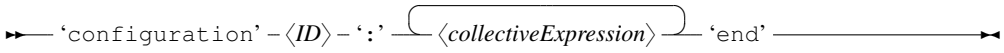


The sensing function updates agent observations, while the dynamic function updates agent features. These functions can collect data from groups of agents using group expressions.

Example:

```
environment:
    sensing:
        Robot [
            north <- any Obstacle: (posy==it.y+1)&&(posx==it.x);
            south <- any Obstacle: (posy==it.y-1)&&(posx==it.x);
            east <- any Obstacle: (posx==it.x+1)&&(posy==it.y);
            west <- any Obstacle: (posx==it.x-1)&&(posy==it.y);
            goal <- it.y==height+1;
        ]
    dynamic:
        Robot [
            x <- x + dirx;
            y <- y + diry;
        ]
end
```

System Configurations System configurations define the initial state of the model:

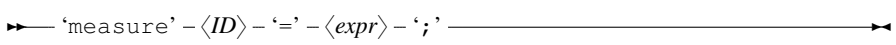


Collective expressions allow for instantiating multiple agents or scene elements, potentially using iterative or conditional constructs.

Example:

```
configuration Main:
    for i sampled distinct na time from floor(U[0,width]) do
        Robot [ x = i; y = 0; ]
    endfor
    for o sampled distinct no time from floor(U[0,width]) do
        Obstacle [ posx = o; posy = floor(U[2,height]); ]
    endfor
end
```

Measures Measures allow for the quantification and tracking of specific aspects of the model over time:



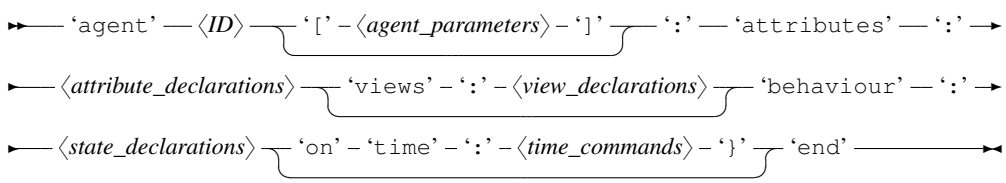
Measures can use group expressions to collect data from sets of agents.
Example:

```
measure inGoalArea = #Robot[it.goal];
measure averageX = mean Robot.x;
```

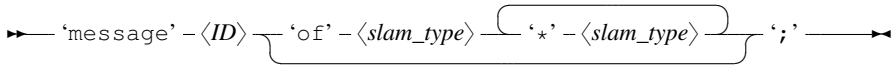
A.5 SLAM Model Specification Language

The SLAM (Stochastic Language for Agent-based Models) specification language provides a rich set of constructs for defining agent-based models. This sectionbylines the key components and syntax of SLAM.

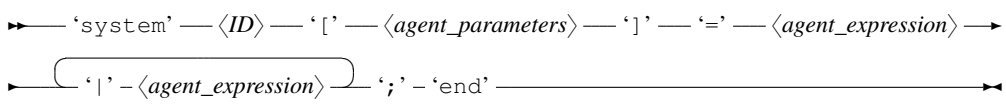
Agents Agents are the core entities in SLAM models. They are defined with attributes, views, and behaviours.



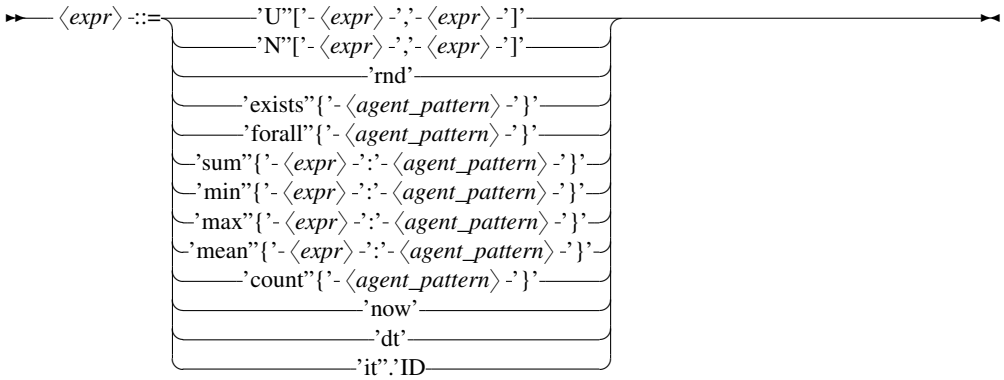
Messages Messages are used for communication between agents.



System The system declaration defines the initial configuration of the model.



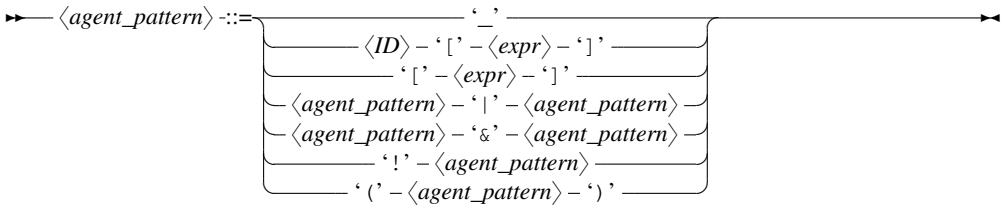
Expressions SLAM extends the common expression syntax with additional constructs for agent-based modelling. The following railroad diagram shows only these additional elements:



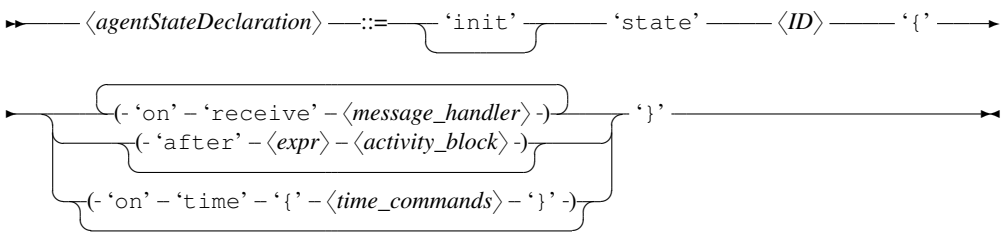
Where:

- `U[from, to]` - Uniform distribution sampling
- `N[mean, sigma]` - Normal distribution sampling
- `rnd` - Random value generation
- `exists{agentPattern}` - Agent existence check
- `forall{agentPattern}` - Universal quantification
- `count{agentPattern}` - Agent counting
- `now` - Current time reference
- `dt` - time step
- `it.attributeName` - Pattern reference:
- Aggregation operations for agents : `sum`, `min`, `max`, `mean`

Agent Patterns Agent patterns are used to select specific subsets of agents based on their properties.



State Declarations States define the behaviour of agents under different conditions.



APPENDIX B

Sibilla Shell Commands

The Sibilla shell provides a powerful command-line interface that primarily interacts with the framework's simulation and analysis capabilities. Users can configure simulations, manage model parameters, collect data, perform analyses, and synthesis tasks with minimal complexity through a set of commands. This interface bridges the gap between Sibilla's advanced computational capabilities and the practical needs of researchers and practitioners.

The shell's command structure follows established conventions from Unix-like systems, providing a familiar environment for users while incorporating domain-specific features tailored to stochastic model analysis. Each command is designed to handle a specific workflow aspect, from basic tasks such as loading models and setting parameters to complex operations such as statistical analysis and data collection.

This chapter comprehensively references all available shell commands, detailing their syntax, parameters, and typical usage patterns. For each command, we provide examples that demonstrate its application in everyday scenarios, explanations of its behaviour and any relevant considerations for its practical use. Understanding these commands is crucial to exploiting Sibilla's full potential in both interactive exploration and automated analysis workflows.

B.1 Commands

`modules`

show the list of available modules

`module <name>`

load the module a language specified by the name (e.g. `population`).

`load <filename>`

load the specification from the given file name (for example, for a `population` specification a file `file.pm`)

env or environment

shows the current assignment of parameters.

For example, the output will be:

```
> env
    par1 = 1,000000
    par2 = 0,250000
    par3 = 0,050000
```

set <name> <real>

set the given parameter identified by a string <name> to the given real value <real>.

