

Blakes, Jonathan (2013) Infobiotics : computer-aided synthetic systems biology. PhD thesis, University of Nottingham.

Access from the University of Nottingham repository:

http://eprints.nottingham.ac.uk/13434/1/Jonathan_Blakes-PhD_thesis-bindable_20130605.pdf

Copyright and reuse:

The Nottingham ePrints service makes this work by researchers of the University of Nottingham available open access under the following conditions.

- Copyright and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners.
- To the extent reasonable and practicable the material made available in Nottingham ePrints has been checked for eligibility before being made available.
- Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.
- Quotations or similar reproductions must be sufficiently acknowledged.

Please see our full end user licence at:

http://eprints.nottingham.ac.uk/end_user_agreement.pdf

A note on versions:

The version presented here may differ from the published version or from the version of record. If you wish to cite this item you are advised to consult the publisher's version. Please see the repository url above for details on accessing the published version and note that access may require a subscription.

For more information, please contact eprints@nottingham.ac.uk

Infobiotics: Computer-Aided Synthetic Systems Biology

Jonathan Blakes BSc (Hons) MSc

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

May 2012

Abstract

Until very recently Systems Biology has, despite its stated goals, been too reductive in terms of the models being constructed and the methods used have been, on the one hand, unsuited for large scale adoption or integration of knowledge across scales, and on the other hand, too fragmented. The thesis of this dissertation is that better computational languages and seamlessly integrated tools are required by systems and synthetic biologists to enable them to meet the significant challenges involved in understanding life as it is, and by designing, modelling and manufacturing novel organisms, to understand life as it could be. We call this goal, where everything necessary to conduct model-driven investigations of cellular circuitry and emergent effects in populations of cells is available without significant context-switching, “one-pot” *in silico* synthetic systems biology in analogy to “one-pot” chemistry and “one-pot” biology. Our strategy is to increase the understandability and reusability of models and experiments, thereby avoiding unnecessary duplication of effort, with practical gains in the efficiency of delivering usable prototype models and systems. Key to this endeavour are graphical interfaces that assists novice users by hiding complexity of the underlying tools and limiting choices to only what is appropriate and useful, thus ensuring that the results of *in silico* experiments are consistent, comparable and reproducible.

This dissertation describes the conception, software engineering and use of two novel software platforms for systems and synthetic biology: the Infobiotics Workbench for modelling, *in silico* experimentation and analysis of multi-cellular biological systems; and DNA Library Designer with the DNALD language for the compact programmatic specification of combinatorial DNA libraries, as the first stage of a DNA synthesis pipeline, enabling methodical exploration biological problem spaces. Infobiotics models are formalised as Lattice Population P systems, a novel framework for the specification of spatially-discrete and multi-compartmental rule-based models, imbued with a stochastic execution semantics. This framework was developed to meet the needs of real systems biology problems: hormone transport and signalling in the root of *Arabidopsis thaliana*, and quorum sensing in the pathogenic bacterium *Pseudomonas aeruginosa*. Our tools have also been used to

prototype a novel synthetic biological system for pattern formation, that has been successfully implemented *in vitro*.

Taken together these novel software platforms provide a complete toolchain, from design to wet-lab implementation, of synthetic biological circuits, enabling a step change in the scale of biological investigations that is orders of magnitude greater than could previously be performed in one *in silico* “pot”.

Acknowledgements

I would like to thank the following people who contributed to the delivery of this tome.

Firstly, thanks to those in the School of Computer Science at the University of Nottingham. Thanks to my supervisor Natalio Krasnogor for his insight, persistence and encouragement, and for giving me the opportunity to study there. Thanks to colleagues, past and present: Jamie Twycross, Francisco J. Romero Campero, Daven Sanassy, James Smaldon, Pawel Widera, Hongqing Cao, Maria Franco, Jerzy Kozyra, Claudio Lima, Jaume Bacardit - who contributed directly - and the other members the Automated Scheduling, Optimisation and Planning & Interdisciplinary Computing and Complex Systems research groups.

Thanks to colleagues in the Centre for Biomolecular Sciences: Karima Righetti, Steve Higgins, Miguel Camara and Stephan Heeb.

Thanks to colleagues at the Weizmann Institute: Ofir, Yair, Uri, Tuval and Udi.

Thanks to my examiners, Marian Gheorghe and Jaume Bacardit, for a very pleasant conversation.

Big thanks to my family, especially my mum for her timely hugs and dad for his continuing support. Thanks to Anna for her love and patience, her parents Les and Chris for their care, and to the rest of my family and friends who had to listen to me talk about it.

Finally, I want to thank the organisations that have provided financial support for my PhD studentship and research position - from public funds - through the following grants:

- *(Semi)Formal Artificial Life Through P-systems & Learning Classifier Systems: An Investigation into InfoBiotics* [EP/E017215/1] Engineering and Physical Science Research Council (EPSRC)
- *CADMAD: Paving the Way for Future Emerging DNA-based Technologies: Computer-Aided Design and Manufacturing of DNA libraries* [265505] European Commission ICT Research in FP7

For my parents, and Anna.

Contents

1	Introduction	1
1.1	Background and motivation	1
1.1.1	Exemplar molecular-multicellular systems	2
1.1.2	Multicellular modelling	8
1.1.3	Biomatter compilation	10
1.2	Aims and scope	11
1.3	Main contributions	14
1.3.1	The Infobiotics Workbench	14
1.3.2	The DNALD language and DNA Library Designer	16
1.4	Published and presented work	17
1.5	Structure of the dissertation	18
2	Computer-aided design for synthetic systems biology	20
2.1	Biomodels	20
2.2	Stochastic simulation algorithms	25
2.3	Model checking	31
2.4	Model optimisation	32
2.5	Outlook	34
3	Biomodel specification	36
3.1	Introduction	36
3.2	SBML	37
3.3	Boolean networks	38
3.4	Petri nets	39
3.4.1	Petri net variants	41
3.5	Process calculi	41

3.5.1	π -calculus	42
3.5.2	Stochastic π -calculus	43
3.5.3	BioAmbients	46
3.5.4	Brane calculi	46
3.5.5	Beta-binders and BlenX	46
3.5.6	PEPA and Bio-PEPA	48
3.6	Rule-based approaches	49
3.6.1	κ	49
3.6.2	P systems	50
3.6.3	MGS	58
3.7	Conclusions	59
4	Lattice Population P systems	62
4.1	Introduction	62
4.2	Formal definitions	63
4.3	Machine-readable data formats	67
4.3.1	MCSS-SBML	67
4.3.2	LPP systems XML	72
4.3.3	LPP systems DSLs	78
4.4	Reusability of LPP systems	81
5	The Infobiotics Workbench	85
5.1	Getting started	85
5.2	Simulation with MCSS	91
5.3	Model checking with PMODELCHECKER	107
5.4	Parameter and model structure optimisation with POPTIMIZER	119
5.5	Summary	125
6	DNALD: a language for DNA Library Design	127
6.1	Background	127
6.1.1	Artificial DNA synthesis	127
6.1.2	DNA libraries	128
6.1.3	Programming combinatorial DNA library specifications	130

6.2	The DNALD language	130
6.2.1	Origins	131
6.2.2	Specification	137
6.3	A real combinatorial DNA library using DNALD	143
7	DNA Library Designer	148
7.1	An IDE for DNALD	148
7.2	Features	149
7.3	Conclusions	158
8	Software engineering	160
8.1	DNA Library Designer	160
8.1.1	Software stack	160
8.1.2	DNALD grammar implementation	161
8.1.3	Parsing DNALD into Ecore models	165
8.1.4	Validation	165
8.1.5	Evaluating DNALDs to DNA libraries	166
8.2	The Infobiotics Workbench	170
8.2.1	Software stack	170
8.2.2	Experiment parameter classes of the Infobiotics Dashboard	173
8.2.3	Handling simulation results with the McssResults class	178
9	Conclusions and future directions	183
9.1	Restatement of motivations	183
9.2	Overview and contributions	184
9.3	Evaluation	186
9.4	Future directions	188
	Bibliography	189

List of Figures

1.1	Schematic of a cross-section of the <i>A. thaliana</i> root cells	3
1.2	Phenotypes regulated by quorum sensing in bacteria.	5
1.3	The quorum sensing hierarchy in <i>P. aeruginosa</i>	6
1.4	Programmed Turing pattern formation in synthetic bacterial colonies.	9
1.5	An overview of the Infobiotics Workbench and its components.	15
1.6	Suggested paths through the dissertation.	19
3.1	π -calculus processes with paired communication channels.	43
3.2	Graphical π -calculus of positively-regulated transcription	45
3.3	Bitonal operations in Brane calculi.	47
3.4	The initial configuration of a P system.	51
3.5	Operations of P systems with activate membranes.	53
3.6	Editing an MP graph with MetaPlab.	58
4.1	Graphical representation of a (stochastic) SP system.	64
4.2	SP systems containing reactions of a gene regulatory network	65
4.3	MCSS-SBML compartment naming conventions for four adjacent cells	71
4.4	Schematic of the <i>Const</i> module	73
4.5	Schematic of the <i>Neg</i> module composing a <i>Const</i> module	74
4.6	Schematic of a <i>bacteria</i> P system with <i>Neg</i> module and translocation rule.	77
4.7	Modules 'flattened' to a set of rules.	77
5.1	Infobiotics Dashboard main window showing the available experiments.	85
5.2	The Infobiotics Dashboard with multiple LPP files open.	86
5.3	Example simulation parameters file.	87
5.4	Infobiotics Dashboard command line interface	88

5.5	Standalone interfaces for Infobiotics Workbench components	88
5.6	The Infobiotics Dashboard with multiple experiment interface tabs	89
5.7	Flow of information through the Infobiotics Workbench.	90
5.8	All simulation parameters	92
5.9	Main stochastic simulation results interface.	94
5.10	CSV options	99
5.11	Histogram plotting interface.	101
5.12	Timeseries plot styles.	103
5.13	Comparison of error bar alternatives	105
5.14	Paired surface plots showing expression patterns of fluorescent proteins.	106
5.15	A stitched frame from an exported video of a surface plot.	107
5.16	Frames from animations of three synthetic biology models	108
5.17	Compare editor-produced images of pattern formation model.	109
5.18	Confocal microscopy of bacteria with pattern formation circuit.	110
5.19	Comparison of <i>in silico</i> and <i>in vivo</i> Turing patterns	111
5.20	PMODELCHECKER parameterisation interfaces	112
5.21	Temporal formulas interface.	113
5.22	Editing a temporal formula.	114
5.23	Help with temporal formula syntax	114
5.24	Viewing the generated PRISM model	115
5.25	PRISM-only PMODELCHECKER parameters.	116
5.26	MC2 input parameters	117
5.27	Reusing previously simulated results with MC2	117
5.28	Edited partially parameterised simulation for MC2	118
5.29	A running model checking experiment	118
5.30	Model checking experiments results interface.	120
5.31	POPTIMIZER input parameters.	122
5.32	Target timeseries tooltip.	123
5.33	Accessing POPTIMIZER documentation via the Help menu.	123
5.34	POPTIMIZER evaluation and optimisation algorithm parameters.	123
5.35	Choice of two fitness functions	124
5.36	The four parameter optimisation algorithms provided by POPTIMIZER.	124

5.37	OPTIMIZER results interface.	125
6.1	A single Y-operation, the basic recursive unit of CADMAD	129
6.2	The basic example DNAPL library visualised.	133
6.3	Set of example DNAPL library visualisations.	136
6.4	Zoomed in view of the output sequences computed from the azurin library . . .	146
6.5	DAWG of fragments constituting all outputs of the azurin library	147
7.1	DNA Library Designer interface.	150
7.2	Multiple editors and collapsible views	150
7.3	Syntactic validation and colouring.	151
7.4	The Outline view.	152
7.5	Marking occurrences of references throughout the library.	152
7.6	Finding and replacing text within and between files.	153
7.7	Validation errors and quick fixes.	155
7.8	Comparing differences between libraries and versions.	155
7.9	Definitions view.	156
7.10	Sequence Fragments view.	157
7.11	Sequences view.	157
7.12	Library visualisation view.	159
8.1	Dependencies of DNA Library Designer and its components.	161
8.2	Railroad diagram of the file/library structure from the DNALD grammar.	162
8.3	Railroad diagram of the DNALD expression grammar.	163
8.4	Railroad diagram of terminals in the DNALD grammar.	164
8.5	The Infobiotics Workbench software stack	171
8.6	Example parameter file for a simulation experiment.	174
8.7	An example parameters template file	175
8.8	Generated parameterisation interface for model checking	176
9.1	Screenshots of a prototype Xtext-based editor for Infobiotics DSLs	187

List of Tables

4.1	Library of P system modules for basic transcriptional regulation and translation.	68
4.2	Three simple models of unregulated (constitutive) expression, positive autoregulation and negative autoregulation of a gene.	69

List of Listings

4.1	<i>Const</i> module XML	73
4.2	<i>Neg</i> module XML	75
4.3	Module library XML	75
4.4	Rule constant with additional parameters used by POPTIMIZER	75
4.5	Individual P system XML	76
4.6	Example LPP system model XML file	78
4.7	Library of promoter modules used in synthetic biology models.	79
4.8	Pulse propagation SP system DSL alphabet.	80
4.9	Individual SP system with a single top-level compartment (single cell).	80
4.10	Initial multiset of objects in the DH5alpha compartment.	80
4.11	Boundary rules modelling complexation of quorum sensing effectors.	80
4.12	Promoter module instantiation in DH5alpha cell rule set.	81
4.13	Transport rules relocating a quorum sensing signal.	81
4.14	Reusable square lattice definition.	82
4.15	Composed LPP system of colony designed for pulse propagation.	83
5.1	Interactive Python session working with data exported in NPZ format.	100
6.1	Example of a simple DNAPL library.	132
6.2	FASTA file of the output sequences in the DNAPL library example.	132
6.3	Example DNAPL codon table.	134
6.4	DNALD's default codon table (<i>E.coli</i> K12)	138
6.5	Summary of current DNALD functionailty as a DNALD library.	142
6.6	DNALD library investigating post-transcriptional regulation of azurin.	145
6.7	Extract from generated FASTA file of azurin library <code>azurin.fasta</code>	146
8.1	<code>McssResults.functions_of_amounts_over_runs</code>	180
8.2	NumPy reduce functions used by <code>functions_of_amounts_over_runs</code>	182

Chapter 1

Introduction

Chapter abstract

This chapter sets the scene for the dissertation, describes the goals of the research undertaken, the methodology used and the projects this work underpins. It will delineate the scope of the thesis and discuss how the contributions made compare to and depart from previous approaches. Finally, the results of the research will be summarised and the structure of the text explained.

1.1 Background and motivation

Systems Biology [1, 2] seeks to understand *life as it is* through the *in silico* reconstruction of molecular interactions that combine to produce emergent cellular behaviours. This approach has led to the discovery of design principles of biological systems such as recurrent motifs of gene networks [3, 4], which perform information processing tasks such as filtering specious fluctuations in environmental molecules and produce controlled responses.

Synthetic Biology, a new ten-year-old discipline [5], seeks to understand *life as it could be* through the construction of modularised genetic circuits (known generally as bioparts) and programming of minimal cellular chassis (from the top-down [6] and from the bottom-up [7]). This engineering approach assumes that the encoded molecular networks can be made similarly modular (due to chemical specificity or spatial localisation [8]) and can be maintained and exploited for the hierarchical integration of biological parts into devices, and devices into systems, where intracellular subsystems are orthogonal to each other. Whether this assumption holds true is unclear, but emulating orthogonality is one of the grand challenges of synthetic life [9]. It may be that cells, rather than circuits, are the most appropriate embodiment of modules, in which case the understanding and principal application of cellular interactions will be crucial for the successful programming of synthetic biological devices.

Although these fields may seem to have quite different aims, there is considerable overlap between Systems and Synthetic Biology. In terms of complementary gains, knowledge from one field is often directly applicable in the other, for instance the elucidation of evolved control mechanisms suggests patterns that can be implemented in novel gene regulation circuits. In

terms of how their investigations are conducted, both require and are driving the improvement of methods and tools for *in silico* modelling, simulation and analysis of molecular and cellular populations in space and time. Dynamic models, each a design or hypothesis, are the primary means of establishing our level of understanding and validating it by making experimentally testable predictions.

The majority of biomodels consider just one scale of biological organisation, molecular networks of single cells in particular, whereas ultimately it is desirable to integrate models at many scales and of multiple subsystems. The key motivation is to move away from prescriptive implementation of behaviours at one level and instead enable those to emerge as the consequence of behaviours at a lower level. This concept was elegantly summarised by a colleague in the context of vesicle computing [10]:

“there are few simulation and modelling approaches that take into account the fact that biological systems work over different length and time scales. Understanding a system as being composed of units which in turn are composed of other smaller units is the essence of hierarchical reductionism. It is import to realise however that abstractions of scale are constructs designed to aid understanding, all processes in a biological entity are emergent from interactions at the lowest possible level. Therefore, the creation of realistic high level design technique for vesicle computing systems will only succeed if the assumptions and abstractions made at the high level are correct in terms of low level interactions.”

A virtual human against which we could trail future synthetic cellular medicines is still remote, but an achievable step forward is to scale up from the current generation of models by bringing molecular and multicellular systems together into one suitably generic modelling framework. The first goal of the research presented here is the development of a such a framework. We elucidate its requirements from a selection of example systems relevant to our collaborators.

1.1.1 Exemplar molecular-multicellular systems

Two natural multicellular systems whose behaviour is determined both by molecular interactions and spatial organisation are quorum sensing in the pathogenic bacteria *Pseudomonas aeruginosa* and the root node development of the model plant *Arabidopsis thaliana*. Investigations into the structure and function of these systems were conducted in collaboration with colleagues at the University of Nottingham’s Centre for Biomolecular Sciences and Centre for Plant Integrative Biology. A third system of *in silico* designed and synthetically reprogrammed bacteria leverages that knowledge to implement Turing pattern formation in colonies of *Pseudomonas aeruginosa*.