For example:

```
> env
    par1 = 1,000000
    par2 = 0,250000
    par3 = 0,050000

> set "par1" 3.0
    Ok

> env
    par1=3,000000
    par2=0,250000
    par3=0,050000
```

please note: after you set one or some parameters, you need to set again the initial state of the model by the command `init`:

```
> set "par1" 0.5
    Ok

> set "par2" 0.1
    Ok

...

> init "initial"
    Ok
```

After the`init` command, you can now perform other operations like:

`simulate`

`fpt`

`probreach`

...

clear

reset the shell content.

reset <name>

reset the given parameter to the default value.

states

show the list of initial configurations.

For example:

```
> states
    Configurations:
    initial
    initial1
    initial2
```

init <name>

set initial state to the given value.

replica <num>

set the number of simulation replicas.

deadline <real>

set the simulation deadline.

dt <real>

set the sampling time.

To better understand what the `dt` parameter does let's look at the Figure B.1:

- `dt 0.25` means 20 points were sampled for 5 time units
- `dt 1.00` means 5 points were sampled for 5 time units
- `dt 0.25` means 2 points were sampled for 5 time units

Consequently, the greater the `dt`, the smaller the points sampled per unit of time and the greater the trajectory approximation.

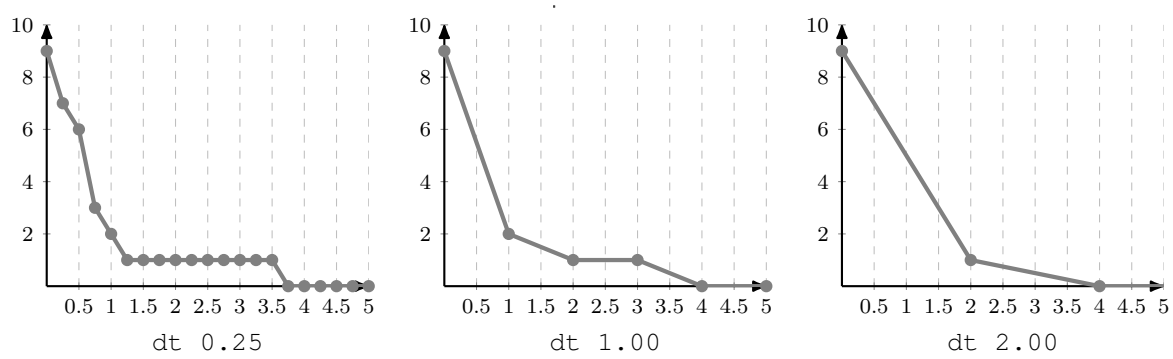


Figure B.1: *Same trajectory with a different sampling interval*

measures

show the list of available measures.

For example, referring to the code:

```
> measures
    List of available measures:
    #M1 *
    #M2 *
    #M3
    %M1
    %M2 *
    %M3
```

The measures with the $\star$ are available, while those without are not available.

add measure <name>

add the given measure to the ones collected during the simulation.

For example, referring to the code:

```
> measures
    List of available measures:
    #M1 *
    #M2 *
    #M3 *
    %M1
    %M2
    %M3
> add measure "%M1"
    Ok
> measures
    List of available measures:
    #M1 *
    #M2 *
    #M3 *
    %M1 *
    %M2
    %M3
```

remove measure <name>

remove the given measure from the ones collected during the simulation.

For example, referring to the code:

```
> measures
    List of available measures:
    #M1 *
    #M2 *
    #M3 *
    %M1
    %M2
    %M3
> remove measure "#M1"
    Ok
> measures
    List of available measures:
    #M1
    #M2 *
    #M3 *
    %M1
    %M2
    %M3
```

add all measures

add all the available measures to the simulation.

For example, referring to the code:

```
> measures
    List of available measures:
    #M1
    #M2
    #M3
    %M1
    %M2
    %M3
> add all measure
    Ok
> measures
    List of available measures:
    #M1 *
    #M2 *
    #M3 *
    %M1 *
    %M2 *
    %M3 *
```

remove all measures

remove all measures from the simulation.

For example, referring to the code:

```
> measures
    List of available measures:
    #M1 *
    #M2 *
    #M3 *
    %M1 *
    %M2 *
    %M3 *
> remove all measure
    Ok
> measures
    List of available measures:
    #M1
    #M2
    #M3
    %M1
    %M2
    %M3
```

save output <output_folder> prefix <prefix> postfix <postfix>

save the collected results in the given folder. Each measure will be saved in a csv file named <prefix><measurename><postfix>.csv. The generate file will contain three columns: time of sampled value, mean, variance and standard deviation.

For example, to save results, we will after simulate:

```
> save output ".../result" prefix "model" postfix"__"
```

after the execution of this command, six files are generated in the folder result :

```
model#M1__.csv
model#M2__.csv
```

```
model#M3__.csv
model%M1__.csv
model%M2__.csv
model%M3__.csv
```

run <script_file>

execute the script using the given file name.

Suppose that you have file `sir.pm`:

```
module "population"
load "examples/sir/sir.pm"
init "initial"
add all measures
deadline 100
dt 1.0
replica 100
simulate
save output "./results" prefix "sir" postfix "__"
```

Listing B.1: *sir.pm file*

You can run the command sequence in the script file by using the command `run` command:

```
> run ".../examples/sir.bin"
```

fpt <predicate_name>

`fpt` stands for **Frist Passage Time**(FPT). This command takes as a parameter a predicate and returns:

Tests: number of test performed.

Hits: number of time that the predicate is reached over the number of test.

Mean: mean time to reach the predicate.

SD: standard deviation of the reachability of the predicate.

MIN: the shortest time of all to reach the predicate.

Q1: first quartile.

Q2: second quartile.

Q3: third quartile.

MAX: the longest time of all to reach the predicate

For example, the output will be:

```
> fpt "allRecovered"
    First passage time allRecovered:
    Tests = 100
    Hits = 18
    Mean = 96.22748256816122
    SD = 4.978249673955105
    MIN = 84.54573300703788
    Q1 = 93.94657017428625
    Q2 = 97.9292317314421
    Q3 = 100.3644655106698
    MAX = 101.8111752310038
```

In Figure B.2 you can see those results plotted.

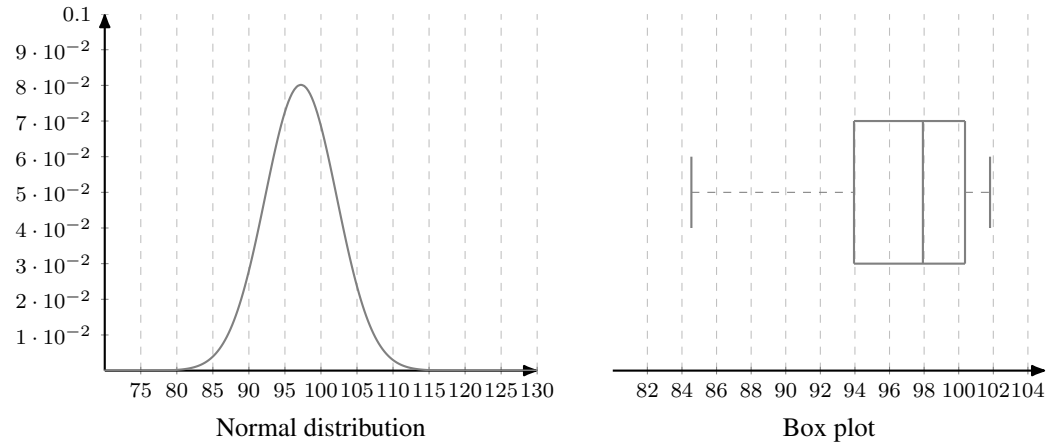


Figure B.2: *rapresentation of the data*

`probreach <predicate_name> with alpha = <real> and delta = <real>`

Returns the probability that the predicate is reached in a certain time step.

```
> probreach "allRecovered" with alpha = 0.1 and delta = 0.1
    ...
    Probability: 0.49333333333333335
```

`probreach <predicate_name_1> while <predicate_name_2> with alpha = <real> and delta = <real>`

Returns the probability that the predicate is reached in a certain time step.

```
> probreach "allRecovered" with alpha = 0.1 and delta = 0.1
    ...
    Probability: 0.49333333333333335
```

`quit <script_file>`

Terminates execution.

`run synthesis <path> [verbose]?`

`run synthesis` initiates a model synthesis process using a configuration file. The command structure consists of:

run synthesis: The base command indicating a synthesis operation.

path: Required parameter specifying the path to the YAML configuration file.

verbose: Optional flag to enable detailed output reporting.

The command produces a synthesis record containing:

Optimal Coordinates: The parameter values that achieve the best results.

Objective Function Value: The actual performance at the optimal point.

Surrogate Function Value: The surrogate model prediction at the optimal point.

When the verbose flag is included, additional information is provided:

Algorithm Configuration: Selected optimisation algorithm, surrogate model, and sampling strategy.

Optimisation Parameters: Technical settings, including sample count and training proportions.

Search Space: Parameter boundaries and constraints.

Performance Metrics: Both in-sample and out-of-sample evaluation metrics.

For example, the output will be:

```
> run synthesis "config/synthesis/optFeasibility.yaml" verbose
    Synthesis results:
    Optimal Coordinates: {x: 0.5, y: -0.3}
    Optimal Value (Objective Function): 0.85
    Optimal Value (Surrogate Function): 0.83
    Optimization Algorithm: PSO
    Surrogate Model: RandomForest
    Sampling Strategy: LHS
    Use Surrogate: true
    Is Minimization Problem: false
    Number of Samples: 1000
    Training Portion: 0.8
    Dataset: DataSet[size=1000]
    Search Space: HyperRectangle[(-1.0,1.0),(-1.0,1.0)]
    Properties: {maxIterations=100, populationSize=50}
    In-Sample Metrics: {R2=0.95, RMSE=0.03}
    Out-of-Sample Metrics: {R2=0.92, RMSE=0.04}
```

APPENDIX C

Numerical Validation Results

This appendix provides the complete numerical data from the validation study comparing Sibilla's statistical simulation approaches against PRISM's analytical solutions for the sodium-chlorine ionization reaction system. The comprehensive dataset presented here supports the quantitative validation analysis discussed in the main text and enables detailed examination of the accuracy achieved by statistical model checking techniques. The data is systematically organized in three tables corresponding to the different methodological approaches analyzed in the validation study. Table C.1 presents the complete transient probability distributions $P(\text{Na} = i, t)$ obtained from PRISM's analytical engine using continuous-time Markov chain analysis. These results serve as the definitive ground truth for validation purposes, providing exact probability values computed via symbolic model checking techniques. The analytical solutions capture the complete system dynamics from the initial deterministic state through the rapid transition phase to the approach toward equilibrium distribution. Table C.2 shows the empirical probability distributions obtained from Sibilla's individual trajectory analysis methodology. This approach collected data from 1000 independent single-replica simulations, constructing empirical probability estimates by systematically counting the frequency of occurrence for each possible sodium molecule count across all simulation runs. This methodology provides the most direct statistical comparison with PRISM's analytical probability distributions, as it generates purely empirical estimates without imposing any distributional assumptions on the underlying stochastic process. Table C.3 presents the comprehensive results from Sibilla's standard operational mode using multi-replica simulations with 1000 replicas per time point. The table includes both the primary statistical summaries (sample mean and standard deviation) computed across simulation replicas and the derived probability distributions constructed using normal approximation with continuity correction. This approach demonstrates how Sibilla's typical output format can be transformed into probability estimates suitable for direct comparison with analytical solutions. All three approaches computed probability distributions at 30 equidistant time points carefully selected over the interval $[0, 0.006]$ to capture the full temporal evolution of the chemical reaction system. The probability calculations encompass all possible sodium molecule counts $i \in 0, 1, \ldots, 10$, providing complete distributional information at each temporal sampling point. The numerical data demonstrates excellent quantitative agreement between analytical and simulation-based approaches across the entire temporal range, providing the robust quantitative foundation for the distance metrics and error analysis reported in the main validation study. The consistency of results across different methodological approaches validates both the accuracy of Sibilla's statistical simulation capabilities and the reliability of the normal approximation technique for discrete molecular population dynamics.

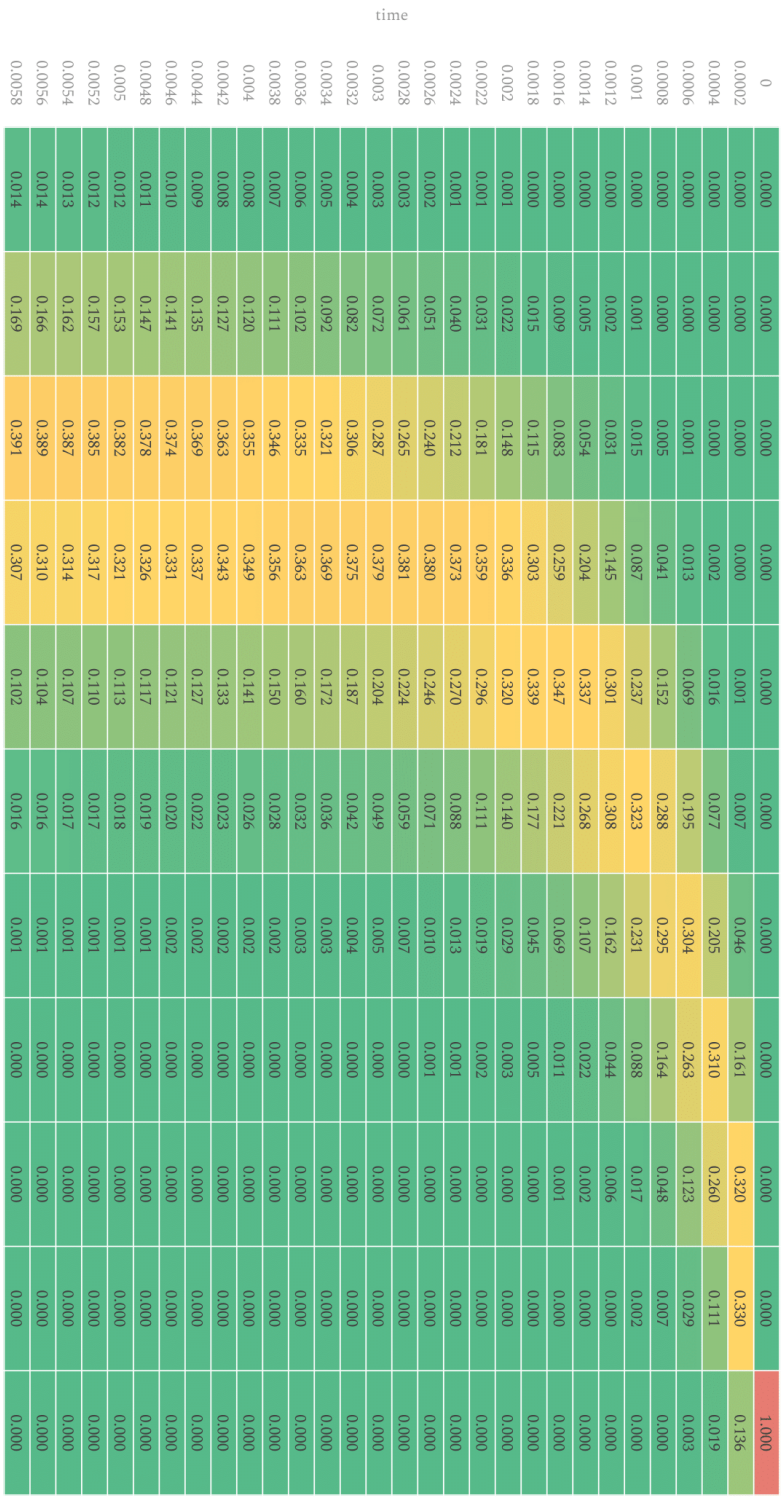
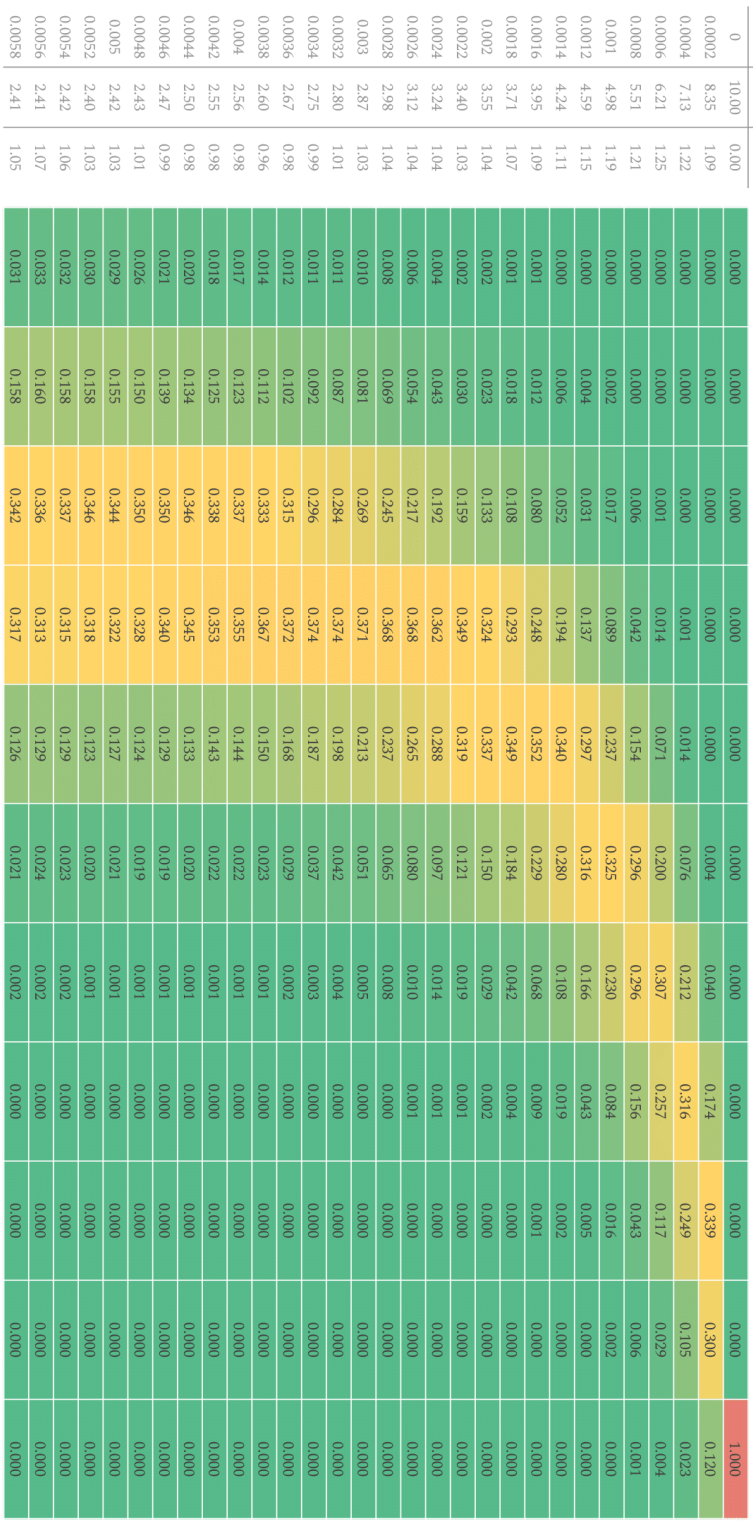