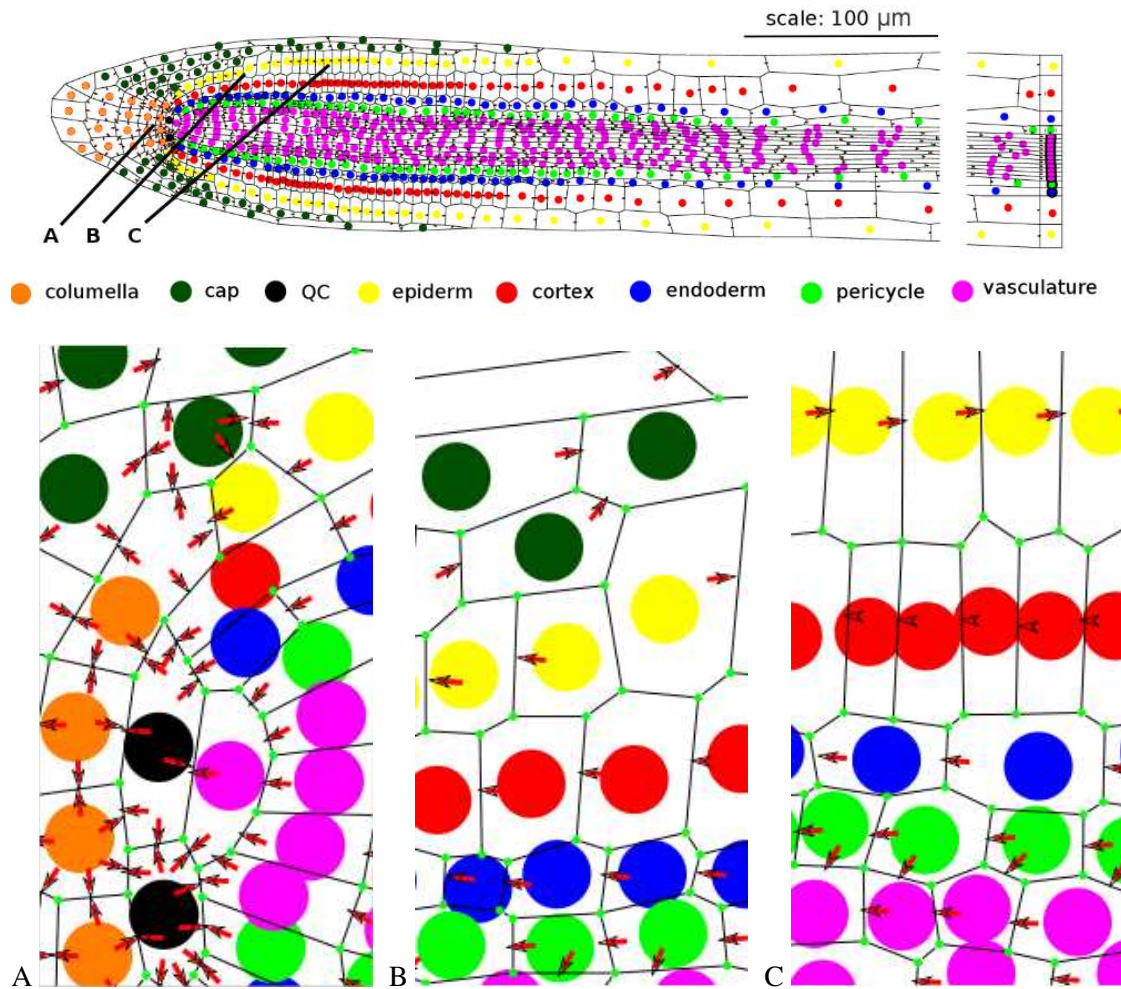


Figure 1.1: Schematic of a cross-section of the *A. thaliana* root cells showing position and directionality of PIN transporter proteins responsible for efflux of the plant hormone auxin (represented by red arrows). The nuclei of each cell coloured according to tissue type. The three panels below show the direction of efflux across the cell membrane at different points of the tip, located by A, B and C above.

The root node of *Arabidopsis thaliana*

The cells of the root node are organised as concentric layers of tissue, with the cells of each layer arranged in single cell width “files” extending from the root tip. The cubic nature of plant cells, as defined by the cell wall, and the elongation of root cells which occurs only in the developing tip of the root node leads to a somewhat regular structure where the faces of cells in adjacent layers are aligned. Figure 1.1 shows a schematic of the roots organisation, with the nuclei of each cell coloured according to tissue type.

The software developed as part of this PhD has been successfully used to model auxin flow [11] and abscisic acid related signal transduction networks [12] in the root tip of *A. thaliana*. Observations of intracellular components by staining/fluorescent tagging and microscopy producing cross-sectional slices were used to abstract a two dimensional grid of cells. Figure 1.1 shows

the directions of auxin transport by PIN proteins to neighbouring cells. In order to faithfully reproduce the observed behaviour, the modelling framework needed to capture the organisation of the one-way transporters embedded in some, but not all, intercellular interfaces.

Quorum sensing systems in *Pseudomonas aeruginosa*

Different species of bacteria colonise almost every accessible environment on the planet, including humans. That they are able to persist in challenging and changing environments is due to adaptations that have created diverse, differential social behaviours. One such adaptation is quorum sensing, which in *Pseudomonas aeruginosa* is a clinically-relevant target for systems-biological investigation.

Quorum sensing (QS) is the phenomenon of coordinated group behaviour correlated to the **local** density of a bacterial population. Bacteria that perform QS constantly produce and secrete small diffusible signal molecules that bind to and activate constitutively expressed receptor (*R*) proteins, which either regulate or are transcription factors that activate certain genes, including (*I*) *proteins* which synthesise the signal molecule, a process known as **autoinduction**. The concentration of signalling molecules required to fully bind the *R protein* receptors is such that only when the bacterial population is sufficiently dense will there be a enough signal diffusing in the environment to activate the receptor and effect gene activation. The subsequently (auto)induced production of additional signal molecules creates a positive feedback loop that reinforces the activation of individuals and the population, causing all of the cells to begin transcribing QS-associated genes within a short time of each other.

This ability to switch phenotypes in a synchronised manner means the bacteria can grow within a host without alerting the immune system, until such a time as there are sufficient numbers (sensed by the concentration of signal molecules in the environment) to weather an immune response, at which point they can be more aggressive, invade tissues and establish treatment resistant biofilms. Figure 1.2 shows some more of the phenotypes regulated by quorum sensing. The overall effect is to limit in a small population (i.e. individual cells) behaviours that would only be of benefit in a large population, allowing sparse populations to gain a foothold in their surroundings.

Pseudomonas aeruginosa is a free-living Gram-negative bacteria, commonly found in soil and water. It is also an opportunistic pathogen of humans that infects compromised tissues, in particular the respiratory tract of patients with cystic fibrosis and the urinary tracts of patients fitted with catheters. The quorum sensing system of *P. aeruginosa* controls expression of 10% of the genome including genes responsible for the production of virulence factors - elastase, lectins,

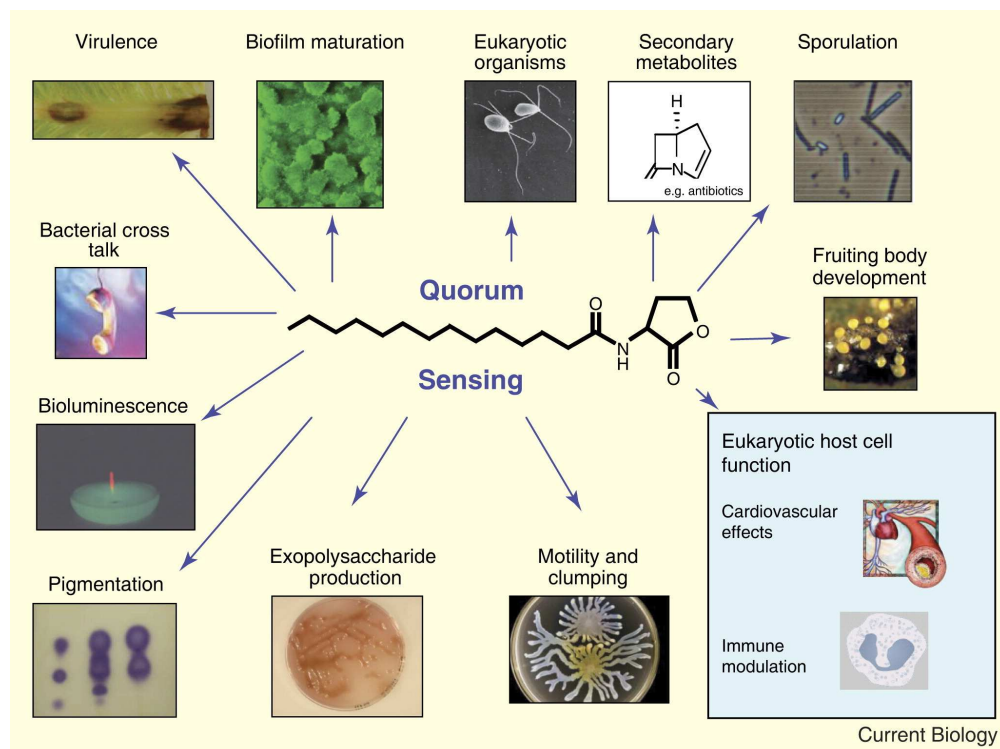


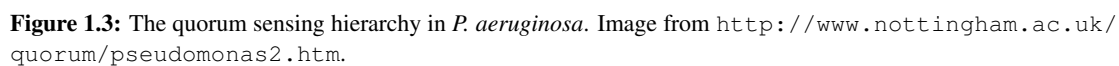
Figure 1.2: Phenotypes regulated by quorum sensing in bacteria (taken from [13]). The signalling molecule 3-oxo-C12-HSL (centre) is responsible for coordinating biofilm clumping and maturation, swarming motility, exopolysaccharide and virulence factor production in *Pseudomonas aeruginosa*.

exopolysaccharides and pyocyanine - which enable the bacteria to inhibit lymphocyte proliferation, disrupt tight junctions between tissue cells, and form biofilms.

To date the *P. aeruginosa* QS system is one of the most complex studied, with three subsystems mediated by two N-acyl homoserine lactone signalling molecules (3-oxo-C12-homoserine lactone and *N*-butanoylhomoserine lactone, AHLs similar to those first discovered as the bioluminescence determinant in *Vibrio fischeri*) and two quinolones (2-heptyl-3-hydroxy-4-quinolone, referred to as the *Pseudomonas* quinolone signal or PQS, and 4-hydroxy-2-heptyl-quinoline, also known as HHQ).

The Las subsystem controls both the Rhl and HHQ/PQS subsystems, as shown in figure 1.3. The gene pairs *lasI/lasR* and *rhlI/rhlR* encode the synthase and receptor proteins for the 3-oxo-C12-HSL (3OC12) and C4-HSL signals of the Las and Rhl subsystems respectively. When bound to 3OC12, LasR activates transcription of *lasI* and *lasR* (as it is an autoinducer), as well as *rhlI* and *rhlR*. Similarly, when bound to C4-HSL, RhlR autoinducer activates transcription of *rhlI* and *rhlR*. The promoter of *lasI* is bi-directional so that while increasing LasI expression, and therefore 3OC12 levels, LasR also increasing *rsaL* expression. RsaL is an inhibitor of *lasI* which therefore contributes to a decreasing 3OC12 levels.

As figure 1.3 shows, LasR also activates transcription of *pqsR* and *pqsH*, the former encoding



the receptor protein PqsR and the latter encoding an enzyme which catalyses the conversion of HHQ into PQS. Both HHQ and PQS bind to and activate PqsR which in turn upregulates (differentially depending on the signal bound) expression of the *pqsABCDE* operon that encodes four enzymes required for HHQ synthesis, and PqsE, required for the production of pyocyanin and swarming behaviours [14]. Activated PqsR also upregulates *rhlR* and *rhlI*, while C4-HSL activated RslR downregulates *pqsR* and *pqsABCDE*; a mechanism of self-regulation similar to LasR and RsaL.

Alongside these three quorum sensing subsystems are others that respond to additional extra-cellular signals. RetS, GacS and LadS mediate the phosphorylation of GacA, which when phosphorylated promotes expression of *rsmZ* and *rsmY* small RNAs that form hair-like structures with 5-loops, each of which sequesters an RsmA protein. RsmA is known to bind to the mRNAs of the virulence genes *lecA*, *hcnA*, increasing or decreasing their stability and rates of translation. PQS is thought to interact with a mystery protein that upregulates the transcription of the *rsm* genes, as does RhlR. Through these mechanisms the quorum sensing circuits achieve fine control of translation (enabling swifter changes in phenotypes than at the genetic level) in response to population density and other environmental factors.

The complexity of the *P. aeruginosa* quorum sensing system, together with its relevance to clinical medicine, makes it a suitable target for the systems biological approach, where new understanding of the interaction networks involved could be used to rationally design novel “quorum quenching” antimicrobials. The biomodel components required include gene regulation (decomposable into transcriptional motifs), stochastic signal production and extra-cellular diffusion.

Pattern formation in synthetic bacterial colonies

Components with inducible promoters from the *Pseudomonas* quorum sensing systems are suited to the design of synthetic microbiological systems exhibiting population-level control of gene expression. Uses in synthetic biology include synchronisation [15] of the famous synthetic oscillator the Repressilator [16], and program pattern formation in bacterial colonies [17].

From these components members of our laboratory have built modularised models of bacterial colonies that exhibit pattern formation arising from the activation and repression of reporter genes mediated by exchange of signal molecules. The goal was to design and experimentally validate the role of double negative feedback loops in the emergence of patterns in developmental processes by implementing a synthetic circuit in *Escherichia coli* colonies that resembles natural circuits found in organisms like the sea urchin [18].

A sketch of the design of this synthetic circuit is presented in the figure 1.4a. It consists of six individually well-characterised modules arranged in two branches, one formed by modules 1,

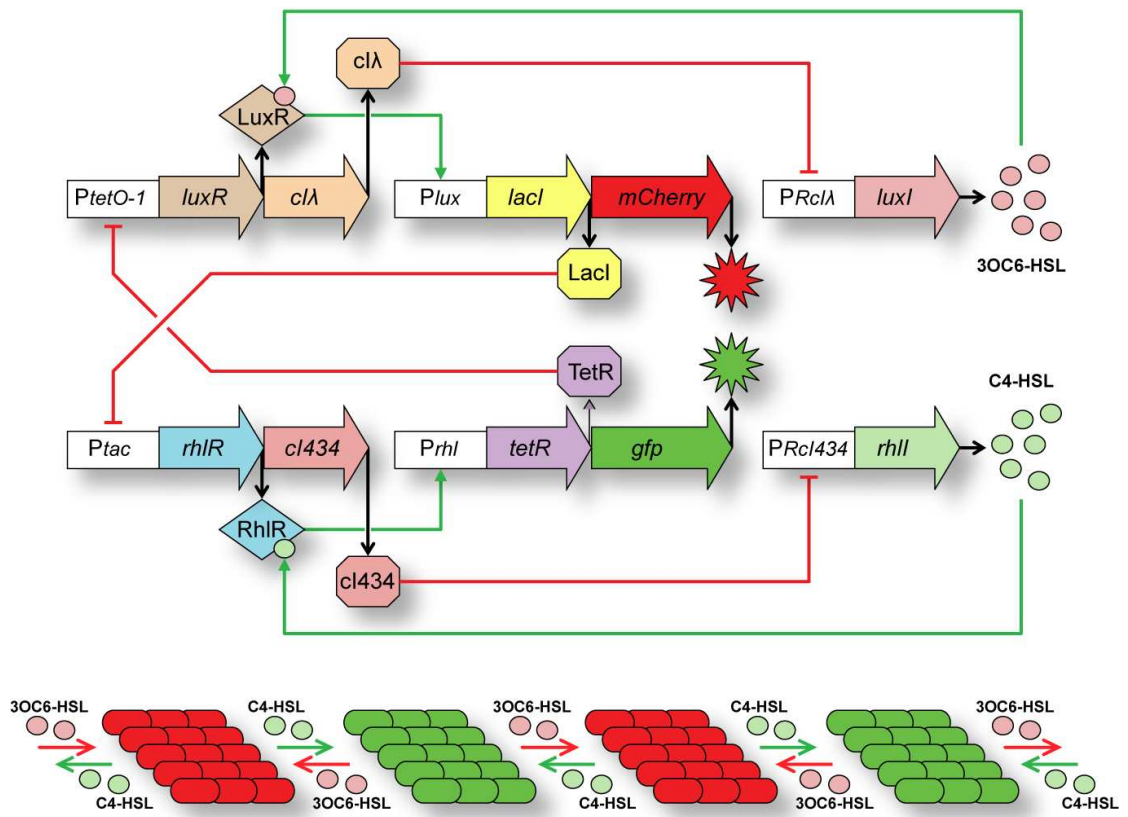
2 and 3 and the other by modules 4, 5 and 6. Together these compose a device with a double negative feedback loops resembling a latch electronic circuit that ensures exclusive activation of the two different branches.

Stochastic simulation (described in sections 2.2 and 5.2) of a model colony of the transformed cells resulted in the emergence of spatio-temporal patterns similar to that shown in figure 1.4b. Exhaustive analysis of the dynamics of the circuit using simulations and model checking aided the identification of potential flaws in the circuit design and proposed solutions for them.

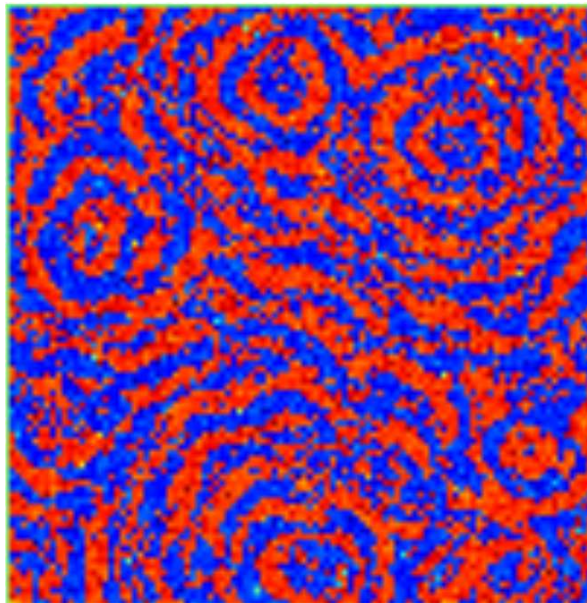
1.1.2 Multicellular modelling

Each of three exemplar systems discussed in section 1.1.1 has a strong spatial component where molecule exchange between adjacent cells, coupled with stochastic noise from low levels of certain molecules, determines the eventual phenotype. In order to investigate biological phenomena such as quorum sensing and development in higher eukaryotes, or characterise the emergent behaviours of many synthetic cellular devices operating in concert, it is vital that we have a suitable means of formally describing and simulating these systems. Quantitative predictions from models are required for designing wet-lab experiments to validate or refute the hypotheses they represent. When system dynamics are subject to stochastic noise at molecular and population levels this must be properly accounted for in our models as discrete, depletable quantities and not simply abstracted away. The mechanisms cells have evolved to manage and exploit stochasticity will be an important guide to ensuring the correct functioning of embedded synthetic circuits.