Table C.1: *PRISM Analytical Results - Complete Probability Distributions*

Probability
of there being i Na molecules at time T

time	P(Na=0)	P(Na=1)	P(Na=2)	P(Na=3)	P(Na=4)	P(Na=5)	P(Na=6)	P(Na=7)	P(Na=8)	P(Na=9)	P(Na=10)
0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
0.0002	0.000	0.000	0.000	0.000	0.001	0.014	0.051	0.157	0.295	0.338	0.144
0.0004	0.000	0.000	0.000	0.002	0.019	0.097	0.213	0.306	0.232	0.107	0.024
0.0006	0.000	0.000	0.000	0.017	0.083	0.201	0.303	0.239	0.128	0.022	0.007
0.0008	0.000	0.000	0.008	0.048	0.166	0.300	0.277	0.149	0.043	0.009	0.000
0.001	0.000	0.001	0.018	0.088	0.258	0.314	0.221	0.078	0.021	0.001	0.000
0.0012	0.000	0.002	0.029	0.160	0.315	0.306	0.136	0.040	0.011	0.001	0.000
0.0014	0.000	0.006	0.052	0.224	0.347	0.246	0.101	0.017	0.007	0.000	0.000
0.0016	0.000	0.009	0.075	0.306	0.331	0.200	0.069	0.008	0.002	0.000	0.000
0.0018	0.000	0.022	0.100	0.358	0.304	0.162	0.047	0.006	0.001	0.000	0.000
0.002	0.001	0.024	0.136	0.387	0.288	0.132	0.027	0.004	0.001	0.000	0.000
0.0022	0.001	0.037	0.172	0.375	0.292	0.097	0.022	0.004	0.000	0.000	0.000
0.0024	0.001	0.045	0.217	0.371	0.267	0.082	0.017	0.000	0.000	0.000	0.000
0.0026	0.003	0.055	0.227	0.391	0.248	0.066	0.010	0.000	0.000	0.000	0.000
0.0028	0.005	0.051	0.277	0.377	0.233	0.051	0.006	0.000	0.000	0.000	0.000
0.003	0.005	0.070	0.288	0.390	0.191	0.051	0.005	0.000	0.000	0.000	0.000
0.0032	0.006	0.089	0.291	0.376	0.189	0.043	0.005	0.001	0.000	0.000	0.000
0.0034	0.010	0.103	0.307	0.356	0.179	0.040	0.005	0.000	0.000	0.000	0.000
0.0036	0.009	0.101	0.340	0.354	0.158	0.032	0.006	0.000	0.000	0.000	0.000
0.0038	0.010	0.114	0.346	0.343	0.145	0.038	0.004	0.000	0.000	0.000	0.000
0.004	0.012	0.120	0.366	0.339	0.128	0.030	0.005	0.000	0.000	0.000	0.000
0.0042	0.011	0.141	0.366	0.326	0.129	0.021	0.006	0.000	0.000	0.000	0.000
0.0044	0.009	0.150	0.370	0.307	0.141	0.019	0.004	0.000	0.000	0.000	0.000
0.0046	0.011	0.156	0.382	0.302	0.126	0.021	0.002	0.000	0.000	0.000	0.000
0.0048	0.013	0.170	0.372	0.309	0.112	0.021	0.002	0.001	0.000	0.000	0.000
0.005	0.016	0.171	0.379	0.307	0.111	0.013	0.003	0.000	0.000	0.000	0.000
0.0052	0.018	0.172	0.384	0.306	0.102	0.016	0.002	0.000	0.000	0.000	0.000
0.0054	0.017	0.178	0.386	0.304	0.094	0.021	0.000	0.000	0.000	0.000	0.000
0.0056	0.012	0.185	0.388	0.306	0.090	0.018	0.001	0.000	0.000	0.000	0.000
0.0058	0.013	0.179	0.400	0.309	0.085	0.014	0.000	0.000	0.000	0.000	0.000