There are many barriers to effectively modelling colonies or tissues of cells at the discrete-stochastic level, which have not become apparent or manifested sooner due to the necessary focus on developing foundational models of molecular networks in single cells. Building large-scale or reusable models is hindered by on the one hand by the inappropriateness of existing largely mathematically oriented standards (in particular SBML) and on the other by the proliferation of overly complicated alternatives drawn from computer science such as process calculi or novel formalisms which focus only on certain aspects (e.g. κ with protein complexes), and the relatively immaturity of their software implementations compared to conventional modelling approaches (reviewed in chapter 3). This is especially true with regard to stochastic models at the colony and tissue levels which have so far received scant attention and therefore no concerted efforts have been made to investigate the difficulties of either formally describing multicellular models or consistently interrogating and plotting the high-dimensional datasets simulations of these can generate (i.e. quantities of molecular species in many cells and potentially subcellular compartments, sampled over many timepoints and independent runs). Addressing these issues



(a) Synthetic gene network for pattern formation.



(b) A simulated pattern formed by a 100 × 100 grid of cells each containing a model of the circuit in (a).

Figure 1.4: Programmed Turing pattern formation in synthetic bacterial colonies.

is vital as the ability to compare alternative models and contrast their dynamics with reality in a reproducible manner is a fundamental requirement for the model-driven investigation and design of biological systems. The software frameworks that do so must scale with the complexity and size of the models. Furthermore, transitioning from models to the DNA sequences that are needed to create knockout strains and implement synthetic bioparts for purposes of model validation has only just begun [19].

1.1.3 Biomatter compilation

Much experimental biology depends on the manipulation of DNA sequences, as genomic DNA provides a mechanism by which we can effect lasting change in cells. Researchers modify and recombine natural DNA to uncover its function and to modify or to create new functions and reporters. Whereas the editing of computer programs is as easy as word processing, the editing of DNA is still performed in a slow and expensive labour-intensive fashion. Improvements in DNA synthesis technologies are unburdening skilled molecular biologists of the target DNA creation process, so that they are free to focus on more interesting problems such as designing more extensive versions of their experiments with which to gather more conclusive evidence and explore alternative hypotheses.

Emerging second-generation DNA synthesis technologies [20, 21, 22], that exploit subsequences commonalities aim to maximise DNA reuse through robotic molecular biological manipulation, offer a scalable alternative to chemical synthesis and assembly, and a realistic means by which to conduct *combinatorial* investigations of synthetic biological systems by manufacturing *libraries of variants*. These could include the exhaustive exploration of *biopart* interactions [23], necessary to characterise the orthogonality of modules on which the engineering approach to synthetic biology is predicated.

Designing the DNA sequences combinations required for such a study is not a trivial task. Doing so manually using copy-and-paste takes time for complex combinations, is error prone and increasing likely to lead to accidental deletions or missing combinations. Scripting languages like Python are ideal for coding simple programs that can avoid these kind of errors when the task is a straightforward pairwise cross-product. However, as the complexity of the task assumes a more combinatorial aspect, with the exclusion or manual patching of certain combinations, the chance of introducing errors also increases. Spotting “off by 1” errors in long contiguous blocks of four letters is just one example of subtle error that can have severe consequences for a large DNA library, which a proliferation of *ad hoc* solutions only increases the likelihood of replicating. Also, communicating sets of plain sequences to manufacturers fails to effectively convey the intentions behind them and their interrelationships, that might otherwise be useful for troubleshooting and prioritisation.

As with the specification of large mesoscopic multicellular models, the difficulty is precise management of an unwieldy artifact that has defied suitable abstraction because the dimensionality of the problems investigated thus far (primarily due to the difficulty of obtaining arbitrary DNA sequences cheaply) was small enough that manual manipulation was not sufficiently onerous as to necessitate an alternative. The correct and intelligible specification and communication of combinatorial DNA libraries would be made significantly easier by the availability of a high quality tool for the succinct manipulation of DNA sequences shared by both consumers and producers.

The thesis of this dissertation is that sequence-aware, model-driven hypothesis generation and *in silico* prototyping, complemented by scalable DNA library manufacturing (in short, *biomatter compilation*), will facilitate exploration of biological problem spaces at much grander scales than are possible today. **We call this computer-aided synthetic systems approach to biology *Infobiotics*¹.**

1.2 Aims and scope

The goal of this dissertation is to develop suitable abstractions and software tools for computer-aided design in synthetic systems biology. The specific aims are improving the means by which specifications of (1) models of molecular reaction networks in multicellular systems, and (2) combinatorial libraries of DNA, are produced, interpreted and verified. The outputs of this research will be the tools and methods produced, and the attention paid to their usability.

With regard to aim (1) we are primarily concerned with reaction networks involving genes, their products and regulatory behaviours. Other than the compartmentalisation of reactions by membranes we do not consider the basal functioning of host cells, or the chassis in synthetic biology parlance, which is assumed not to interfere with our genes of interest. This dissertation is *not* about modelling metabolism, a better studied and understood aspect of biology [24] when compared to biomodelling of information processes in cellular populations. However, the modular stochastic-discrete modelling formalism we will present for transcriptional and signalling networks could equally be used to formalise the reaction networks underlying metabolism (although the large molecular populations of metabolites would, for simulation and comparability purposes, be more usefully represented as real-valued concentrations). Indeed, a number of chassis-specific metabolic models would be a helpful adjunct both as a base on which to build models of synthetic circuits that influence or are influenced by metabolic processes and as a sizable test case for our modelling approach and simulation algorithms.

¹EPSRC grant EP/E017215/1: (Semi)Formal Artificial Life Through P-systems & Learning Classifier Systems: An Investigation into InfoBiotics <http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/E017215/1>

This research aims to build on the recent work of [25] (instigated by [26] and developed further in [27, 28, 29]) which advocates the extension of *P systems* with a stochastic semantics to create *stochastic P systems*, as a suitable modelling framework for intracellular processes in multicompartmental models of genetic [30] and signalling networks [31, 32].

Briefly, as they will be described at length in section 3.6.2, *P systems* are a formal language inspired by the compartmentalised structure of Eukaryotic cells, which consist of three types of component: *membranes* defining regions of space and typically arranged in a nested hierarchy, formally a tree (intracellular compartments), *multisets of objects* representing compartment states (discrete molecular populations), and *sets of multiset rewriting rules* associated with each membrane (reactions).

Stochastic *P systems* (SP systems) assign a rate constant to each rule, that together with the multiset of objects on the left hand side of the rule provides enough information to simulate realistic trajectories of the temporal dynamics of the *P system*. Stochastic simulation of SP systems can be performed either with a multi-compartmental stochastic simulation algorithm (SSA) [32], or with a standard SSA such as the Direct Method [33] (other state-of-the-art methods are reviewed in section 2.2), but only after *flattening* the *P system* to a single compartment and renaming the objects appropriately (e.g. subscripting with the compartment index) so as to preserve the topology of the interaction network defined by the rules.

In addition to simulation with SSAs, SP systems are amenable to *probabilistic model checking* [34, 25], meaning that properties of a *P system* model, for example the expected number of LasR molecules not bound to 3OC12 with respect to time (in a hypothetical model of our QS example in section 1.1.1), can be expressed using an appropriate temporal logic formula and automatically checked with a probabilistic model checker such as PRISM (Probabilistic and Symbolic Model Checker) [35].

We extend this formalism, to create *Lattice Population P systems*, which enable the specification of cells as stochastic *P systems* distributed on a two-dimensional lattice, with rulesets that are optionally composed of *modules of rules*. None, some or all of the objects, stochastic rate constants and compartment labels used by the rules in the module may be exposed as *module parameters*, capturing, at one extreme, purely abstract network motifs [3], and at the other, fully-specified, immutable reaction networks such as synthetic biological devices. Modules may be instantiated multiple times in the same compartment with different parameter values, or in multiple compartments with the same or different values.

We have further modularised our implementation of LPP systems by separating out individual and population *P system* specifications from lattice definitions and libraries of module definitions, to facilitate the reuse of cohesive collections of components between models. Together

these approaches make enable the incremental specification of large multi-cellular models in a highly parsimonious manner.

With regard to aim (2) genetic manipulation is one of the key skills of any molecular biologist, although more by necessity than choice. Once the hypothesis for a sequence-based experiment is developed, that for instance a particular set of genes is responsible for a certain phenotype, then the molecular biologist can proceed to create knockout strains where each gene (or combination of genes) is deactivated. The necessary steps of extraction, mutation, ligation and cloning are time consuming and error prone. Often a junior but highly trained scientist will spend months assembling the sequences required to test their hypothesis. The time delay from inception to evaluation prevents a thorough search of the space of possibilities and the investigation must proceed in an iterative manner, following promising avenues (determined from limited data) at the expense of other, potentially successful but seemingly less promising, ones due to unknowably incorrect observations and unforeseen implementation challenges.

As well as disabling genes biologists may also wish to modify sequences for improved protein yields, create fusions with fluorescent reporters or investigate entirely novel sequences. The issues around construct assembly are amplified when, as part of a synthetic biology project, one needs to implement operons of new or preexisting devices made of bioparts, but the optimal, or at least viable, distribution of parts between operons or the order of parts within an operon is unknown. In this instance, a library of variant operons representing each permutation of the chosen parts would enable an exhaustive search of the combination space, where each variant could be relatively quickly cloned into the chassis of choice and functionally screened. That data would facilitate not just a principled selection of most efficacious combinations but may also bring to light incompatible combinations revealing unknown cis/trans-regulatory interactions.

The BioBricks strategy initiated at MIT [36, 37] can help in the construction of combinatorial libraries because BioBricks are designed to be composed together, reused and repurposed. BioBricks enable researchers to combine parts and devices in a sequential manner using one of the specific flanking restriction sites universal to all BioBricks. The process of chaining together a pair of BioBricks removes the restriction sites between the two (creating a scar), allowing the same restriction enzyme to be used again for chaining subsequent bricks.

Rearranging the order of bricks to construct a combinatorial library means either restarting the process from the scratch for each combination, or forking the process after each additional BioBrick to create all combinations in parallel with the less but still considerable effort and planning. Moreover the BioBrick approach is restricted to only the parts available in the registry² which may or not suit a particular synthetic biology project. The parts registry is growing rapidly due to contributions from iGEM³ teams but a perception that this are diluting the quality

²<http://partsregistry.org/Catalog>

³<http://www.igem.org>

of the parts registry has spurred the emergence of new efforts, notably BioFab⁴, aiming to serve industrial quality parts. Thus we aim at taking a first step towards enabling through suitable “tooling” the rapid design of combinatorial libraries of DNA.

1.3 Main contributions

This work involved contributions to two systems and synthetic biology related projects: “(Semi)Formal Artificial Life Through P-systems & Learning Classifier Systems: An Investigation into Infobiotics” for which we developed *the Infobiotics Workbench*; and “CADMAD: Paving the Way for Future Emerging DNA-based Technologies: Computer-Aided Design and Manufacturing of DNA libraries” for which we are developing *DNALD and DNA Library Designer*.

1.3.1 The Infobiotics Workbench

The major deliverable of this research is the Infobiotics Workbench software suite, incorporating simulation, model checking and optimisation experiments for our novel spatial discrete-stochastic generic modelling framework based on P systems and tailored towards systems and synthetic biology research. This work is the result of a critical investigation of the current state of modelling tools for biological systems and a direct response to the lack of frameworks for building large spatially discrete and stochastic models of multicellular systems.

In addition to designing the framework and coordinating the integration of the experimental components, the authors principal contribution has been the development of a graphical user interface, the Infobiotics Dashboard, for conducting *in silico* experiments and analysing the results. Figure 1.5 gives an overview of the Infobiotics Workbench capabilities, showing how model inputs in a range of formats are fed to the model checking, simulation and optimisation executables, processed and those outputs collected and interrogated using functions of the Infobiotics Dashboard.

The Infobiotics Workbench is open-source software available for Windows, Mac and Linux from the Infobiotics website at <http://www.infobiotics.org/> and released under the GNU General Public License (GPL) version 3. A hosted repository for the Infobiotics Dashboard source code is reachable at <https://bitbucket.org/jvb/infobiotics-dashboard/overview>. The latest revision (807) of the Infobiotics Dashboard source distribution consists of 22,622 lines of Python code (213 Python modules, 240 classes, 1171 functions/methods) plus a small amount of shell scripts required for building.

⁴<http://biofab.org>

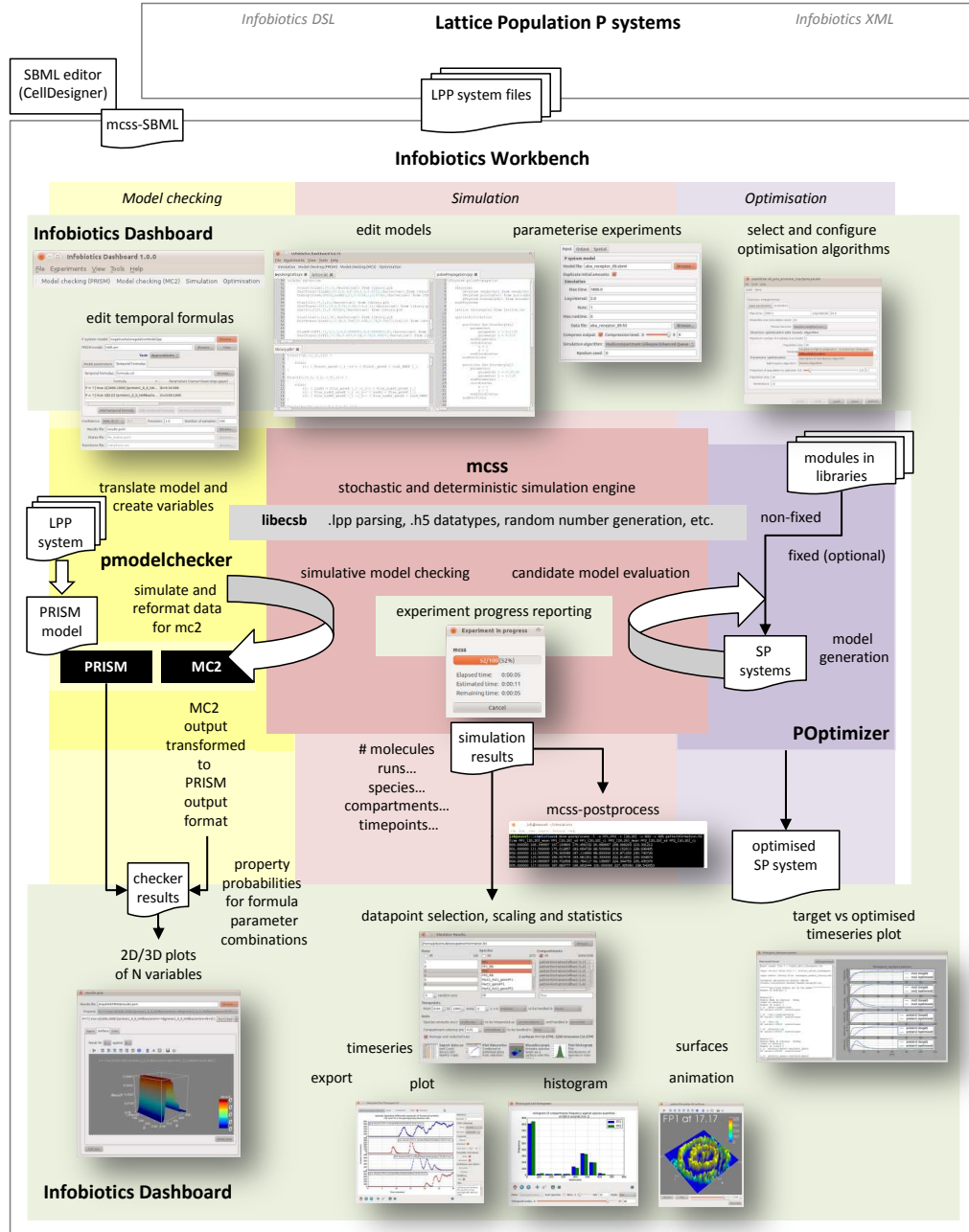


Figure 1.5: Flow of information through the components of the Infobiotics Workbench (revisited in section 5.1). The contribution of the Infobiotics Dashboard, which integrates the components of the Workbench, is highlighted by a green background.

1.3.2 The DNALD language and DNA Library Designer

DNALD is our language for the specification of combinatorial DNA libraries. DNA Library Designer is an implementation of DNALD and an integrated development environment (IDE) for working with it. DNA Library Designer uses the state-of-the-art domain-specific language framework Xtext⁵ to implement its DNALD parser and customisable user interface within Eclipse, on top of which we have added the interpreter and advanced validation. The combination of the Infobiotics Workbench *in silico* experimentation platform for a biologically-tailored modelling language that associates DNA sequences with the genetic components of reusable modules of reaction rules, and a language, interpreter and IDE for the precise specification of non-random combinatorial DNA libraries, forms a complete *in silico* toolchain for the rational prototyping, testing, tuning and production of a manufacturable requirements list for a synthetic biological system.

Downloads of DNA Library Designer for 32/64-bit Windows, Mac and Linux are available at <http://gandalf.cs.nott.ac.uk/dnald/>.

⁵<http://www.eclipse.org/Xtext/>

1.4 Published and presented work

Parts of the work described in this dissertation have resulted in the following publications:

Journal articles

J. Blakes, J. Twycross, F. J. Romero-Campero, N. Krasnogor, “The Infobiotics Workbench: an integrated in silico modelling platform for Systems and Synthetic Biology”, *Bioinformatics* 27(23), p.3323-3324, 2011

Conference papers

F. J. Romero-Campero, J. Twycross, H. Cao, J. Blakes, N. Krasnogor, “A Multiscale Modeling Framework Based on P Systems”, Workshop on Membrane Computing (WMC9), Lecture Notes in Computer Science (5391), p.63-77, 2008

J. Smaldon, J. Blakes, N. Krasnogor, D. Lancet, “A multi-scaled approach to artificial life simulation with P systems and dissipative particle dynamics”, in Genetic and Evolutionary Computation Conference, Atlanta, Georgia, USA, p.249-256, 2008. *Nominated for a best paper award in the Artificial Life, Evolutionary Robotics, Adaptive Behavior, Evolvable Hardware track.*

Extended abstracts

G. Rampioni, F. J. Romero-Campero, J. Blakes, S. Heeb, P. Williams, N. Krasnogor, M. Camara, “The quorum sensing system of *Pseudomonas aeruginosa* incorporates an incoherent feedforward loop”, SIMGBM 29th National Meeting, Pisa, Centro Congressi CHR, September 2011

D. Sanassy, J. Blakes, J. Twycross, N. Krasnogor, “Improving Computational Efficiency in Stochastic Simulation Algorithms for Systems and Synthetic Biology”, European Conference on Artificial Life, Paris, France, August 2011

J. Blakes, N. Krasnogor, F. J. Romero-Campero, J. Twycross, “An Executable Biology Methodology for Systems and Synthetic Biology”, Proceedings of the ECCB Satellite Meeting on Probabilistic Modelling in Computational Biology, Cagliari, Sardinia, September 2008

1.5 Structure of the dissertation

This dissertation will provide the reader with a background on existing approaches in computer-aided design in systems and synthetic biology, and experimental methods for simulation, model checking and optimisation. It will then present new software methods which increase the scope of *in silico* and *in vivo/vitro* biological investigations. Therefore, the dissertation is organised into the following chapters (paths through which are suggested in figure 1.6):

Chapter 2 outlines the rationale for applying computer-aided design methods to biology. It emphasises the model-driven systems approach to understanding biological systems, describing what goes into a formal model and the types of inquiries that can be made *in silico*. A distinction is made between macroscopic continuous mathematical models and mesoscopic discrete computational models on which the dissertation mainly focuses.

Chapter 3 reviews the literature on computational formalisms applied in biomodel specification, that motivated and informed our choice of modelling framework.

Chapter 4 introduces our novel discrete spatial and stochastic formalism: Lattice Population P systems. It contains formal mathematical definitions for each component of an LPP system, and goes on to describe the conversion of those into the three machine-readable data formats that evolved with and are supported by the components of the Infobiotics Workbench.

Chapter 5 is a visual tour of the functionality of the Infobiotics Workbench from the users perspective of performing *in silico* experiments and working with the results. Particular attention is paid to the simulation results interface of the Infobiotics Dashboard which accounts for a large proportion of the creative and software development effort of the author.

Chapter 6 moves on from dynamic models to address the issue of obtaining the genetic material necessary for experimental validation and implementation of models or designs in modified organisms. We take a view which recognises the potential for reuse between alternative candidate models and modular synthetic gene networks, and exploits this in the DNALD language yielding concise specifications of combinatorial DNA libraries.

Chapter 7 presents DNA Library Designer, our integrated development environment for editing and interpretation of DNA library designs with DNALD; the second major software contribution of the PhD.

Chapter 8 addresses individually the software engineering aspects of the Infobiotics Dashboard and DNA Library Designer.

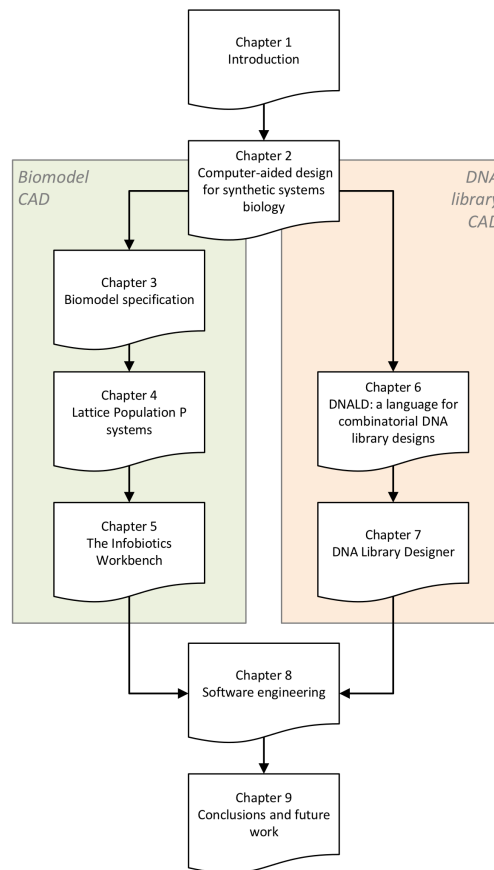


Figure 1.6: Suggested paths through the dissertation. The next chapter offers a rationale for systemic model-driven biological research in which CAD of both biomodels and DNA libraries has a role to play. Thereafter the dissertation can be considered to run in two parallel tracks addressing these topics separately, before converging on the engineering of the described tools and drawing some conclusions about the research.

Chapter 9 concludes the dissertation with a reflection on the topic and an appraisal of the current state of the Infobiotics Workbench and DNA Library Designer with a view to their future integration.

Chapter 2

Computer-aided design for synthetic systems biology

Chapter abstract

In this chapter, the key methods of modelling and *in silico* experimentation for Systems and Synthetic Biology are introduced.

2.1 Biomodels

Systems are comprised of *components* and their *interactions*, features that act in concert to implement the processes or functions that determine the systems behaviour. Within the system, subsets of features can be identified as *subsystems*, as could the system itself in the context of a subsuming larger one.

Systems can be separated from their surroundings by a conceptual or physical boundary that distinguishes what is part of the system from what is not [38, 39].

The cell is the fundamental unit of biological organisation, for which the cellular membrane provides a suitable boundary by which we can define the cell as a system of molecular components. Through mitosis or fission, cells ensure that they do not exist in isolation, forming tissues or bacterial colonies at the next level of biological organisation. These systems are comprised of cellular subsystems, the spaces they inhabit and the molecules they exchange with the environment and each other.

As components of both cellular and multi-cellular systems, molecules are therefore key observables through which the functioning of multi-cellular systems can be elucidated. Gene expression, diffusion, transport, non-covalent interactions and chemical reactions are all modellable processes which change the quantities of populations of distinct molecular species, as the system realises itself in time and space.

Modelling is intrinsic to any scientific activity. Simplified models are necessary for us to understand, reason about, communicate and compute with, systems that are too complex to be dealt

with whole, as is often the case with biological systems. The objectives of modelling are to capture the essential features of a phenomenon, to disambiguate the evidence behind those features and their interactions, and ultimately move from a qualitative understanding to a quantitative one.

Model development is an iterative process which begins in the mind of the individual researcher [1]. Published data, personal findings, assumptions and questions go into an initial model which is used to make falsifiable predictions. Inconsistencies between the model and experimental results lead to refinement of the model and the process of prediction-validation is repeated until a consistent picture emerges that explains all known observations and achieves good enough predictive accuracy. In this way the model drives its own development by bootstrapping the acquisition of data needed to improve it.

The attempt to formalise results and open problems in a model often uncovers a lack of knowledge, disagreements between sources and ambiguities arising from language that require clarification [40]. How you select features, disambiguate and quantify depends on the *goals* behind your modelling enterprise.

For *systems biology* the basic goal of modelling is to clarify current understanding by formalising what the constitutive elements of a system are and how they interact. This requires knowledge coming from the literature, databases and expert opinion. The intermediate goal is to test current understanding against experimental data: does the model, our statement of what we think is happening, agree with what we observe in experiments. Here there must be a common form in which both results can be compared directly, often a series of measurements over time.

The advanced goal of systems biology is to predict beyond current understanding and available data and techniques, moving to *in silico* experimentation, conducting infeasible or out-of-reach experiments using validated models. The grand challenge for systems biology will be the integration of many models, in multiple formalisms and at multiple scales into comprehensive, dynamic models of model organisms such as *Saccharomyces cerevisiae* [41], *C. elegans* [42], and eventually a virtual human [43].

For *synthetic biology* the dream goal of modelling is to be able to program and optimise new biological systems on a computer, before implementing those designs in the laboratory and having them functioning as expected. From the top-down assembly of minimal genomes [44] to bottom-up protocells with lipid vesicles [7, 45, 46] (touching on artificial life), synthetic biology aims to design, construct and develop artificial biological systems; offering new routes to genetically modified organisms for bioremediation and the production of biofuels, biosensors [47, 48], smart drugs [49], hybrid computational-biological devices and synthetic living entities.

A formalism (sometimes known as a metamodel) defines an organising system for model components which limits what is expressible and therefore what can be modelled using that formalism.

The level of detail at which the system can be modelled depends greatly on the choice of formalism in which to encode the model. The well-defined syntax and semantics of the formalism ensures that interpretation of the model can be done by machines in a regular and consistent manner.

In what follows we first describe the two main classes of models used in systems biology, namely phenomenological *mathematical continuous models* and mechanistic *computational discrete models*, and how these can be simulated, optimised and checked.

Mathematical continuous models

The vast majority of reaction network models used in systems biology have up until recently been mathematical, based on systems of coupled ordinary differential equations (ODEs). An early and famous example of an ODE model is the 1952 Hodgkin-Huxley model of neuronal action potentials [50] for which the authors received the 1963 Nobel Prize in Physiology or Medicine. The same model has since been applied to action potentials in cardiac myocytes [51].

A differential equation model is the set of coupled differential equations that describe the dynamics of the system. A linear ODE is an equation that describes the change in concentration of a molecular species (the derivative) with respect to a single experimental variable, usually time. A non-linear ODE describes the change in concentration of a molecular species in terms of the concentration of other species with respect to time. Differential equations containing derivatives with respect to multiple experimental variables such as time and space are called partial differential equations (PDEs) [52].

In an ODE model each molecular species in the model is defined as continuous variable $X_i(t)$ which represents the concentration of species i at time t . An ODE is written for each species that describes the change in X_i over time as a function of other variables in the system. The rate of each reaction is represented using a function (called a kinetic law) which depends on one or more rate constants. Equation 2.1 gives the general form.

$$\frac{dX_n}{dt} = F_n(X_1, \dots, X_n) \quad (2.1)$$

Kinetic laws

Kinetic laws approximate various reaction schemes. The choice of which to use is at the discretion of the modeller. First order reactions (a single species reacting) such as a complex dissociation, transformation and degradation are modelled using the *exponential decay law* where the rate of the reaction k is proportional to the concentration of the reactant: $k \cdot X(t)$.

Second order reactions are typically modelled in one of two ways:

1. Reactions where two molecules react or form a complex use the *mass action law* where the rate is proportional to the product of the concentrations of the reactants: $k \cdot X_1(t) \cdot X_2(t)$.

2. Enzyme-catalysed reactions are modelled using *Michaelis-Menten dynamics*:

$$\frac{k_p EX}{K_m + X}$$

where E represents the concentration of the enzyme and K_m (the Michaelis-Menten constant) $= \frac{k_d + k_p}{k_b}$, a composite of the kinetic constants associated with the binding, dissociation and production reactions.

Often constitutive transcription and translation are modelled as first order reactions but when a gene is under negative control the production of mRNA is modelled using *Hill dynamics* $\frac{k_p}{1 + \left(\frac{X}{K_h}\right)^n}$ where where $K_h = \frac{k_r}{k_f}$ the forward and reverse constants for repressor binding.

Solutions

Small systems of ODEs can be solved mathematically by setting the concentrations of each species at time zero and calculating the derivatives using the rate constants and kinetic laws. Each set of initial concentrations deterministically produces a unique trajectory or time-course that can be compared with observations made in the laboratory. The combination of differential equations with initial conditions is called the *well-posed initial value problem* for which a unique solution is guaranteed under weak conditions. Large or highly non-linear systems of coupled ODEs cannot be solved in feasible time and are instead approximated by numerical analysis such as the Euler or classical 4th-order Runge-Kutta (RK4) methods available off-the-shelf in many software packages.

Assumptions and violations

The correctness of an ODE model relies on two assumptions holding:

1. The system is well-stirred so that concentrations are the same in all places.
2. Concentrations vary *continuously* and *deterministically*. This assumption is only valid when the number of molecules is sufficiently high (an approximate lower bound is 10^3 molecules) and reactions are fast.

These assumptions are often violated by cellular systems due to intracellular crowding and prolonged reactions such as transcription and translation. Bacteria are perhaps small enough to be considered well-mixed but eukaryotic cells which are large and compartmentalised are definitely not. Crucially, some molecular species, particularly genetic elements, are often present in very small numbers.

Stochasticity

It is well understood that chemical reactions involve discrete, random collisions between individual molecules. Theoretical statistical physics states that randomness or fluctuation level in a system are inversely proportional to the square root of the number of particles: $noise \sim \frac{1}{\sqrt{n}}$ [33]. This random element determines the next state of the system and therefore the system is not deterministic but *stochastic*. Stochastic ordinary differential equations exist but these do not avoid the problems of continuous concentrations. Cellular systems, where genes typically exist in a single copy and are either available or unavailable (1 or 0) to participate in reactions, are stochastic in the extreme. It is known that some biological systems actually exploit stochastic noise to perform a particular function. This phenomena is called *stochastic resonance* [53].

Despite these facts, ODEs still dominate as a modelling technique due to compactness (sometimes combining several species into one abstract species as in the Oregonator [54] series of models) and because they produce time course data. Another explanation for their dominance is that it is a product of cultural inertia since differential calculus has been used for more than 300 years.

Computational discrete models

The formalisation of biological systems using alternatives to mathematical equations has recently received much interest as a deeper mechanistic understanding of molecular networks is fed into, and obtained from, models of biological systems. The distinguishing feature of discrete computational models as opposed to continuous mathematical models is that, unsurprisingly, some aspects of these models - space, time, quantity, states - belong to discrete domains. Formalisms where interactions are modelled as discrete events have come to be known collectively as *Executable Biology*. The dynamics of such discrete-event systems is highly dependent on how events are selected to occur. A somewhat unifying event selection is stochastic simulation discussed below. The representational and computability aspects of a wide selection of executable biology formalisms is discussed in detail in chapter 3.

A discrete quantities model of a chemical reacting system defines the state of the system as the number of molecules of each species at any given time. The *Chemical Master Equation* (CME) completely determines the probabilities of each reaction in a well-mixed chemical system, at constant temperature and volume, given the current state. Unfortunately the CME is actually a system of as many coupled ordinary differential equations as there are combinations of molecules that can exist in the system, and can only be solved analytically for a very few simple systems [55]. Fortunately a more tractable approach exists. Instead of solving the CME we can construct numerical realisations of the systems state over time, that is, generate trajectories

of system using a Monte Carlo algorithm: Gillespie's *stochastic simulation algorithm* (SSA) [33, 56], elaborated below.

2.2 Stochastic simulation algorithms

Stochastic computational models of biochemical systems have several advantages over deterministic mathematical models. When the populations of chemical species present are low these ODEs are inaccurate due to noise within the system. The Chemical Master Equation (CME) models the chemical kinetics of the system as a Markov process that captures this noise (stochasticity) and Stochastic Simulation Algorithms (SSAs) are procedures to generate trajectories in compliance with the CME. Simulating trajectories of the CME computationally has the advantage of being similar to the nature of biological experiments as it is a discrete, mechanistic approach.

Stochastic Simulation Algorithms SSAs were first described by Gillespie in the form of an exact method to numerically calculate the time evolution of a well-stirred spatially homogeneous system of reacting molecules with specified reaction channels [33]. Gillespie showed how possible trajectories of the CME can be obtained by applying a kinetic Monte Carlo approach. He initially produced two *exact* techniques: the First Reaction Method (FRM) [56] and the simpler but equivalent Direct Method (DM) [33] and subsequently showed these to be a rigorous derivation of the CME [57]. DM is easier than FRM to explain and understand. It is also faster: the main drawback of the FRM over DM being that it requires a random number to be drawn *for each reaction* at each iteration, whereas DM requires only 2 random numbers per iteration.

Stochastic rate constants and propensities

To each reaction a stochastic rate constant is assigned. Usually this is the rate constant of the equivalent differential equation's mass action kinetics divided by Avogadro's number to get the number of molecules per unit time from the concentration. The *propensity* of a reaction is its stochastic rate constant multiplied by the 'hazard function', the number of unique combinations of reactants. According to Gillespie reactions can have zero, one or two reactants but no more than two as the probability of three molecules colliding with the correct orientation and energy is infinitesimal. Ternary or higher order reaction must be decomposed into a series of second order reactions; the SSA however makes no guarantee that subsequent reactions will occur immediately after.

Direct Method

Each iteration of DM selects a reaction to occur and time to have elapsed before it occurred. The steps of the algorithm are:

1. Calculate the *propensity* of each reaction and store them in an array
2. Sample the index of the reaction to fire from a probability distribution according to its relative propensity by:
 - (a) Summing the propensities to obtain the total propensity a_0
 - (b) Multiplying a_0 by a uniform random number between 0 and 1 to get a random number a_f in the range of the sum
 - (c) Set an index variable i to 0
 - (d) Set a cumulative sum variable a_c to 0
 - (e) Add the propensity of reaction at index i to a_c
 - (f) If a_c now exceeds a_f the index of the reaction to fire is i , otherwise increment i and loop to (e)
3. Update the state vector to reflect the changes in the number of molecules brought about by the occurrence of the reaction: decrement each of the reactants and increment each of the products
4. The time interval in which the reaction τ_0 occurs is sampled from a negative exponential distribution with a_0 as the parameter using the formula: $\tau_0 = \frac{-\log(rand())}{a_0}$.
5. Add τ_0 to the simulation time

This procedure is repeated until the simulated time exceeds the predetermined maximum simulation time.