Table C.2: *Sibilla Approach 1 Results - Empirical Probability Distributions*



Time	Mean	SD	P(Na=0)	P(Na=1)	P(Na=2)	P(Na=3)	P(Na=4)	P(Na=5)	P(Na=6)	P(Na=7)	P(Na=8)	P(Na=9)	P(Na=10)
0	10.00	0.00	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
0.0002	8.35	1.09	0.000	0.000	0.000	0.000	0.000	0.004	0.040	0.174	0.339	0.300	0.120
0.0004	7.13	1.22	0.000	0.000	0.000	0.001	0.014	0.076	0.212	0.316	0.249	0.105	0.023
0.0006	6.21	1.25	0.000	0.000	0.001	0.014	0.071	0.200	0.307	0.257	0.117	0.029	0.004
0.0008	5.51	1.21	0.000	0.000	0.006	0.042	0.154	0.296	0.296	0.156	0.043	0.006	0.001
0.001	4.98	1.19	0.000	0.002	0.017	0.089	0.237	0.325	0.230	0.084	0.016	0.002	0.000
0.0012	4.59	1.15	0.000	0.004	0.031	0.137	0.297	0.316	0.166	0.043	0.005	0.000	0.000
0.0014	4.24	1.11	0.000	0.006	0.052	0.194	0.340	0.280	0.108	0.019	0.002	0.000	0.000
0.0016	3.95	1.09	0.001	0.012	0.080	0.248	0.352	0.229	0.068	0.009	0.001	0.000	0.000
0.0018	3.71	1.07	0.001	0.018	0.108	0.293	0.349	0.184	0.042	0.004	0.000	0.000	0.000
0.002	3.55	1.04	0.002	0.023	0.133	0.324	0.337	0.150	0.029	0.002	0.000	0.000	0.000
0.0022	3.40	1.03	0.002	0.030	0.159	0.349	0.319	0.121	0.019	0.001	0.000	0.000	0.000
0.0024	3.24	1.04	0.004	0.043	0.192	0.362	0.288	0.097	0.014	0.001	0.000	0.000	0.000
0.0026	3.12	1.04	0.006	0.054	0.217	0.368	0.265	0.080	0.010	0.001	0.000	0.000	0.000
0.0028	2.98	1.04	0.008	0.069	0.245	0.368	0.237	0.065	0.008	0.000	0.000	0.000	0.000
0.003	2.87	1.03	0.010	0.081	0.269	0.371	0.213	0.051	0.005	0.000	0.000	0.000	0.000
0.0032	2.80	1.01	0.011	0.087	0.284	0.374	0.198	0.042	0.004	0.000	0.000	0.000	0.000
0.0034	2.75	0.99	0.011	0.092	0.296	0.374	0.187	0.037	0.003	0.000	0.000	0.000	0.000
0.0036	2.67	0.98	0.012	0.102	0.315	0.372	0.168	0.029	0.002	0.000	0.000	0.000	0.000
0.0038	2.60	0.96	0.014	0.112	0.333	0.367	0.150	0.023	0.001	0.000	0.000	0.000	0.000
0.004	2.56	0.98	0.017	0.123	0.337	0.355	0.144	0.022	0.001	0.000	0.000	0.000	0.000
0.0042	2.55	0.98	0.018	0.125	0.338	0.353	0.143	0.022	0.001	0.000	0.000	0.000	0.000
0.0044	2.50	0.98	0.020	0.134	0.346	0.345	0.133	0.020	0.001	0.000	0.000	0.000	0.000
0.0046	2.47	0.99	0.021	0.139	0.350	0.340	0.129	0.019	0.001	0.000	0.000	0.000	0.000
0.0048	2.43	1.01	0.026	0.150	0.350	0.328	0.124	0.019	0.001	0.000	0.000	0.000	0.000
0.005	2.42	1.03	0.029	0.155	0.344	0.322	0.127	0.021	0.001	0.000	0.000	0.000	0.000
0.0052	2.40	1.03	0.030	0.158	0.346	0.318	0.123	0.020	0.001	0.000	0.000	0.000	0.000
0.0054	2.42	1.06	0.032	0.158	0.337	0.315	0.129	0.023	0.002	0.000	0.000	0.000	0.000
0.0056	2.41	1.07	0.033	0.160	0.336	0.313	0.129	0.024	0.002	0.000	0.000	0.000	0.000
0.0058	2.41	1.05	0.031	0.158	0.342	0.317	0.126	0.021	0.002	0.000	0.000	0.000	0.000

Table C.3: *Sibilla Approach 2 Results - Statistical Summaries and Normal Approximations*

APPENDIX D

Case Study Specifications

This appendix provides detailed specifications YAML for each case study discussed in Chapter 1 6. These specifications define the parameter synthesis problems, model configurations, and evaluation metrics used in our experimental analysis. Each specification follows a structured format that includes the synthesis strategy, the definition of the search space, the sampling method, the configuration of the surrogate model, and the task-specific parameters.

D.1 SIR Model Specifications

The SIR model case study explores two distinct parameter synthesis scenarios: the first specification (Listing D.1) addresses the optimisation of infection and recovery rates using qualitative evaluation metrics. This specification defines a two-dimensional search space for the kinetic parameters k_i and k_r, employing Latin Hypercube Sampling with 100 samples to explore the parameter space efficiently. The population module is configured with initial conditions of 90 Susceptible and 10 Infected individuals, and the desired system properties are expressed through STL formulae that capture the temporal evolution of the infection.

The second specification (Listing D.2) focuses on synthesising temporal parameters with fixed kinetic rates. This configuration maintains the same basic model structure but shifts the optimisation focus to temporal parameters a and b, which define time windows for analysing infection patterns. The specification employs quantitative evaluation metrics to assess the robustness of temporal properties, providing a different perspective on the system's behaviour.

```
synthesisStrategy:
  searchSpace:
  - parameterName: "k_i"
    lowerBound: 0.005
    upperBound: 0.3
  - parameterName: "k_r"
    lowerBound: 0.005
    upperBound: 0.2
  sampling:
    name: "lhs"
    datasetSize: 100
  surrogate:
    name: "ID_SURROGATE"
  optimization:
    name: "ID_OPTIMIZATION"
synthesisTask:
  type: "optimalFeasibility"
  taskSpecs:
      objective: "maximize"
      evaluation: "qualitative"
      model:
          module: "population"
          initialConfiguration: "initial"
          modelSpecification: |
              param k_i = 0.05;
              param k_r = 0.05;

              species S;
              species I;
```

```
30              species R;
31
32              rule infection {
33                  S|I -[ #S * %I * k_i ]-> I|I
34              }
35
36              rule recovered {
37                  I -[ #I * k_r ]-> R
38              }
39              system initial = S<90>|I<10>|R<0>;
40      formulae: |
41          measure #I
42          formula formula_id  : ( \E[100,120][ #I == 0] )&& (\G[0,100][ #I > 0 ]) endformula
43
```

Listing D.1: *YAML specification for the first SIR model case study, defining the parameter synthesis problem for infection and recovery rates optimisation with qualitative evaluation metric*

```
1  synthesisStrategy:
2    searchSpace:
3    - parameterName: "a"
4      lowerBound: 0.0
5      upperBound: 50.0
6    - parameterName: "b"
7      lowerBound: 0.001
8      upperBound: 50.0
9    sampling:
10     name: "lhs"
11     datasetSize: 100
12   surrogate:
13     name: "ID_SURROGATE"
14   optimization:
15     name: "ID_OPTIMIZATION"
16 synthesisTask:
17   type: "optimalFeasibility"
18   taskSpecs:
19       objective: "maximize"
20       evaluation: "quantitative"
21       model:
22           module: "population"
23           initialConfiguration: "initial"
24           modelSpecification: |
25               param k_i = 0.265;
26               param k_r = 0.05;
27
28               species S;
29               species I;
30               species R;
31
32               rule infection {
33                   S|I -[ #S * %I * k_i ]-> I|I
34               }
35
36               rule recovered {
37                   I -[ #I * k_r ]-> R
38               }
39               system initial = S<90>|I<10>|R<0>;
40       formulae: |
41           measure %I
42           formula formula_id [a=0,b=100] : ( \G[a,(a+b)][ %I > 0.3]) endformula
43
```

Listing D.2: *YAML specification for the second SIR model case study, focusing on temporal parameter synthesis with fixed kinetic rates and quantitative evaluation metrics"*