Computational expense

Exact SSAs must consider every single reaction that occurs in a system in time order, resultantlly these are very computationally expensive because a large number of reactions may happen at over a very small time-step. The phenomena of *stiffness*, present in systems where there are large differences in the scale of some reaction rates, can affect worsen this situation. Stiffness is exacerbated when there are many reactants of the faster reactions, in which case the propensities of these reactions dominate meaning they are selected highly frequently, and the time increment

becomes very small, so that these come to dominate processing time and slow the speed of the simulation relative to reality.

The output of the SSA for a model represents a single stochastic realisation of a system's behaviour, and often many runs of the simulation are required to generate an average output that falls within a suitable error range. Ensembles of 1000s of trajectories are often necessary to achieve statistical significance. Unless run in parallel, on a multi-core processor or cluster, the running time will increase linearly with the number of simulations required.

Performance improvements

Since the advantages of mechanistic stochastic models, in terms of realism and intuitivity, makes them highly desirable, but the considerable simulation cost involved makes their application prohibitive, a considerable amount of research has grown up around improving the performance of SSAs [58]. In order to address computational expense, many different variants of exact SSAs have been subsequently introduced (reviewed in [55]), including:

- Next Reaction Method (NRM) [59] which uses one rather than two random numbers, introduced the reaction dependency graph to reduce the number of propensity calculations made at each step to the only those that are directly affected (have reactants whose quantities are changed) by the fired reaction, and an indexed priority queue (a min-heap) for $O(1)$ access to the next reaction. Both of these measures work well for systems that are loosely coupled (few reactions depend on any other) but the cost of maintaining order in the heap can be detrimental for tightly coupled systems. Nevertheless it has come to be seen as the fastest exact SSA, as evidenced by its inclusion in many simulation packages [60].
- Optimized Direct Method (ODM) [61] runs a pre-simulation (typically 10% of the actual time) and uses information about the most frequent reactions to speed up the linear search by reindexing the reaction channels in descending likelihood of application. ODM also uses a reaction dependency graph.
- Sorting Direct Method (SDM) [62] too uses a reaction dependency graph but unlike the ODM reorders reactions dynamically by bubbling up applied reactions. The authors demonstrate its efficacy using a model of quorum sensing [63] where autoinducer producing genes are switched on late into the simulation and whose associated reactions quickly come to dominate, a situation which would totally undermine the efficiency of the ODM simulation.

- Logarithmic Direct Method [64] calculates an array of cumulative propensities when summing the total propensity and performs a binary search on these to achieve $O(\log M)$ performance when finding the next reaction. LDM may not necessarily be as fast as SDM or ODM but it avoids a potential inaccuracy that these methods introduce where the wrong reaction fires due to numerical truncation [55]
- the Composition-Rejection [65] achieves constant time performance for selecting the next reaction but due to the cost of the rejection sampling technique this only pays off when there are $> 10,000$ reactions.
- Partial-Propensity Direct Method (PDM) [66, 67] and variants [68, 69] use a data structure for partially calculated propensities and species dependency graph which offers better scalability better than optimisations based on the reaction dependency graph. When reimplementing the PDM we discovered a mistake in the published algorithm and informed the authors who then published a correction on their website: http://www.mosaic.ethz.ch/research/docs/PDM_NotesAndCorrections.pdf

Reusable implementations of most of the above SSAs can be found in StochKit [70], although in practice most simulator developers choose to reimplement a few exact SSAs, mainly DM and NRM, to better match their internal data structures for storing and recording species and reactions.

Approximate methods Distinct from exact methods, an approximate class of SSAs have been introduced which potentially deliver significant increases in computational efficiency for models with larger species populations. By appropriately sampling a probability distribution, multiple applications of some reactions can be made in a single algorithmic step. Therefore, a trade off between computational efficiency and an acceptable drop in accuracy is made. Notable *approximate* SSA methods include τ -leaping [71] and subsequent refinements [72, 73, 74, 75], and hybrid formulations such as the slow-scale method [76, 77, 78], an approach specifically created for modelling stiff systems (also reviewed in [55]).

Work has also been done on solving the CME without using Monte Carlo techniques. STOCKS implements the maximal time step method [79]. The Finite State Projection (FSP) method [80] claims to be more efficient than SSA techniques whilst providing a guaranteed higher accuracy as it can either consider the entire solution space or iteratively increase a truncated solution space to meet a specified level of accuracy. FSP has been extended to reaction diffusion systems [81].

As well as developing more efficient serial algorithms, another method of alleviating the simulation execution times of expensive algorithms is to formulate new parallel algorithms, or parallel

versions of the existing algorithms, which leverage the processing power offered by modern GPGPU and HPC systems. However, SSAs are inherently sequential algorithms and therefore difficult to parallelise [82]. This makes potential routes for significant optimisation of SSAs non-trivial. Li & Petzold created a GPGPU implementation of Logarithmic Direct Method by distributing multiple runs over GPU cores, which achieved an impressive $\sim 200\times$ speedup [83]. To cut down data access times they implemented a Mersenne-Twister (high periodicity PRNG) on the card. This was for multiple runs of a very small system as opposed to a single run of a larger system. In this ensemble approach to parallelisation the SSA is not itself being decomposed to run in parallel so the size of simulation is limited by the amount of shared memory that is considered essential for efficient computation on the GPGPU. Recent work by Klingbeil et al. [84] showed that, contrary to perceived wisdom, by ignoring shared memory (“thin-threading” as opposed to “fat-threading”) ensembles simulations can be performed faster and without size limitations. A GPGPU implementation which distributed the reactions of the FRM method over multiple threads to achieve a certain level of parallelism within the algorithm, yielded only a disappointing $2\times$ speedup [85]. Another avenue for hardware-acceleration of SSAs is to use Field Programmable Gate Arrays (FPGAs) [86, 87], chips that can be reconfigured during use to provide the optimal circuit for each application. For stochastic simulation this can be extrapolated to providing the optimal algorithm for every system [88].

Multiple volumes and subvolumes

As previously mentioned the Gillespie algorithm assumes molecules are homogeneously distributed in the liquid phase (well-mixed). This assumption can be violated in two ways that lead to models that do not match observed data. Firstly when the system is sub-divided by membranes and these membranes bound regions of different sizes or dynamically change over the course of the observation. Secondly when local concentrations of reactants determine the behaviour of the system.

The simplest exact implementation of a Gillespie algorithm over multiple membranes is to run a Gillespie algorithm in each compartment and then schedule the update of the system by the compartment containing the selected reaction with the lowest τ (in series [89, 31, 25] or parallel [90]). Normally the rate constants are implicitly bound to the volume of that compartment and therefore the same reaction in two compartments with different volumes must have different constants. These constants are inflexible and cannot change over time as a compartment grows, in the build up to cell division, or shrinks under hypotonic stress. At worst each reaction in each compartment would require a unique rate constant, when accurate rate information is already scarce. Also, a compartment containing other compartments, such as mitochondria in the cytoplasm, must enclose a volume larger than the sum of the volumes of its child compartments, but

for that same region the volume in which reactants mix will be less than the enclosed volume due to the space occupied by the enclosed compartment(s). Therefore rate constants would need to be perturbed as compartments are created, move and merge.

Deterministic approaches for modelling cell growth in the SSA where propensity is a function of time were previously considered [91, 92]. Before every iteration of the Gillespie algorithm the volume is calculated using the following formula: $V = (1 + t/T)$ where t is the time of the simulation and T generation time. Then, the stochastic rate constants of both types of second order reactions are divided by V (the rates of first order reactions remain unchanged). These linear volume change approaches do not account for dynamically changing compartment volumes that occur due to the simulated reactions. A plausible solution is to assign each compartment a volume variable incorporating this and the volumes of its immediate sub-compartments explicitly into the propensity calculation at an additional computational cost, as in our early work [93]. An alternative was proposed [94, 95, 96] whereby each molecular species is assigned a volume and that *the sum of volumes of the molecules* within each compartment determines the volume in which reactants mix. This is even more computationally expensive but remains in perfect agreement with the intention of the original SSA.

For systems with volumes too large to be considered well-mixed, or reactants that are known to diffuse slowly, volumes can be voxelised into subvolumes small enough that they can be considered well-mixed. The position of molecules within a volume can be modelled by these finer-grained simulations, as can the shape of those volumes, which are exemplified by the Next Subvolume Method [97] (NSM) implemented in MesoRD [98] and SmartCell [99].

Other factors affecting reaction rates

Other factors aside from volume that directly affect the rate of reactions are not modelled explicitly and remain implicit in the rate constant. Temperature and pressure are mostly assumed constant and uniform, and for most biological systems this is an acceptable assumption. Atomic interactions such as hydrogen-bonding, electrostatic effects and Van der Waals forces are disregarded, nor is bulkiness of some macromolecular species versus others considered.

In summary, despite the computational costs, SSAs **should** always be applied when populations of some molecular species are low, otherwise stochastic effects that determine the systems actual behaviour will not be captured [100, 101, 102].

Comparability to deterministic simulations

For systems that evolve into a single steady-state the time to reach that state will vary between simulation runs but the average of an ensemble of many stochastic simulations will reproduce the deterministic approximation provide the number of runs is large enough. However, for

systems that do not reach a single steady-state such as those that exhibit oscillatory behaviour [103] or where different regions of the model (several subpopulations of cells for instance) can exist in one of several steady-states, it is not appropriate to average over a number simulation runs or cells as this will give a false representation of the actual behaviour of the system. In such cases it can be desirable to ascertain what percentage of the runs or cells fall into one or other of the end-states, for which we apply model checking.

2.3 Model checking

By encoding a natural system into a formal system we can make inferences about the natural system and discover novel knowledge about the systems properties over and above simulating many trajectories of that system in the stochastic case, or stability and bifurcation analyses in the deterministic. A central mission of executable biology is to apply *model checking* to biological systems. Model checking goes beyond repeated simulation and observation to provide a formally verification that the model of real-life system is correct in all circumstances. It is most usually applied to mission-critical hardware, a nuclear power plant for example, and accompanying software controllers where the inputs (and outputs) are known and finite.

Some examples of questions that can be asked of a biological model with a model checker are:

- Does the number of molecules of transcription factor exceed 100 within 60 seconds in 90% cases?
- Until the concentration of the activated transcription factor promoter complex is greater than 0.01nM, is the probability of expressing the gene less than 0.5?
- Will the concentration of signal molecule ever drop below the threshold 0.1nM?
- In the steady state, what is the probability that the number of signal molecules is between 20 and 40?
- After the concentration of signal molecules exceeds 0.2nM, what is the probability of the concentration of activated transcription factor being greater than 0.1nM?

To check a model it must be converted into an appropriate specification formalism, which can be done automatically in some cases. Verifying a model means exhaustively enumerating all of its possible states over the range of possible inputs and transitions to produce every possible sequence of events (the chain of causality which can be summarised as a Markov chain). There is a combinatorial explosion of the number of states for any reasonably complex system and so the size of the models that can be checked is often very small. Constraining the ranges of inputs

with strict lower and upper bounds, and discretising continuous values can help ameliorate but not eliminate the problem.

Probabilistic model checking is a probabilistic variant of classical model checking augmented with quantitative information regarding the likelihood that certain transitions occur and the times which they do so. Probabilistic model checking works with discrete and continuous time Markov chains (CTMCs) or Markov decision processes. A continuous time Markov chain (CTMC) is defined by a set of states, a set of initial states and a transition rate matrix from which the rate at which a transition occurs between each pair of states is taken as a parameter of an exponential distribution. Queries which check model properties are defined in as logical statements, often temporal logics: CSL for CTMCs, PCTL for DTMCs and MDPs. PRISM¹ (Probabilistic and Symbolic Model checker) [35] is a probabilistic model checker that uses CSL, a stochastic quantified Continuous Temporal Logic for reasoning about propositions qualified in terms of time. PRISM was initially applied in the checking of models of signalling pathways [104] and subsequently to many other biological systems reviewed in [105, 106].

2.4 Model optimisation

Both stochastic and deterministic models are dependent on rate constants to accurately reproduce cellular behaviour. Unfortunately well-characterised rate constants are in very short supply, and those that are known for some models are used as ersatz values in models of similar systems. Finding good quality rate constants in the literature can be onerous and text mining approaches [107] endeavour to automate this task. In the scenario where the components and interactions are known but other parameters are not it is acceptable to try to estimate the rate constants using parameter optimisation to fit model dynamics to laboratory observations.

Parameter optimisation

Given a time series of quantities for one or more molecular species and a model of that system with incomplete rate information it is possible to explore the rate parameter space of the system using different combinations of rate values to simulate a set of timeseries that closely match the observed behaviour of the model entities in the real system. By this method models can bootstrap their own development. Additionally, there may be several distinct combinations of parameters that yield essentially the same time series, presenting several hypotheses to be explored, or additional knowledge about the system such as a particular dependence on one rate or a wide range for another. Some reaction networks tolerate wide ranges of rates for some

¹PRISM: <http://www.prismmodelchecker.org/>

reactions because their actual function is to control the production of a protein and keep its concentration within certain bounds.

The basic methodology of parameter optimisation is to execute a system with a set of parameters and compare the resultant output time series with the target time series using a measure of the distance such as root mean square deviation (RMSD). For deterministic models this entails solving the system with that set of parameters once and for all. For stochastic models it is necessary to run a statistically significant ensemble of simulations and compute the average of these before comparison (systems with oscillatory or switching behaviour will have exhibit large standard deviations). Based on the distance measure the parameters are tweaked and the execute-compare cycle iterated until the distance measure is minimised, at which point the rates are optimal.

Because the number of unknown parameters (degrees of freedom) may be large and the possible range of values also large or unknown, the parameter-space can be very large indeed and impervious to a brute-force method that would test every possible combination to find the best. In such circumstances we can use heuristic optimisation methods that make an adaptive exploration of the search-space: genetic [108] and memetic algorithms [109], differential evolution [110] and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [111], simulated annealing [112], ant-colony [113] and particle swarm optimisations [114] to name a few. Other non-heuristic numerical optimisation techniques as applied to computational systems biology have been extensively reviewed elsewhere [115].

Model structure optimisation

Parameter estimation is dependent on the model having the correct structure: components and interactions. If it proves impossible to find a set of parameters that reproduces the observed behaviour then it is reasonable to suspect that the model structure is incorrect. Work on the estimation of structure and parameters for systems biology models has shown promise. A particularly interesting example [116] is the reverse engineering of metabolic pathways using Genetic Programming (GP). Using only the time series of diacyl-glycerol, the final product of the phospholipid cycle, Koza's GP algorithm created the complete pathway including a feedback loop, bifurcation and accumulation points, estimated 3 out of 4 parameters to 3 significant digits (the other within 2%), and even postulated the existence of two intermediate substances. For the synthesis and degradation of ketone bodies GP produced a perfect fitness of 0.000 (zero RMSD) with all rates to 3 s.d. and the correct topology of the pathway.

In the optimisation component of the Infobiotics Workbench we use a genetic algorithm to recombine modules of reaction rules that constitute the model structure of single compartments. Genetic algorithms evolve a population of candidate genotypes, an encoding of a set of values

for each model object or parameter, preferentially selecting those with the fittest phenotype according to a fitness function. The genotypes of the fittest are recombined and/or mutated to yield new, potentially improved individuals that make up the next generation and the population gradually evolves towards an optimal solution. Techniques such as elitism, niching and tabu search can be used improve the performance of the EA and ensure the discovery of the global optima (or minima depending on the fitness function) over suboptimal local attractors.

2.5 Outlook

Practitioners of synthetic biology assume that modelling, simulation, optimisation and verification methods from engineering, mathematics and computer science, commonly employed in the creation of non-biological systems, can also be used to specify, design, construct, validate and deploy novel biological systems. This has resulted in a proliferation of formal languages for modelling synthetic biological systems with an emphasis on compositionality [117, 118] that reflects synthetic biology's central dogma of parts, devices, systems. These efforts are aiming to create integrated platforms where novel biological functions are designed *in silico* by composing validated models of parts (obtained from libraries) into new models of devices, behaviourally characterised by simulation and automatic verification, optimised for a particular model chassis, and compiled into DNA sequences ready for manufacturing. The computerisation of synthetic biology practice combined with emerging technologies for DNA synthesis and robotic manipulation of cells leads to a future where design, construction and screening of novel systems can be largely automated and thus scaled up to high-throughput and parallel production of engineered organisms. We can envisage a day when the pipelines for creating synthetic biological systems are controlled by programs which given a high-level description of some desired functioning perform the necessary rounds of design, production, testing and refinement required to meet that criteria without human intervention; with the associated benefits of improved reproducibility and debuggability. It might even be possible to conduct an exhaustive exploration and characterisation of the design space of living organisms through the combinatorial assembly of biological components, complementing the search algorithm of evolution by natural selection.

In this chapter we introduced the roles and importance of models in the research cycles of systems and synthetic biology, and how, when formalised as computational artifacts, they can be queried by *in silico* experimentation. We identified mathematical models, specifically ordinary differential equations, as the predominant means of phenomenologically representing and calculating the dynamics of cellular components in a continuous-deterministic manner. We then showed how the properties of cellular systems (small numbers of certain molecular species) violate the assumptions of this framework, and introduced the stochastic simulation algorithm as

a mechanistic means of computing realistic trajectories of molecule fluxes that correctly captures the stochasticity of molecular reactions, but which is also capable - for systems of limited stochasticity - of producing results that are consistent with the mathematical formulation. We discussed how formal analysis methods such as model checking can be used to verify (with a certain probability in the case of non-deterministic systems) the expected behaviour of biological models, and how optimisation methods, in conjunction with simulation, can be used to estimate parameters when rate constants data is unavailable.

In the next chapter we examine a number of discrete computational formalisms that have been applied to biomodel specification and evaluate these with regard to modelling multicellular systems from a stochastic perspective.

Chapter 3

Biomodel specification

Chapter abstract

This chapter presents a literature review of established and emerging mathematical and computational formalisms used to model biological systems. The formalisms reviewed are loosely grouped according to similarities in meta-models and historical relationships, as many are variants and continuations of others. Our aim is to assess the applicability of these to multicellular systems.

3.1 Introduction

Executable biology [119, 120] and *algorithmic systems biology* [121] propose the application of novel discrete and state-based formalisms such as: P systems, statecharts [119, 120], process algebras/calculi [122], Petri nets [123], boolean networks, BlenX [124] (formerly Beta-binders) and κ [125]; for the specification and implementation of biological models. These formalisms have *operational* semantics where the meaning of a model is realised through the sequence of steps of their execution. This is in contrast to mathematical models which have a *denotational* semantics where the meaning in the input-output mapping of each variable to some others, described by the set of equations, is realised by approximation or solution using other methods.

The motivation for developing and using these computational approaches is to benefit from the tremendous success computer science had in developing and understanding the only synthetic systems that come close in complexity to biological ones: software systems. Thus computer science has formulated and embraced useful ideas such as hierarchical abstraction for managing the complexity, and retaining understandability, of large concurrent systems, characteristically similar to the biological systems we hope to model. It is felt that they have a more appropriate structure for the problem domain than differential equations.

For example, modelling biological systems with P systems means specifying sets of rules describing the conversion of some reactant species into other product species for each type of compartment and constructing the model from those compartment types. In κ rules are made up of graph-like expressions that generalise to all molecular species which contain those sub-graphs,

thus avoiding the problem of having to specify many rules in systems where combinatorial complexity often arises such as signalling pathways. For π -calculus biomodelling means defining the state-space of a process (that could at different times be several distinct molecular species) and the communication-channels through which that process changes itself and other processes. A particular advantage of having this wealth of alternative codifications of biological systems is that each is amenable to different sorts of analysis. Boolean networks for instance, can be used to find attractors in the state space that correlate to steady-states. For probabilistic approaches the continuous time Markov chain (CTMC) representing the entire state-space of the model can be extracted, although it is probably unfeasibly large. Petri nets and P systems can be checked using model checking software, meaning that queries of properties of those models formulated in a suitable temporal logic can be automatically validated by building the CTMC or simulating the model over ranges of input variables.

However, executable programs are essentially creative processes that can have implausible, even impossible, behaviours compared to the biological systems they can be used to represent. Observations of real biological systems place limits on the possible sequences of events, fast reactions like diffusion occur more frequently than slow reactions like gene expression, and therefore the dynamics or trajectories of the system's state are constrained. To faithfully model biological reality these formalisms must be executed in such a way as to correctly reflect these limits. In same way that biological systems are executed by the application of universal physical laws to their chemical constituents, or binary and byte-compiled programs are executed by the physical or virtual machine they are compiled on, models of biological systems can be executed using simulation algorithms. Therefore the operational semantics of an executable biology model (the derivation of meaning from the series of steps) is realised not only by the chosen formalism but also by the choice of simulation algorithm.

3.2 SBML

The Systems Biology Markup Language (SBML) [126, 127] is a XML dialect for exchanging systems biology models. Today SBML is the *de facto* standard for interchange of models between biological modelling and simulation software, supported by over 230 software packages¹. For some packages SBML is the primary format for representing biological models, for others it is an export format allowing models developed with that package to be read by other packages with complementary modelling/simulation/analysis capabilities, and/or an input format allowing models developed in other packages to be read and manipulated. Due to its popularity a

¹<http://sbml.org/Community> (2012-01-04)

large number of models are encoded in SBML and available in databases such as the BioModels database [128], “a repository of peer-reviewed, published, computational models”².

SBML is designed to be agnostic in terms of the how biological interactions (i.e. chemical reactions) are encoded, enabling discrete-stochastic models that operate on populations of molecules as integers to be represented, as well as deterministic-continuous models (typified by ODEs) that use real-valued concentrations. However, due to the dominance of mathematical models and the ability of SBML to support kinetic laws in the definition of reaction rates, the majority of publicly available models using SBML belong to the deterministic-continuous modelling domain. Unless a deterministic model uses **only** (up to 2nd-order) mass-action kinetics the conversion to a discrete-stochastic is not possible to automate. This is because the entities implied by the kinetic laws may not correlate to molecular species, may be equilibrium constants, or in the case of Hill kinetics have a single parameter modelling the number of monomers in a complex, which must be extrapolated to the various intermediate species making up those complexes, plus many missing constants for the complexation reactions. Such conversions must be made manually, with whomever is performing the conversion guessing the original modellers assumptions and likely introducing some of their own.

3.3 Boolean networks

Introduced by Kauffman in the late 1960s as random models of genetic networks [129], boolean networks are the oldest example of an executable biological modelling formalism. A boolean network is a directed graph where each node represents a gene that is either 1 (active) or 0 (inactive). Edges between nodes contribute either positively (activation) or negatively (inactivation) to the node at which they are directed, providing the node from which the edge extends is active. Edges can be weighted to model the relative influence of contributing factors, just as in neural networks. The next state of a node in the network is determined by summing its positive and negative inputs in current state, becoming 1 if the total is greater than zero and 0 if the total is less than or equal to zero. (Elementary cellular automata are a particular case of a boolean network, where the state of a node is determined by the nodes in its spatial neighbourhood.)

Boolean networks are deterministic given their starting configuration for which there are 2^n possible system-wide states where n is the number of nodes. A complete assessment of the state space is therefore exponentially complex. However computing the behaviour of the network is computationally inexpensive and sooner or later the network will reach a previously visited state and due to determinism, converge onto one of a small number of attractors [130], curtailing the need for further execution.

²<http://www.ebi.ac.uk/biomodels-main/>

Boolean networks are qualitative in terms of quantities and time, can usually be constructed when data is scarce, and are therefore often chosen as a modelling formalism for their amenability to analysis rather than realism [131].

When two levels of activity becomes an insufficient granularity at which to model the systems components, Petri nets (3.4, next) are a natural alternative that offer finer improved specificity while preserving most of the analytical abilities of boolean networks.

3.4 Petri nets

Introduced in 1962 by Carl Adam Petri, Petri nets (place-transition nets, PT-nets, or simply PNs) have a “*concise and unambiguous representation*” [132] that model systems with concurrent behaviour and are particularly suited to modelling discrete asynchronous distributed systems.

Thoroughly validated in the analysis of engineering and computational systems, business processes, communication networks and manufacturing, Petri nets were first applied to biological pathways in the early nineties [133, 134] for a qualitative analysis on the combined glycolytic and pentose phosphate pathway of the erythrocyte cell. This confirmed their suitability for the representation of biological pathways and with the introduction of various extensions - stochastic, coloured and hybrid-functional - Petri nets have been used to produce high quality systems biology models [135].

A bibliography [136] of Petri nets applications in biochemistry for modelling, analysis and simulation summarises developments up to 2002; more recent contributions include the ubiquitously studied ERK signal transduction pathway [137], receptor signalling and kinase cascades, cell-cycle regulation and wound healing [138], and synthetic biology [139].

Petri nets visualise the topology of an interaction network formally as a bipartite graph composed of places and transitions (nodes), connected by directed arcs (edges) annotated with weights (1 is usually omitted). A place node typically holds a discrete non-negative quantity of ‘tokens’. The distribution of tokens in places at any one time is said to be the ‘marking’ of the net and represents the state of the system being modelled. A transition is ‘enabled’ if the places connected to it contain enough tokens to satisfy the weights on the edges. The basic execution model of Petri nets is non-deterministic and sequential. At each step multiple transitions may be enabled but only one randomly selected transition fires. When an enabled transition is ‘fired’ tokens are consumed from the input places and produced at the output places according to the weights on the arcs connecting them. The net can be initialised with any configuration of tokens and transitions fired to determine various properties such as whether another configuration is reachable from that point. In biological models molecular species are typically mapped

to places, molecules to tokens, transitions to reactions and the arc weights to the stoichiometric coefficients of the reactions.

One of the strengths of the basic Petri net formalism is to observe qualitatively, without the need for hard to obtain kinetic rates, whether experimentally observed states can be reached in the model. This 'first pass' validation is clearly illustrated by a PN model of the sucrose breakdown pathway in potato tuber metabolism [140], where an expected property of a metabolic system is that the net will never reach a dead end (no transitions can fire) while substrate is provided. A Petri net is said to be *reversible* if the initial state can be reached again from each reachable state. A *reachability* graph can be constructed where each marking reached is a node and directed arcs annotated with the label of a transition are edges showing the sequence of events between states. This shows the initial marking to be a parameter of the model as it potentially restricts the possibility of reaching any other particular marking. The reachability of marking M is whether or not it is in the reachability graph $R(N, M_0)$ of net N . It is possible that not all possible markings will not necessarily be reached in one or many runs due to non-determinism. The topology of the full reachability graph is equivalent to the underlying Markov chain of the system.

There are a number of other properties that can be discovered for a Petri net. *P-invariants* are the set of places for which the number of tokens is constant regardless of the sequence of transition firing (conservation relations). *T-invariants* are the set of transitions that return the net to a particular marking and indicate cyclical patterns in the firing of the net. In a model of apoptosis [132] it was shown that every T-invariant corresponds to an apoptosis pathway.

Liveness is whether all transitions are potentially fireable for all reachable markings of the net and can be used to determine the presence/absence of metabolic blocks on the progress of the system. *Boundedness* dictates that the number of reachable states is finite, an upper limit on the number of tokens in each place. For biological systems unbounded places might be suggestive of a disease state or simply that the model does not account for all phenomena.

Several additions to the basic formalism have been introduced that augment the functionality of the net:

- Transitions without input places are permanently enabled; places without input transitions have a constant number of tokens.
- Bounded places place a limit on the number of tokens and disable connected transitions when full.
- Bidirectional 'test' arcs enable places to influence whether a transition is enabled changing the number of tokens that residing there.

- Inhibitor arcs, ending in an open circle, enable a transition only when there are no tokens in the connecting place.
- Certain network motifs are formally interchangeable and allow the network to be simplified, improving readability. A motif may be abstracted into a single transition, a process of structural reduction (black-boxing). Such motifs are composable only if they do not duplicate places in other motifs (orthogonality). Similarly reversible reactions can be modelled using one hierarchical transition, where 2 concentric squares replace 2 transitions.

3.4.1 Petri net variants

A number of “higher-level” Petri nets have been introduced for systems modelling including **hierarchical** (Petri nets as tokens) and **prioritised** (an enabled transition cannot fire if another enabled transition has higher priority).

Time in standard Petri nets is represented only by the order of transitions firing, permitting only a qualitative analysis of the system it represents. A quantitative, notion of time is introduced by **stochastic Petri nets** (SPNs), where each transition has an associated rate from which a period of time is calculated upon firing and added to the global clock.

The firing delays in a SPN can follow one the Gillespie family of stochastic simulation algorithms [141], but other time elapse semantics are also found in the literature. General stochastic Petri nets allow for immediate and delayed firing. One execution of an SPN results in time series of the number of tokens in each place which can be compared to laboratory observations.

Standard PN and SPN places hold a discrete number of tokens but places can hold real-valued number of tokens, for instance representing concentration of molecular species (exemplified by ODEs). These are termed **continuous Petri nets**.

Coloured Petri nets (CPNs) provide a novel way of dealing with the combinatorial explosion of states addressed in rule-based modelling. Places in CPNs contain differently coloured tokens which can represent molecules of the place’s species but in different states. While nets with coloured tokens can be represented in the basic formalism using a greater number of places this reduces the understandability for all but very small models.

3.5 Process calculi

Process algebra (or process calculi) are a diverse family of related formalisms that describe distributed concurrent processes, such as the objects inside a computer program or a collection of programs, interacting. They are languages with a compact syntax and clear semantics providing a tractable algebraic theory that allows processes to be manipulated and reasoned

about. π -calculus [142], for concurrent mobile processes, is now a widely accepted model for interacting systems with a dynamically evolving communication topology. Other foundational process algebras are the Calculus of Communicating Systems (CCS) [143] and Communicating Sequential Processes (CSP) [144].

3.5.1 π -calculus

In π -calculus processes interact through communication channels with a shared name and complementary $!$ and $?$ symbols (send and receive action/co-action pairs). The result of each communication for that process follows the period ($.$). For the parallel composition $P|Q$, both P and Q result. For non-deterministic choice $P + Q$, either P or Q result. For example, in the system:

$$P := a?.(P|R)$$

$$Q := a!.0$$

P communicates with process Q along a and is replaced by P and R while Q is replaced by the null process 0 .

Channels can also send other channels as data and so transform the receiving process with new capabilities. In the system:

$$P := x!\langle y \rangle$$

$$Q := x?(y)$$

P sends and Q receives channel y along channel x . In this way complex processes are constructed from simpler processes and atomic actions. Processes with private channels $(\nu x)P$ resulting from a communication are only able to communicate on that channel with the process that communicated it.

Researchers postulate [145, 146] that what is true of processes inside a computer - binary interaction through selected channels in a particular scope - is also true for processes inside a cell. Cellular components are acting independently or interacting, dispersed in solution, producing localised changes that subtly alter the global state of the system. For biological models, process algebras consider molecules with binding sites as processes with *communication channels*. Interactions occur between two molecules, or complexes, at any one time through private channels, such as ligand binding sites, when molecules come into contact. The analogy was first made by Regev and Shapiro [147] and is elegantly demonstrated by their illustration in figure 3.1.

Whilst the application of process algebras to biological modelling has been thoroughly reviewed elsewhere [148, 149, 150, 151], we introduce here the most recent variants and discuss their development.

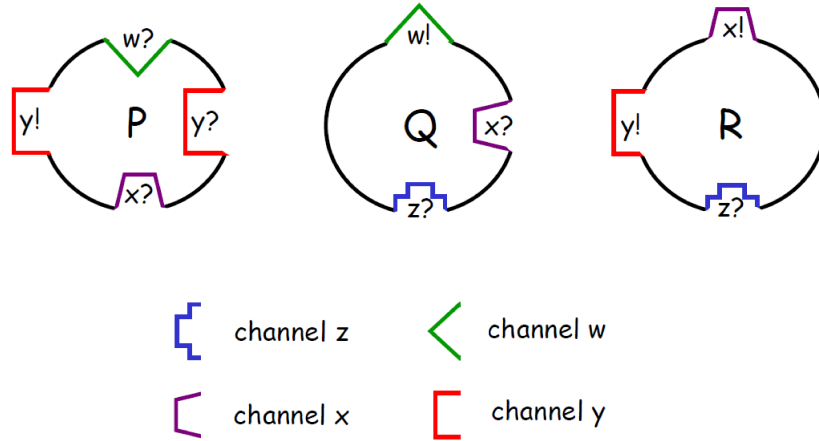


Figure 3.1: Processes (P, Q and R) sharing (w, x, y and z) communication channels, along which they can send (!) or receive (?); adapted from [146]. A P process can interact with an R process or another P process through a y channel.

Initially classes of processes are defined with their collection of channels and associated outcomes. At execution each molecule in the system is a process of one of these classes. An execution step is a non-deterministically chosen communication along a channel between two processes. In the system:

$$geneI := prodI?.(geneI|mrnaI)$$

$$mrnaI := degI?.0$$

$$proteinA := prodI!.(proteinA)$$

a $geneI$ process interacts with the transcription factor $proteinA$ along $prodI$. $geneI$ and $proteinA$ are consumed, producing $geneI$, $mrnaI$ and $proteinA$. Unimolecular processes such as natural degradation are modelled using an auxiliary process such as $aux := degI!.aux$ which is never degraded (because it produces another aux). Thus $mrnaI$ communicates with aux along $degI$ and is degraded (replaced by the null process). Complexes and compartments are represented by the scope of private communication channels, movement by the extrusion of a channels scope

3.5.2 Stochastic π -calculus

In standard π -calculus the system evolves in uniform timesteps with each communication being equally likely irrespective of the number of channels. Such a simulation is semi-quantitative in the same way as a standard Petri net. Stochastic π -calculus (initially proposed as $S\pi$ [152]) enables fully quantitative simulations by associating a rate constant τ with each channel. Complex formation and dissociation provides a clear example of the syntax:

$$a := \text{bind}_{k_1}?.c$$

$$b := \text{bind}_{k_1}!.0$$

$$c := \tau_{k_2}.(a|b)$$

In the second order complexation reaction b communicates with a through the channel bind producing c while b is degraded (0). This occurs on average with the rate k_1 and the propensity of this happening is calculated in the SSA by $\text{prop}(\tau_{k_1}) = k_1 \cdot |a| \cdot |b|$.

First order reactions like dissociation use a special channel type, a stochastic delay (τ) with an associated rate. Here c is replaced by a and b with rate k_2 and propensity $\text{prop}(\tau_k) = k \cdot |a|$.

The model is thus opened to exact stochastic simulation by the Gillespie algorithm [33] and the resultant trajectories directly comparable to that of the equivalent stochastic Petri net/P system.

BioSPI [122], the first stochastic π -calculus simulator, was written in PiFCP [153] a surface syntax which compiles to Flat Concurrent Prolog (FCP) procedures and executed on a FCP Logix platform. BioSpi can simulate systems with 100s of processes to deadlock in the order of seconds [148]. The current leading implementation of a stochastic π -calculus simulator is SPiM [154] developed at Microsoft Research, which maps a model and stochastic simulation algorithm to code in the functional programming language F \sharp .

Whereas Petri nets and P systems model the reactions that occur in a system and record the changes in the populations of each reactant, process calculi model traces each individual reactant. This proves to be computationally expensive when the number of molecules is greater than the number of species, a threshold that is quickly reached. SPiM was recently adapted to a new algorithm [155] that scales with the number of species rather than the number of molecules.

The syntax of π -calculus models and to some extent the compositional nature of the formalism can (for anyone unfamiliar with process algebras) be difficult to understand. A more intuitive understanding is made possible by a graphical representation [156, 157] that visualises the state-space of each process as a graph and has been incorporated into SPiM. In [157] a graphical execution model was defined and proved equivalent to $S\pi$. An example of the graphical representation for π -calculus is shown in figure 3.2.

The $S\pi@$ language

The $S\pi@$ language [94, 95, 96] makes some provision for the stochastic simulation of multiple compartments in π -calculus. $S\pi@$ allows (but does not require) the modeller to specify which compartment a process can accept communications from, and associate volumes with processes that are used to correct the propensity calculations in dynamic volumes. The approach is illustrated with examples of a membrane receptor binding an extracellular signal and inducing a cytoplasmic second messenger, the Na⁺/K⁺ ATPase and finally osmotic pressure.