D.2 Repressilator Model Specification

The Repressilator specification (Listing D.3) details the parameter synthesis problem for a synthetic genetic oscillator. This specification focusses on a two-dimensional search space exploring the half-lives of mRNA and proteins (τ_{mRNA}, τ_{prot}). The complexity of the genetic circuit is reflected in the model configuration, which incorporates Hill functions for protein production dynamics and detailed reaction rules that govern transcription, translation, and degradation processes. The sampling strategy employs 300 points 5 simulation replicas.

```
1  synthesisStrategy:
2    searchSpace:
3    - parameterName: "tau_mRNA"
4      lowerBound: 1.0
5      upperBound: 10.0
6    - parameterName: "tau_prot"
7      lowerBound: 1.0
8      upperBound: 10.0
9    sampling:
10     name: "lhs"
11     datasetSize: 300
12 synthesisTask:
13   type: "optimalFeasibility"
14   simulationSetting:
15       replica: 5
16   taskSpecs:
17       objective: "maximize"
18       evaluation: "qualitative"
19       model:
20           module: "population"
```

```
21          initialConfiguration: "initial"
22          modelSpecification: |
23              param n = 2;          /* Hill coefficient n */
24              param KM = 40;        /* K_M*/
25              param tau_mRNA = 2;  /* mRNA half life */
26              param tau_prot = 10; /* protein half life */
27              param ps_a = 0.5;     /* promotor strength (repressed) ( tps_repr ) */
28              param ps_0 = 0.0005; /* promotor strength (full) ( tps_active ) */
29
30              const ln2 = 0.69314718056;
31              const beta = 0.2;
32              const alpha0 = 0.2164;
33              const alpha = 216.404;
34              const eff = 20;
35              const t_ave =  tau_mRNA / ln2;   /* average mRNA lifetime */
36              const kd_mRNA = ln2 / tau_prot;  /* mRNA decay rate */
37              const kd_prot = ln2 / tau_mRNA;  /* protein decay rate  */
38              const k_tl = eff / t_ave;        /* translation rate  */
39              const a_tr = (ps_a -ps_0)*60;    /* transcription rate  */
40              const a0_tr = ps_0 * 60;         /* transcription rate (repressed)   */
41
42              const startY = 20; /* Initial number of Y agents */
43              const startVOID = 1; /* Initial number of VOID agents */
44
45              species VOID;  /* To represent nothingness, fictitious species are created */
46
47              species PX; /*  protein produced by X */
48              species PY; /*  protein produced by Y */
49              species PZ; /*  protein produced by Z */
50              species X; /*  mRNA X (LacI) */
51              species Y; /*  mRNA Y (TetR) */
52              species Z; /*  mRNA Z (CI)*/
53
54
55              rule degradation_of_X_transcripts {
56                  X|VOID -[ kd_mRNA * #X ]-> VOID
57              }
58
59              rule degradation_of_Y_transcripts {
60                  Y|VOID -[ kd_mRNA * #Y ]-> VOID
61              }
62
63              rule degradation_of_Z_transcripts {
64                  Z|VOID -[ kd_mRNA * #Z ]-> VOID
65              }
66
67              rule translation_of_X {
68                  VOID -[ k_tl * #X ]-> VOID|PX
69              }
70
71              rule translation_of_Y {
72                  VOID -[ k_tl * #Y ]-> VOID|PY
73              }
74
75              rule translation_of_Z {
76                  VOID -[ k_tl * #Z ]-> VOID|PZ
77              }
78
79              rule degradation_of_X {
80                  PX|VOID -[ kd_prot * #PX ]-> VOID
81              }
82
83              rule degradation_of_Y {
84                  PY|VOID -[ kd_prot * #PY ]-> VOID
85              }
86
87              rule degradation_of_Z {
88                  PZ|VOID -[ kd_prot * #PZ ]-> VOID
89              }
90
91              rule transcription_of_X {
92                  VOID -[ a0_tr + ( ( a_tr * KM^(n) )/( KM^n + #PZ^n ) ) ]-> VOID|X
93              }
94
95              rule transcription_of_Y {
96                  VOID -[ a0_tr + ( ( a_tr * KM^(n) )/( KM^n + #PX^n ) ) ]-> VOID|Y
97              }
98
99              rule transcription_of_Z {
100                 VOID -[ a0_tr + ( ( a_tr * KM^(n) )/( KM^n + #PY^n ) ) ]-> VOID|Z
101             }
102
103             system initial = Y < startY >|VOID < startVOID >;
104     formulae: |
105         measure #PZ
106         formula f_id [kT = 800, kB = 500 , t = 100, tW = 10, tE = 20] :  \E [0,t] ( \G[tE,tW+tE]([#PZ > kB] && [#PZ < kT]) && \E
      [0,tE][#PZ < kB]  && \E [tW+tE,tE+tW+tE][#PZ < kB] )  endformula
107
```

Listing D.3: *YAML specification for the Repressilator model case study, defining parameter synthesis for mRNA and protein half-life optimisation*