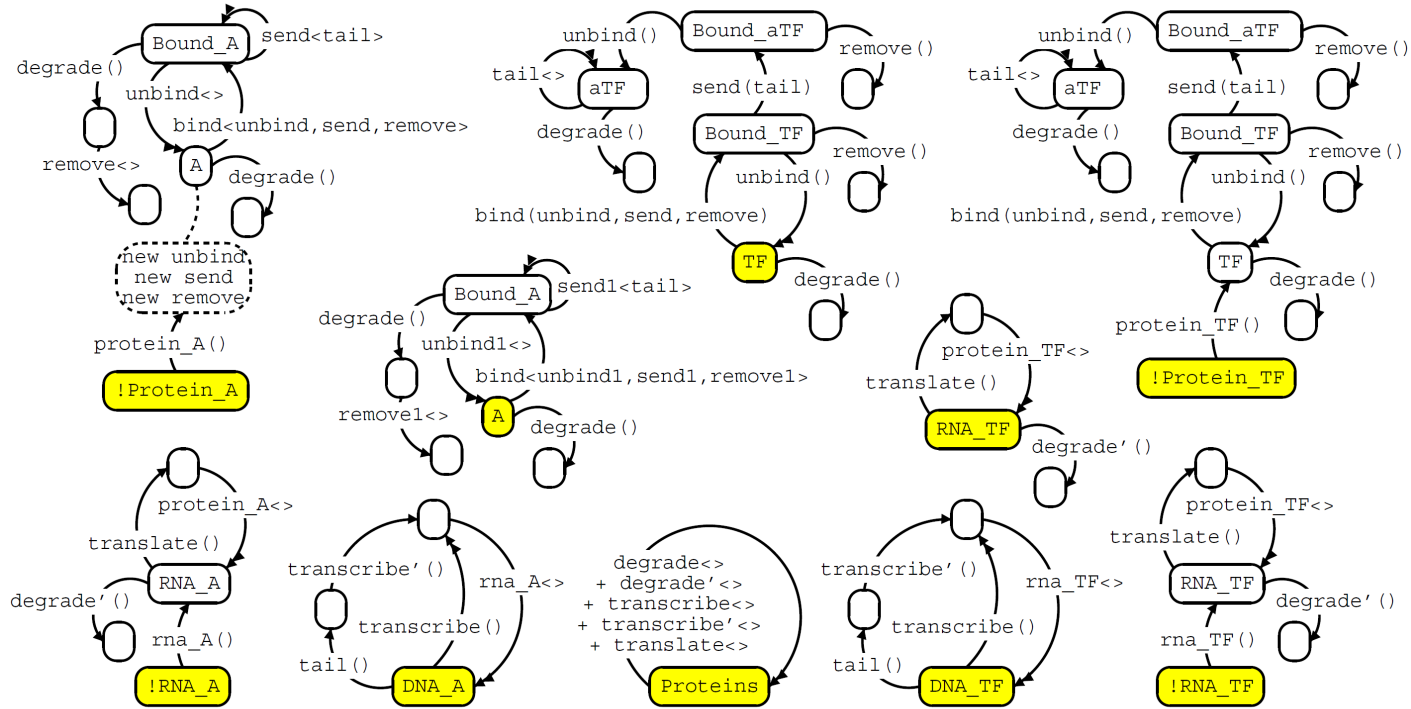


Figure 3.2: Graphical representation of a generic model of positively-regulated transcription in π -calculus. This example, taken from [156], illustrates the compositionality of the formalism where the state-space of each process is visualized as a directed graph of states (only a single instance of each process type is shown whereas during simulation multiple instances of the process currently in state TF would be present). Each edge represents a state transition which is fired either independently of another process (e.g. `degrade()`) or simultaneously when two processes communicate along a channel (e.g. `unbind<>` and `unbind()`).

$S\pi@$ extends the core calculus of SPiM and therefore SPiM programs map directly to $S\pi@$. $S\pi@$ can also represent the atonality of BioAmbients and bitonality of Brane calculi (see sections 3.5.3 and 3.5.4 below) through its core calculus [158].

3.5.3 BioAmbients

It has been acknowledged [146] that using private channels to model compartmentalisation has a number of drawbacks and so BioAmbients [159] were proposed, following on from ambient calculus [160].

An *ambient* is a bounded collection of processes and sub-ambients. An ambient n with p processes and q sub-ambients (m) is denoted $n[P_1 | \dots | P_p | m_1[\dots] | \dots | m_q[\dots]]$. The processes within an ambient instruct it to move through some *capabilities* associated with them. An ambient moves as a whole with its processes and sub-ambients. Ambient capabilities are 3 synchronised pairs: *enter/exit*, *exit/expel* and *merge+/merge-*, which facilitate the movement of an ambient into a sibling ambient, out of a parent ambient, or the merging of two ambients respectively, using named channels similar to process communication.

3.5.4 Brane calculi

Brane calculi [161] extends the reasoning of BioAmbients to model some of the more sophisticated operations occurring at cell membranes using the idea of bitonality [162]. Bitonality requires any two enclosing and enclosed membrane-bound compartments to be of opposite tones, therefore each (paired) Brane calculi operation - *endolexo*, *froth/fizz*, *pinol/phago* and *mitol/mate* - must preserve bitonality. Figure 3.3 shows some of the operations in practice.

3.5.5 Beta-binders and BlenX

Whilst research into π -calculus as a formalism for modelling biological systems is still a very active area of research with many variants being proposed to model specific aspects of the general problem, some parts of the community are seeking to improve on its limitations by initiating a shift in the semantics of the calculus in order to bring it closer the biological reality it hopes to model. They espouse that the fundamental problem of π -calculus in this field is that it was not designed to model biological systems and so the metaphorical relationship between processes and molecules is simply not an appropriate one. For example, copies of processes representing complexes (several processes sharing a private channel) each require a unique private channel on which to communicate, which grows with the number of complexes and is largely irrelevant to what is being modelled. Essentially, complexation and decomplexation are not native in π -calculus, and as a result emergent behaviour has to be programmed which is an error-prone activity. *Beta-binders* [163, 151] sought to resolve this situation by bundling each process into a

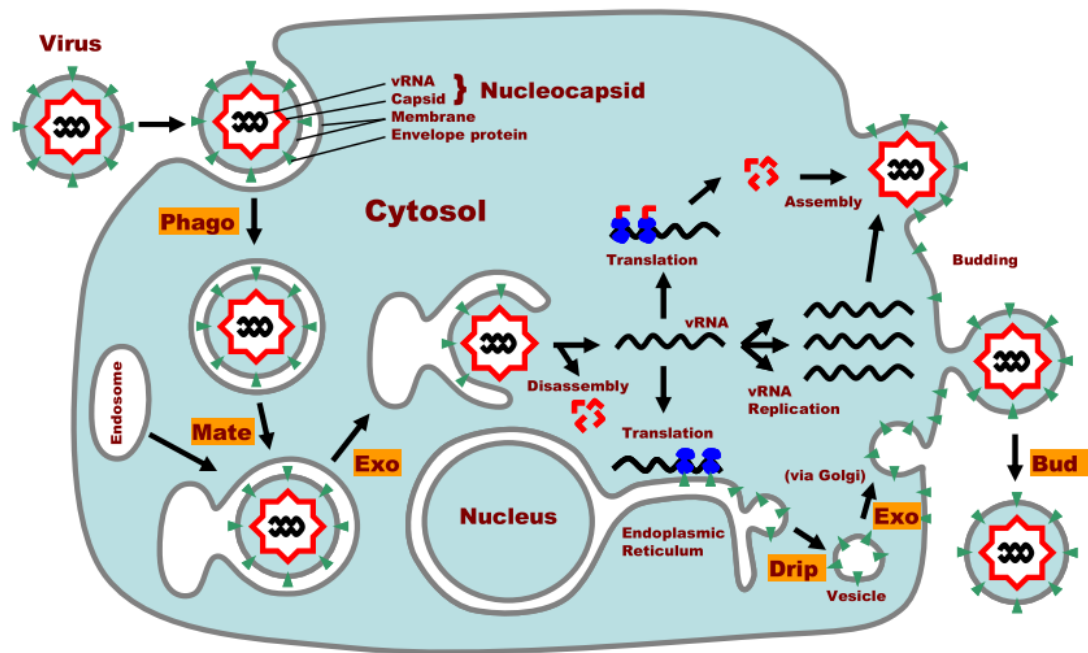


Figure 3.3: Bitonal operations in a Brane calculi model of viral infection by Semliki Forest virus; taken from [161].

binder and introducing new primitives into the calculus: `hide`, `unhide` and `expose`, which update the interfaces between boxes and thus the possible interactions in the system.

BlenX [124, 164] is a language explicitly designed to model biological entities and their interactions which takes up where Beta-binders left off. BlenX introduces several features not found up until now in stochastic process algebras. It uses a *type* file which specifies stochastic rates between interacting types rather than embedded those rates into the model as stochastic constants. The type file first declares the processes by name and then proceeds to associate a rate constant to each pair of interacting processes. More than one rate constant can be associated with each pair which, in a very straightforward manner, allows the complexation of more than two molecules to be modelled, such as the cooperative mechanism of oxygen binding haemoglobin [165]. The decoupling of rates from process definitions simplifies the syntax of process definitions and allows the model to be executed with different rates without changing the model definition, a useful feature when constants might need to be estimated using a genetic algorithms (GAs) for example, where the fitness function of the GA compares simulation traces with laboratory time courses. BlenX also has *events* which are global conditions or perturbations such as a biologist might perform and therefore enable the experimentalists actions to modelled and *in silico* experiments to be performed.

BlenX is supported by a set of tools collectively known as The Beta Workbench (BetaWB or simply BWB) including a stochastic simulator (based on an optimised variant of the Next Reaction Method [59]), bidirectional graphical and textual editors of the model that each reflect

changes made in the other, and a plotter for displaying model execution time courses. A new feature of the plotter is the ability to plot causality, where each simulation event (molecular interaction) is drawn as a box inside the box of the event that led to it. Version 2.0 of BetaWB introduces more primitives: parametric processes, rate functions with variables and constants, continuous time Markov chain generation and SBML export.

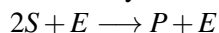
Other prototype tools being developed by Microsoft Research Trento with BlenX support in mind are Snazer, a toolbox for visualising the processes interaction network (similar to graphical π -calculus) and plotting statistics on ensembles of simulation traces (such as the variance in the number of a certain molecular species over time), and KInfer, which performs model and kinetics inference by estimating reactions and rate constants from real concentration data measured at discrete time points. All of the aforementioned tools can be downloaded at <http://www.cosbi.eu/index.php/research/prototypes>.

3.5.6 PEPA and Bio-PEPA

Performance Evaluation Process Algebra (PEPA) is an alternative stochastic process algebra (with roots in CCS) that has recently been applied to modelling signalling pathways [166, 167, 168] and synthetic biology designs [169]. PEPA can be used for reagent-centric and pathway-centric modelling [170]. *Bio-PEPA* [171] is a biologically-oriented modification of PEPA incorporating stoichiometry and the use of kinetic laws in rate functions. Unique rates for each action are kept separate from the process definitions and recorded as functions (as in BlenX). Compartments are currently just a holder for a volume amount that is used in rate calculations. Box 3.5.6 demonstrates the Bio-PEPA syntax for an enzyme catalysed dimerisation reaction.

Syntax for a Bio-PEPA model

In the enzyme catalysed dimerisation reaction:



The substrate S communicates through channel α with a stoichiometry of 2 and both S are consumed by the reaction (vertical arrows below reflect the processes roles):

$$S \stackrel{\text{def}}{=} (\alpha, 2) \downarrow S$$

A P process is produced by the α channel:

$$P \stackrel{\text{def}}{=} (\alpha, 1) \uparrow S$$

An enzyme E is required on the channel but is *not* (\oplus) consumed:

$$E \stackrel{\text{def}}{=} (\alpha, 1) \oplus E$$

From these components we can compose (\bowtie) the model:

$$\text{Model} : (S(150) \bowtie E(10), \alpha \dots)$$

Analysis of Bio-PEPA models is performed using other simulation tools: Bio-PEPA utilizes ODEs solvers in Matlab (and CVODES from SUNDIALS), StochKit and Dizzy to perform

stochastic simulations³ and PRISM as its model checker [172]. PRISM has a nominal upper bound of 10,000,000 states when computing the CTMC. Bio-PEPA works around this limitation by discretizing the range of each continuous variable (molecular concentrations) [173] to reduce the numbers of states to a just a few, depending on the chosen granularity (estimated from stochastic simulations, which must be the same for all interacting processes). This semi-quantitative approach to modelling reflects the number of observations that can realistically be made by biologists and enables granular coverage of larger systems than would normally be amenable to model checking.

3.6 Rule-based approaches

A set of rules is another formal object that can be analysed mathematically (statically) to discover the relationships between rules, between their participants, potential sequences of events and the reachability of certain states. Rule-based formalisms are generally considered to be an appropriate level of abstraction and syntactic complexity to enable effective communication between biologists and computer scientists. The following are some of the rule-based formalisms that have been applied to the modelling of biological systems, at the level of molecular interactions.

3.6.1 κ

κ -calculus, or simply κ , was originally developed with the purpose of representing protein interactions [174]. It is a rule-based modelling approach similar to BioNetGen (BNG) [175]. κ aims to overcome what has been described as “the barrier of objects” [176]; where it is the properties of the entities comprising the systems that creates the dynamics. In practice κ treats macromolecules as individual agents, so that any two molecules of a certain protein can be considered the same, but in different states. Each molecular *agent* has labelled *sites* with states that represent its interaction capabilities or *interface*. An agent of 007 with three sites (x , y and z in states 1, 1 and 3 respectively) can be expressed in κ as: $007(x \sim 1, y \sim 1, z \sim 3)$. Complexes are modelled as agents with linked sites: $A(x \sim 1, y \sim 1, z \sim 3!0), B(u \sim 2!0, v \sim 1)$; in this complex agent A is linked to agent B through sites z and u , the link is called 0.

The elementary actions in κ comprise the modification of the state of a site, binding and unbinding of two agents sites, and the introduction or deletion of agents. These are effected by context-free rules that rewrite the system locally according to patterns of partially specified agents or complexes, with the intention of “don’t care, don’t write”. For example the pattern

³There is some overlap in the functionality of these package; the main differences being that Dizzy offers a GUI for parameter adjustment and model visualisation while StochKit offers a wider range of algorithms including the slow-scale SSA. Overlapping analyses are used to identify bugs in the pipeline.

$A(x, y \sim 1, z \sim 3)$ applies to any A agents with sites y and z in states 1 and 3, irrespective of the state and boundness of site x . The generalisation of rules over many possible instances of agent states in κ allows models with very large state spaces to be expressed in remarkably small sets of rules.

A “*fully scalable*” [177, 178], with respect to the number of different molecular species, stochastic simulator for κ has been developed using a generalisation of Gillespie’s Continuous Time Markov Chain (CTMC) scheme. Scalability is achieved because at no point is the set of possible complexes generated (no flattening of the rules), similar to the concept of species generated on-the-fly in Molecuizer [179]. The dominant time complexity is $O(\log M)$ where M is number of rules. This simulation methodology is intimately tied to the formalism, which unfortunately does not account for other biological phenomena influencing reaction networks, notably compartmentation. Another, non- κ , rule based language and simulator that can account for compartments is the Stochastic Simulation Compiler (SSC) [180]. In SSC’s input format space is modelled using solid constructive geometry, and species with properties are interconverted by pattern-based rules. Stochastic simulations of these models are performed by expanding (flattening) rules into a stochastic model, generating SSA code that is model-specific in C, compiling and running the model/simulation as a native executable.

3.6.2 P systems

P systems [181] are a natural computing paradigm that take inspiration from the structure and information processing capabilities of living cells to solve computationally hard problems (“*trading time with space*” [182]). P systems originated in formal languages and computability theory and majority of research, known as *membrane computing*, has focused on investigating the computational power and proving the Turing equivalence of P systems variants. P systems have subsequently been applied to the modelling of biological systems [25, 183, 184], with the rationale that as an abstraction of cellular organisation they should be suitable for modelling the compartmentation of biological pathways.

A canonical P system [185] is defined by a hierarchical membrane structure delimiting regions containing multisets of objects and multiset rewriting rules (figure 3.4). The outer membrane of a P system is often referred to as the *skin* membrane. Objects transported out of the skin enter the *environment*.

The P system *evolves* by the repeated application of rules on the multisets, mimicking chemical reactions and transportation across membranes, and halts when no more rules can be applied. The original execution model of P systems is maximally-parallel and non-deterministic [181], meaning at each step all objects which can evolve do so, with randomized tie-breaking. In practice all the rules that can be applied (for which there are sufficient objects in that region)

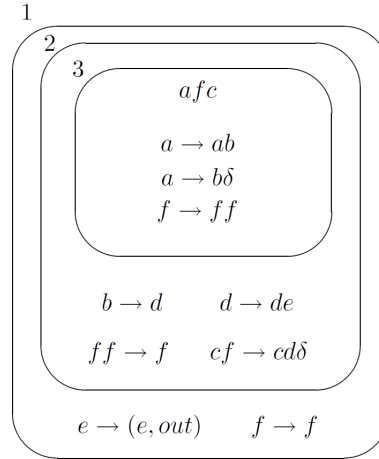


Figure 3.4: The initial configuration of a P system, from [186].

are pooled, one rule is selected from these at random and applied exhaustively in its respective membrane. The affected objects are removed from this step of the computation and retained for the next step. The pool of applicable rules is recalculated and another rule is chosen randomly and applied. This process repeats until no more rules can be applied. Any remaining objects are carried over to the next step. The system clock then advances one step and the execution starts afresh. By this method an object may be used only once in each step.

Maximal parallelism is well suited to discrete systems with synchronous components and is of major importance to proving the computational power of P systems through acceptance checking. However it is less well-suited to biological systems where asynchronous events occur in continuous time [187]. The result of a P system computation is often taken to be the multiset found in the environment when execution has halted.