D.3 Three-State Protocol Specification

The Three-State Protocol specification (Listing D.4) presents a comparative analysis framework examining the relationship between an original consensus model and its spatially-structured variant. The specification defines a six-dimensional search space encompassing node-specific red fractions and malicious agent proportions. Particle Swarm Optimisation handles the constrained optimisation problem, respecting bounds on the distribution of agents across the network. The variant model introduces a ring topology, necessitating careful specification of inter-node interactions and local consensus dynamics. The evaluation framework includes multiple STL formulae that probe different aspects of consensus formation, with a Euclidean distance metric that quantifies behavioural differences between model variants.

```
1
2  synthesisStrategy:
3    searchSpace:
4      - parameterName: "redFractionIn0"
5        lowerBound: 0.1
6        upperBound: 1.0
7      - parameterName: "redFractionIn1"
8        lowerBound: 0.1
9        upperBound: 1.0
10     - parameterName: "redFractionIn2"
11       lowerBound: 0.1
12       upperBound: 1.0
13     - parameterName: "redFractionIn3"
14       lowerBound: 0.1
15       upperBound: 1.0
16     - parameterName: "redFractionIn4"
17       lowerBound: 0.1
18       upperBound: 1.0
19     - parameterName: "malFractionForEach"
20       lowerBound: 0.0
21       upperBound: 1.0
22
23   optimization:
24     name: "pso"
25     constraints:
26       - constraint: "redFractionIn0 + redFractionIn1 + redFractionIn2 +redFractionIn3 +redFractionIn4 <= 3.5"
27       - constraint: "redFractionIn0 + redFractionIn1 + redFractionIn2 +redFractionIn3 +redFractionIn4 >= 1.5"
28     properties:
29       - name: "iteration"
30         value: "1000"
31
32   sampling:
33     name: "lhs"
34     datasetSize: 500
35
36
37
38 synthesisTask:
39   type: "comparative"
40
41   simulationSetting:
42     replica: 20
43
44   taskSpecs:
45       objective: "diverge"
46
47       distanceType: "euclidian"
48
49       modelOriginal:
50
51         module: "population"
52
53         initialConfiguration: "initial"
54
55         modelSpecification: |
56           species RED;
57           species BLUE;
58           species UNC;
59           species MAL;
60
61           const persuasion_rate = 1.0;
62           param becoming_malevolent = 0.1;
63
64           rule UNC_to_BLUE {
65             UNC-[ (%MAL>=0.5? 0:1) * #UNC * persuasion_rate * %BLUE ]-> BLUE
66           }
67
68           rule UNC_to_RED {
69             UNC -[ (%MAL>=0.5? 0:1) * #UNC * persuasion_rate * %RED ]-> RED
70           }
71
72           rule UNC_to_MAL  {
73             UNC-[#UNC * persuasion_rate * %MAL ]-> MAL
74           }
75
76           rule BLUE_to_UNC {
77             BLUE -[ (%MAL>=0.5? 0:1) * #BLUE * persuasion_rate * %RED ]-> UNC
78           }
79
80           rule RED_to_UNC {
81             RED -[ (%MAL>=0.5? 0:1) * #RED * persuasion_rate * %BLUE ]-> UNC
82           }
83
84           measure R = %RED;
85           measure B = %BLUE;
86           measure U = %UNC;
87           measure M = %MAL;
88
89           system initial = RED<10>|BLUE<10>|UNC<75>|MAL<5>;
```

```
90
91
92
93
94      modelVariant:
95        module: "population"
96        initialConfiguration: "initial"
97        modelSpecification: |
98          const K = 5;       /* length of the ring */
99
100         species RED of [0,K];
101         species BLUE of [0,K];
102         species UNC of [0,K];
103         species MAL of [0,K];
104
105         param persuasion_rate = 3.0; /* persuasion rate */
106         param becoming_malevolent = 0.1;
107
108         param totalRedAndBlue = 20;
109         param totalUncAndMal = 80;
110
111         param redFractionIn0 = 0.5;
112         param redFractionIn1 = 0.5;
113         param redFractionIn2 = 0.5;
114         param redFractionIn3 = 0.5;
115         param redFractionIn4 = 0.5;
116
117         const redQuantityIn0 = redFractionIn0 * totalRedAndBlue / K;
118         const redQuantityIn1 = redFractionIn1 * totalRedAndBlue / K;
119         const redQuantityIn2 = redFractionIn2 * totalRedAndBlue / K;
120         const redQuantityIn3 = redFractionIn3 * totalRedAndBlue / K;
121         const redQuantityIn4 = redFractionIn4 * totalRedAndBlue / K;
122
123         const blueQuantityIn0 = (1.0-redFractionIn0) * totalRedAndBlue / K;
124         const blueQuantityIn1 = (1.0-redFractionIn1) * totalRedAndBlue / K;
125         const blueQuantityIn2 = (1.0-redFractionIn2) * totalRedAndBlue / K;
126         const blueQuantityIn3 = (1.0-redFractionIn3) * totalRedAndBlue / K;
127         const blueQuantityIn4 = (1.0-redFractionIn4) * totalRedAndBlue / K;
128
129         param malFractionForEach = 0.0;
130
131         const uncQuantityInEach = (1.0-malFractionForEach) * totalUncAndMal /K;
132         const malQuantityInEach = malFractionForEach * totalUncAndMal /K;
133
134
135         rule UNC_to_BLUE for i in [0,K] {
136           UNC[i]-[ (%MAL[i]>=0.5? 0:1) * #UNC[i] * persuasion_rate * (%BLUE[i] + %BLUE[(i+1)%K] + %BLUE[(i-1+K)%K])/3 ]-> BLUE[i]
137         }
138
139         rule UNC_to_RED for i in [0,K] {
140           UNC[i]-[(%MAL[i]>=0.5? 0:1) * #UNC[i] * persuasion_rate * (%RED[i] + %RED[(i+1)%K] + %RED[(i-1+K)%K])/3 ]-> RED[i]
141         }
142
143         rule UNC_to_MAL for i in [0,K] {
144           UNC[i]-[#UNC[i] * persuasion_rate * %MAL[i] ]-> MAL[i]
145         }
146
147         rule BLUE_to_UNC for i in [0,K] {
148           BLUE[i]-[ (%MAL[i]>=0.5? 0:1) * #BLUE[i] * persuasion_rate * (%RED[i] + %RED[(i+1)%K] + %RED[(i-1+K)%K])/3 ]-> UNC[i]
149         }
150
151         rule RED_to_UNC for i in [0,K] {
152           RED[i]-[ (%MAL[i]>=0.5? 0:1) * #RED[i] * persuasion_rate * (%BLUE[i] + %BLUE[(i+1)%K] + %BLUE[(i-1+K)%K])/3 ]-> UNC[i]
153         }
154
155         measure R = %RED[0] + %RED[1] + %RED[2] + %RED[3] + %RED[4];
156         measure B = %BLUE[0] + %BLUE[1] + %BLUE[2] + %BLUE[3] + %BLUE[4];
157         measure U = %UNC[0] + %UNC[1] + %UNC[2] + %UNC[3] + %UNC[4];
158         measure M = %MAL[0] + %MAL[1] + %MAL[2] + %MAL[3] + %MAL[4];
159
160         system initial = RED[0]<redQuantityIn0>|RED[1]<redQuantityIn1>|RED[2]<redQuantityIn2>|RED[3]<redQuantityIn3>|RED[4]<
        redQuantityIn4>|BLUE[0]<blueQuantityIn0>|BLUE[1]<blueQuantityIn1>|BLUE[2]<blueQuantityIn2>|BLUE[3]<blueQuantityIn3>|BLUE[4]<
        blueQuantityIn4>|UNC[0]<uncQuantityInEach>|UNC[1]<uncQuantityInEach>|UNC[2]<uncQuantityInEach>|UNC[3]<uncQuantityInEach>|UNC
        [4]<uncQuantityInEach>|MAL[0]<malQuantityInEach>|MAL[1]<malQuantityInEach>|MAL[2]<malQuantityInEach>|MAL[3]<malQuantityInEach
        >|MAL[4]<malQuantityInEach>;
161
162
163
164
165     formulae: |
166       measure R
167       measure B
168       measure M
169       formula formula_stability [t=25] : \E[0,t][ R >= 1.0 ]   endformula
170       formula formula_coherence [t=25] : ([ R > 3 * B ]&&(\E[0,t][ B <= 0.0])) endformula
171       formula formula_red_pres [t=25] : (\G[0,t][ R >= 0.5]) endformula
172       formula formula_mal_pres [t=25] : ((\E[0, 1/4 * t]([M >= 0.01]))<->(\G[3/4 * t,t]([M >= 0.25] && [M <= 0.75]))) endformula
173
```

Listing D.4: *YAML specification for the Three-State Protocol (TSP) case study, defining comparative analysis between original and variant models*

These specifications represent more than mere configuration files; they encode our experimental methodology and enable a precise reproduction of the results. Parameterisation and the definition of objectives bridge the gap between theoretical models and practical implementation, providing a foundation for the systematic analysis of complex biological and social systems.

APPENDIX E

Web-Based Interface Prototypes

This appendix presents the current prototype implementations of web-based interfaces for the Sibilla framework. These prototypes are automatically generated HTML reports, similar to test result outputs commonly produced in software development environments, and demonstrate the potential for comprehensive web-based analysis and visualisation tools.

E.1 Dynamic Measurement Dashboard

The Dynamic Measurement Dashboard (Figure E.1) provides an interactive interface for visualising simulation results and statistical measurements. This prototype demonstrates how users can explore simulation data through a comprehensive web-based interface.

E.1.1 Key Features

The dashboard prototype incorporates several key visualisation and interaction features:

- **Measurement Selection Controls**: Interactive checkboxes allow users to selectively display different measurement types (#I, #R, #S, %I, %R, %S) with corresponding statistical indicators (Mean, Standard Deviation, Confidence Interval).
- **Real-time Plot Updates**: The main visualisation area displays time-series data with confidence intervals, automatically updating as users modify their selection criteria.
- **Statistical Summary**: A lower panel provides detailed statistical information for selected measurements, including mean values and standard deviation calculations.
- **Responsive Design**: The interface adapts to different screen sizes and provides intuitive navigation through complex simulation datasets.

E.1.2 Technical Implementation

The prototype utilises modern web technologies including interactive JavaScript plotting libraries and responsive CSS frameworks. The interface demonstrates how simulation results from Sibilla can be seamlessly integrated into web-based analysis workflows, providing researchers with immediate visual feedback on their simulation experiments.

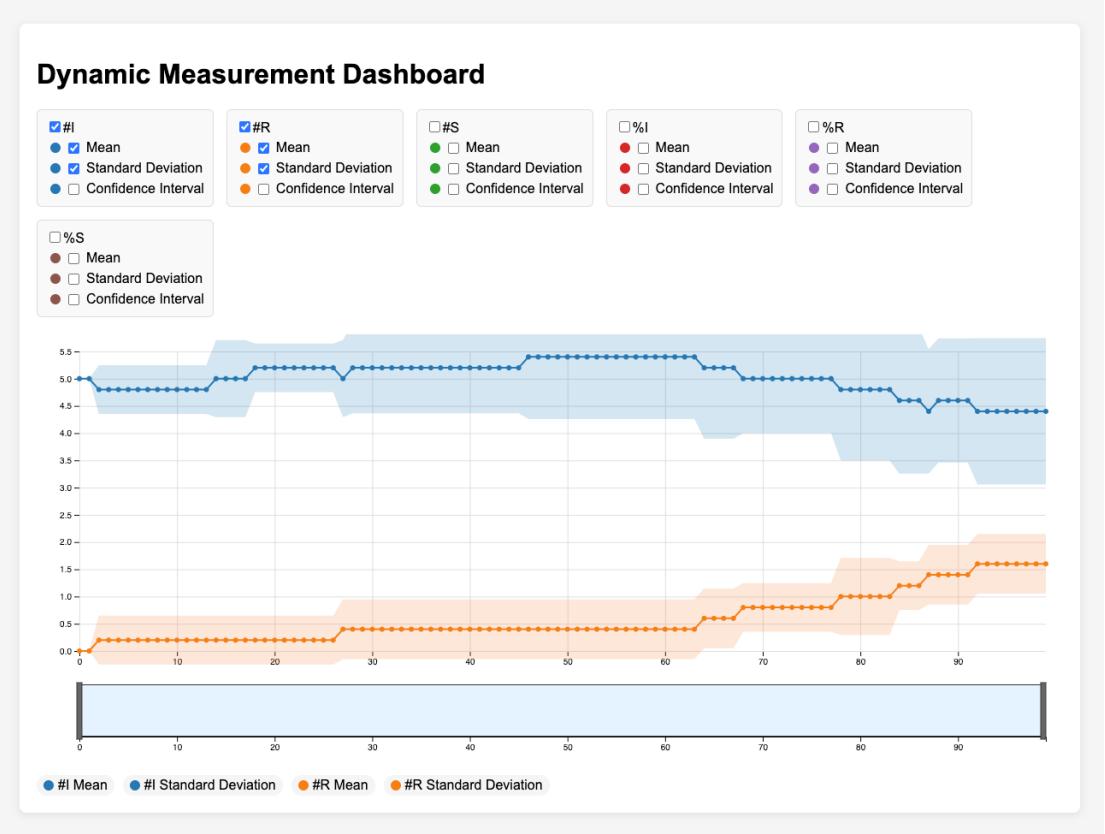


Figure E.1: *Dynamic Measurement Dashboard prototype showing interactive visualisation of simulation results with multiple measurement types and statistical indicators.*

E.2 Synthesis Optimisation Dashboard

The Synthesis Optimisation Dashboard (Figure E.2) presents a comprehensive interface for surrogate-based optimisation analysis. This prototype showcases how complex optimisation results can be presented in an accessible, interactive format.

E.2.1 Dashboard Components

The synthesis dashboard prototype organises optimisation results into several key sections:

- **Optimal Solution Panel**: Displays the best parameter configuration found during optimisation, including both objective function and surrogate function values at the optimal coordinates.
- **Search Space Summary**: Provides a tabular view of the parameter space, showing minimum and maximum bounds for each dimension alongside the optimal values discovered.
- **Optimisation Settings**: Details the configuration used for the optimisation process, including algorithm selection, surrogate model type, sampling strategy, and other key parameters.
- **Model Performance Metrics**: Presents comprehensive evaluation of surrogate model quality through both in-sample and out-of-sample performance indicators, including R^2 values and RMSE measurements.
- **Parallel Coordinates Visualisation**: Offers an interactive parallel coordinates plot that enables exploration of the parameter space and objective function values across different optimisation runs.

E.2.2 Visualisation Capabilities

The parallel coordinates visualisation represents a particularly powerful feature of the synthesis dashboard. This interactive plot allows users to:

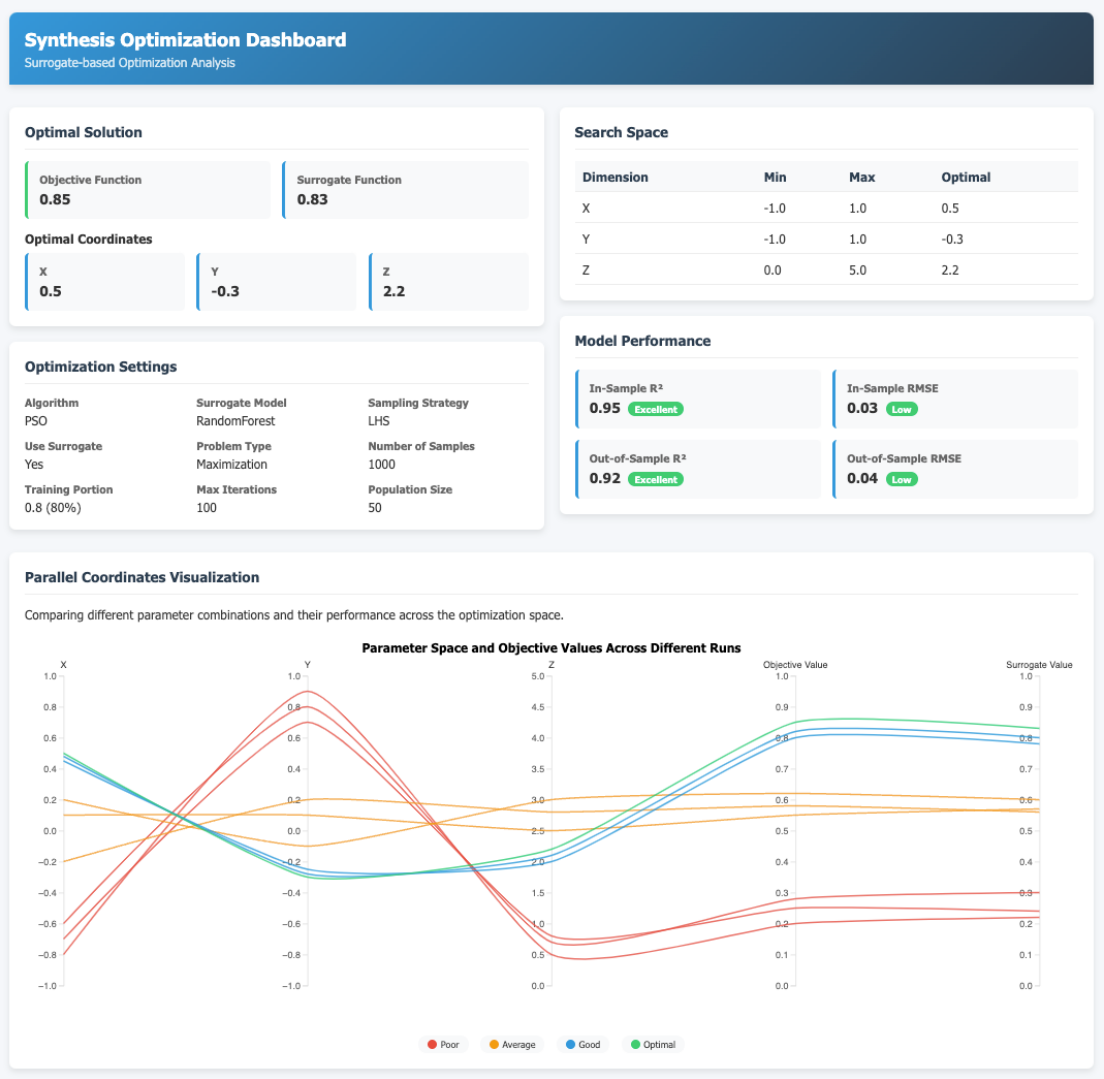


Figure E.2: *Synthesis Optimisation Dashboard prototype displaying comprehensive optimisation results including parameter space analysis, model performance metrics, and parallel coordinates visualisation.*

- Explore relationships between multiple parameters simultaneously
- Identify parameter combinations that lead to optimal performance
- Understand the sensitivity of the objective function to different parameters
- Compare the quality of different solutions through colour-coded performance indicators

The visualisation uses a colour scheme that categorises solutions into performance levels (Poor, Average, Good, Optimal), enabling rapid identification of high-quality parameter regions.

E.2.3 Future Development Directions

These prototypes serve as proof-of-concept implementations that demonstrate the feasibility and value of web-based interfaces for the Sibilla framework. Future development would focus on:

- Integration with the core Sibilla runtime to enable real-time analysis
- Enhanced interactivity including parameter modification and re-analysis capabilities

- Export functionality for high-quality visualisations and detailed reports
- Collaborative features enabling sharing and discussion of analysis results
- Template systems for common analysis workflows and model types

The current prototypes validate the technical approach and user experience design, providing a foundation for the development of production-ready web-based analysis tools that would significantly enhance the accessibility and usability of the Sibilla framework.