Many formal definitions for a P system can be found in the literature. At its simplest a P system of (degree m) may be formally defined as

$\Pi = (\Sigma, \mu, w_1, w_2, \dots, w_m, R_1, R_2, \dots, R_m)$, where:

- Σ is a finite and non-empty alphabet of *objects*;
- $L = \{l_1, \dots, l_k\}$ is a finite alphabet of symbols representing *labels* for the compartments and identifying compartment types;
- μ is a *membrane structure* containing m membranes indexed $1, 2, \dots, m$. The membrane structure can be represented formally as a rooted tree but can also be depicted as a non-intersecting Venn diagram or nested pairs of indexed square brackets, e.g. $[_1 [_2]_2 [_3]_3]_1$;
- w_1, w_2, \dots, w_m are the initial *multisets* of objects present in regions $1, 2, \dots, m$ of the membrane structure, written in the form a^3b^2c ;

- R_1, R_2, \dots, R_m are finite sets of rewriting rules associated with regions $1, 2, \dots, m$.

A common difference to the definition given above is to define a set of labels for membranes and to associate sets of rules with labels, so that the rules a membrane contains are defined by its label. Rewriting can be said to be regulated because rules are localised. Rules that rewrite the multiset of the compartment in which they reside are termed *evolution* rules, rules that rewrite the multisets in enclosed or the enclosing compartment are termed *communication* rules. Both can be represented by the syntax:

$$u \longrightarrow (v, tar) \quad (3.1)$$

where $u, v \in \Sigma$ are multisets of objects from the alphabet Σ and $tar \in \{here, in, out\}$ is the locally evaluated target membrane. Where two or more enclosed membranes exist the target of *in* would be chosen non-deterministically. By associating different targets to different objects symport and antiport membrane transporters can easily be modelled [188].

The membrane structure of a P system is not intended be static. The basic formulation includes a special δ operator that effects the dissolution of membranes, with movement of the contents into the containing membrane, analogous to the disappearance of the nuclear membrane in prometaphase of mitosis.

As the field of P systems has grown researchers have sought to augment these devices with new properties and capabilities such as:

- *permeability*, where δ does not dissolve the membrane but decreases its “thickness” by one unit. For membranes with a thickness of 1, δ functions normally. τ is used to increase the thickness of membranes by 1 unit. A thickness of 2 is considered the maximum and is impermeable. δ and τ applied together cancel out;
- *active membranes*, with rules for operations on membranes such as division, fusion, endocytosis and exocytosis affecting the internal structure of the P system (figure 3.5) [189];
- *gemmation of mobile membranes* [190] (inspired by mobile ambients [160]), where new membranes are created and translocated between membranes, analogous to the budding of vesicles from the Golgi complex;
- *multisets of strings and rewriting rules on strings* which rewrite substrings and produce new strings, analogous to transcription and translation;
- *promoters and inhibitors*, written $u \longrightarrow v \mid z$ and $u \longrightarrow v \mid \neg z$ respectively, where all promoter objects in the multiset z must be present and conversely none of the inhibitor objects in $\neg z$ can be present for the rule $u \longrightarrow v$ to be applied. Promotion can equally

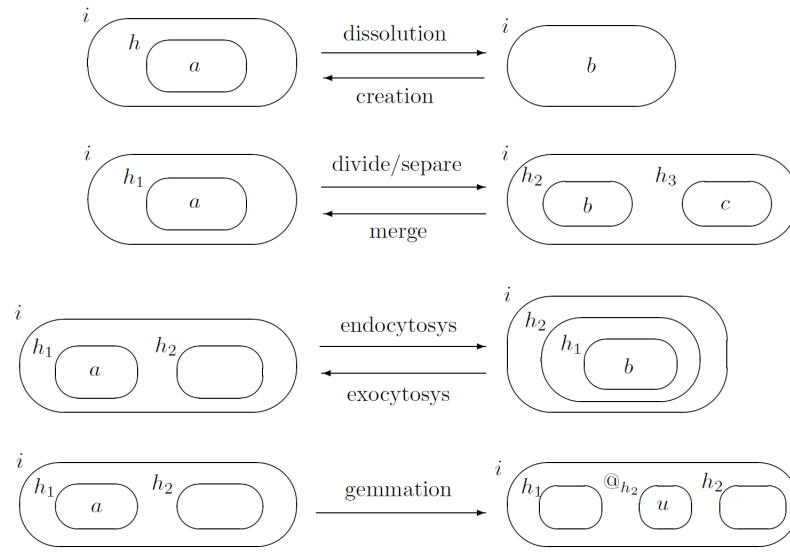


Figure 3.5: Operations of P systems with activate membranes, from [186].

be achieved with catalyst rules ($ac \rightarrow bc$) but the syntax is not so explicit of the objects roles;

- *boundary rules* [191], which can 'see' externally, written $xa [{}_j by \rightarrow xc [{}_j dy$ meaning that when multisets x and y are outside and inside the membrane the multisets a and b can be rewritten to c and d respectively. Boundary rules allow the fetching of objects from the environment. Alternatively, evolution-communication rules can use the target *come* as in $a \rightarrow (a, come)$ to bring objects from the environment into the skin;
- *teleportation*, where for in_j , j can be any label. in^* signifies the deepest membrane and out^* the skin;

P system variants

Other variants seek to evaluate the computational power of compartmentation using alternative membrane structures to a single tree while retaining only the basic P system rules types. Some of the most well-studied are:

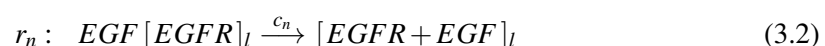
- *Tissue P systems* which abstract the arrangement of cells in tissues with objects being exchanged between membranes of the same tree depth along channels (formally a graph) [192];
- *Population P systems* which extend tissue P systems by introducing rules that reorder the connectivity of membranes by making and breaking channels, to emulate cell division and differentiation [193, 194, 195];

- *P colonies*, a combination of P systems and eco-grammar systems with unconnected membranes interacting through the environment [196];
- *Spiking Neural P systems* which abstract the organisation and behaviour of neurons to investigate the role of structure in the output (timing and sequence) of spikes [197].

Quorum Sensing P systems are perhaps the clearest example of the computation versus modelling dichotomy of P systems. In quorum sensing (QS), discussed more fully in section 1.1.1, bacteria constantly export diffusible chemical signals at a low level. Uptake of those signals leads to upregulation of signal production resulting in a mechanism that allows bacteria to sense the population density in which they find themselves. Once a sufficiently high density of bacteria, or quorum, is achieved other sets of genes are switched on initiating a change in behaviour at the population level [26]. Through the abstracted mechanisms of bacterial quorum sensing QS P systems are shown to be equivalent in power to Turing machines by simulating a counter machine [29]. At the same time P systems have been used to model quorum sensing in the bacteria *Vibrio fischeri* [32] and *Pseudomonas aeruginosa* [198, 28, 29].

For biological modelling with P systems maximally parallel execution creates two inaccuracies: (1) reactions do not occur at the correct rates and (2) all time steps are equal. A number of attempts to constrain the maximal parallelism have been developed, including bounded [199, 200] and minimal parallelism [201], priority relations for rules with strong and weak interpretations, even energetic constraints. But the simulation of biological systems “demands biologically meaningful evolution strategies” [202] with rules that reflect the reality of chemical interactions. The modelling of biological systems using P systems has divided into several parallel tracks based on the method of execution, and these tracks can be categorised as stochastic-discrete, deterministic-discrete and deterministic-continuous in their treatment of rule application and object quantities operating in continuous time.

Stochastic P systems Stochastic simulation algorithms offers the ability to produce realistic trajectories of discrete molecular populations that can be directly compared to laboratory data. *Stochastic P systems* become executable specifications of biological systems by assigning a stochastic rate constant c to each rule (e.g. 3.6.2), and applying rules according to a stochastic simulation algorithm.



The rule syntax, inspired by boundary rules [191], emphasises the presence of membranes across which objects can translocate. Depending on the label of the membrane where the rule is applied, this compact rule form can specify the movement of molecules from outside into that membrane, or, if l corresponds to an enclosed membrane, the movement of objects in and out of that.

To select the next rule to apply across all compartments using SSA the hierarchical membrane structure of the P system can be translated to a flattened representation where the objects of the same type that are in different compartments are treated as separate species. Alternatively, the SSA can be extended to multiple compartments by using the normal method in each compartment but the scheduling rule application appropriately. The Multi-Compartmental Gillespie (MCG) algorithm [25], implemented in SciLab and C [203] by Infobiotics developer Francisco Romero Campero takes the second approach. As for any SSA the propensities of each reaction are computed (from the stochastic rate constant and number of possible combinations of reactants) and used to bias the random selection of the next rule to apply and a waiting time, **but independently for each compartment. The waiting times of all compartments are then sorted and the selected rule in the compartment with the shortest waiting time is applied.** That time is added to the system clock and subtracted from the other waiting times. Any compartments that are affected by the application of the selected rule because the multiplicities of the reactants have changed have their rule selection and waiting times recalculated. The waiting times are sorted, the next rule is applied and so on.

The Infobiotics Workbench simulator MCSS implements an optimised Multicompartment Gillespie Algorithm with Queue [204], pseudocode for which is shown below. The algorithm uses an *indexed priority queue (binary heap)* to schedule the compartments according to the waiting time of the next rule to be applied. The queue ensures that the compartments are always sorted by ascending waiting time, substantially improving performance for models with many thousands of the compartments by reducing the computational complexity of the scheduling operation. The optimised algorithm is equivalent to the standard SSA and both implementations pass the Discrete Stochastic Models Test Suite [205]. We note that these *exact* stochastic simulation algorithm are necessarily sequential and therefore cannot make use of the potential parallelism of compartmentation that was considered key to the power of P systems.

Membrane Systems with Peripheral and Integral Proteins *Membrane Systems with Peripheral and Integral Proteins* [206, 207] are P systems which model proteins attached to or embedded in the membrane by associating multisets with membranes as well as the regions they define, and extending the rule syntax to handle the extra multisets. They are implemented in the software Cyto-Sim [208, 209] and simulated with a proprietary extension of Gillespie.

Algorithm 1: Multicompartment Gillespie Algorithm with Queue

```

begin
  // preprocess
  // calculate waiting time for each compartment
  for  $i \leftarrow 1$  to number of compartments do
    // perform Gillespie Direct Method
     $(\tau, \rho) \leftarrow \text{GillespieDirectMethod}(i)$  // add waiting time to queue
    QueueInsert( $\tau, i, \rho$ )

  // main loop
  while  $(\tau, i, \rho) \leftarrow \text{QueuePeek}()$  do
    if  $\tau > t_{\max}$  then
      halt
    // advance simulation time
     $t \leftarrow \tau$  // apply rule  $\rho$  in compartment  $i$ 
    ExecuteRule( $\rho, i$ ) // calculate waiting time for compartment  $i$ 
     $(\tau, \rho) \leftarrow \text{GillespieDirectMethod}(i)$  // add waiting time to queue
    if  $\tau = 0$  then
      // no rule applicable
      QueueDeleteHead()
    else
      // replace waiting time
      QueueReplaceHead( $t + \tau, i, \rho$ )

    // update target compartment of rule
    if rule  $\rho$  in compartment  $i$  is a translocation rule then
      set  $j$  to index of target compartment of rule  $\rho$  // calculate waiting time for compartment  $j$ 
       $(\tau, \rho) \leftarrow \text{GillespieDirectMethod}(j)$  // add waiting time to queue
      if  $\tau = 0$  then
        // no rule applicable
        QueueDeleteEntry( $j$ )
      else
        // replace waiting time
        QueueReplaceEntry( $t + \tau, j, \rho$ )

```

Cyto-Sim can also accept as input Petri net matrices and is said to cover a large part of the SBML specification [210].

Dynamical Probabilistic P systems *Dynamical Probabilistic P systems* (DPP) [211] were originally motivated by the investigation of maximal parallelism in nature, using standard P systems with a novel rule application method to model biological phenomena in a discrete and stochastic way. In procedure analogous to the Gillespie algorithm propensity calculation DPP rules are dynamically assigned a probability that is the product of the possible combinations of reactant objects and an associated rate. Probabilities are normalised to their sum, a random number 'tossed' and rules selected from the roulette wheel are assigned objects from the multiset until all objects have been assigned, such that "the rules with the highest normalized probability value will be more frequently tossed" [212]. Parallelism can be bounded in DPP by the introduction of 'mute rules' such as $a \longrightarrow (a, \text{here})$ which do not change the multiset but participate in the tossing process.

The approximate τ -leaping SSA has recently been applied to the parallel implementation of DPP [90] in which the computational complexity of the algorithm increases linearly with the

number of reactant species. τ -leaping DPP traces the simulated time of the compartments as well as the time line of the system as a whole. In each membrane a leap time τ is generated, based on its current state. The smallest τ is used to generate probability distributions for each membrane and the system leaps to the next state by applying several rules in each membrane (the order of execution does not matter). Thus the system's evolution is synchronised by the smallest leap and time advances in increments of τ .

τ -DPPs have been used to model Ras protein cycle, activation of adenylate cyclase, production of cyclic AMP, regulatory elements in the activation of cAMP-dependent protein kinase in yeast [213], and predator-prey metapopulations [212] with a weighted undirected graph representation of spatial patches corresponding to membranes.

Metabolic P systems

Metabolic P (MP) systems have diverged considerably from the discrete and non-deterministic, compartmentation-based philosophy of P systems, and would warrant a separate section if not for their limited adoption.

The primary innovation of MP systems is metabolic algorithm (MA) [214], a **continuous and deterministic** modelling approach with reaction rules that transform moles of substances as opposed to molecules over a series of equal timesteps. Time in MP systems is mapped to real time by *multiplication* of a constant value, τ , representing the sampling intervals from real-world observations. The set of reactions is described by a stoichiometric matrix [215]. The degree of transformation (number of times a rule is applied at each step) is controlled by a set of flux regulation maps which determine the number of times to apply each rule. Fluxes are calculated as a function of the substance amounts, global constants and parameters such as temperature, pressure and pH, which can also change as a function of time. The overall effect is to produce the same results as an ordinary differential equations model by repeatedly applying a set of stoichiometric equations on a pool of substances according to a set of functions.

In contrast to other P systems, MP systems have just a single membrane. P systems models with membranes can be flattened to a single membrane where "localization is now encoded into symbols" [216] and rules are altered accordingly. Essentially, pools of substance are renamed with subscripts denoting the membrane they reside in. With a deterministic execution model and without of hierarchy of membranes MP systems have been proved equivalent to hybrid functional Petri nets [217].

However the visual representation, an MP graph, is not a Petri net. MP graphs have five node types: substances, reactions, fluxes, parameters and IO gates. Edges representing the transformation of substances have solid lines annotated with stoichiometries. Edges representing the

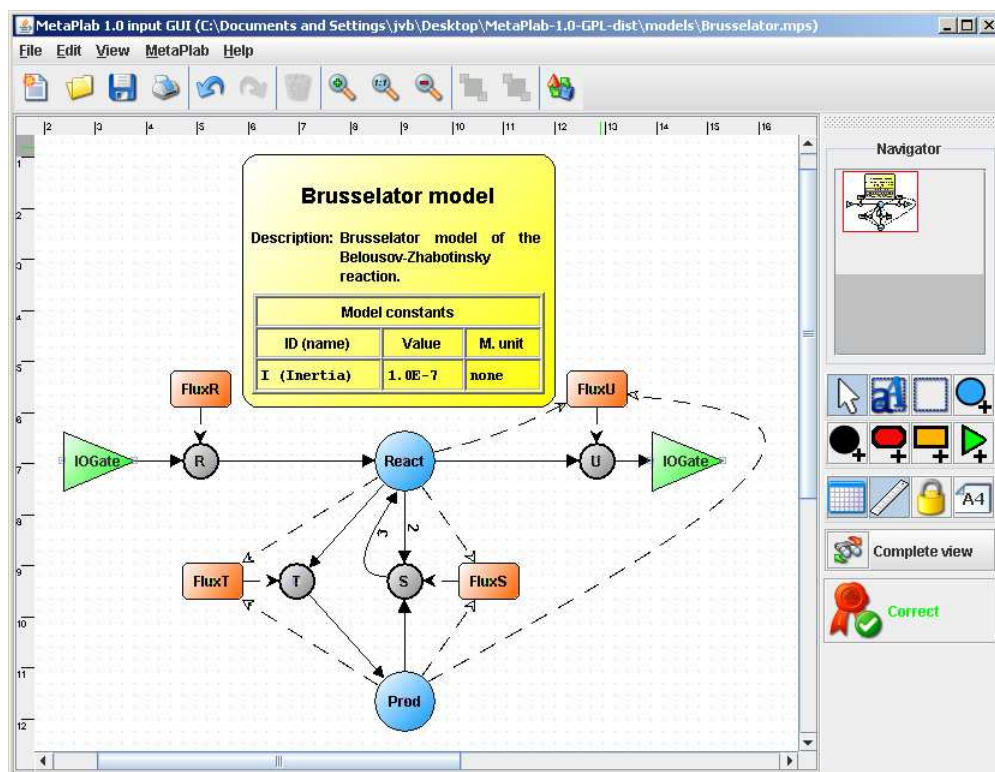


Figure 3.6: Editing an MP graph with MetaPlab.

influence of one node on another have dashed lines. Direct manipulation of MP graphs is supported by the MetaPlab [218] software, previously Psim [202]. Figure 3.6 shows the interface of MetaPlab with an example model.

MetaPlab uses plugins to interact with the underlying model run simulations and optimise parameters. By exposing the model to developers it is hoped that new plugins will be developed that extend the functionality and relevance of the software to more complex problems. A simple HTML plugin - that outputs a webpage containing a table of model parameters - acts as an example for developers. The simulation plugin creates 'vistas' (plots) of substance quantities over time, that are mapped back on to the model. This is made possible because the simulations are deterministic and therefore produce a single solution to the model dynamics.

3.6.3 MGS

MGS⁴ is a programming language that embodies several spatial computing concepts applied to the specification and simulation of dynamical systems with a dynamical structure [219]. MGS extends the notion of rewriting rules by considering the more general structure of *topological collections*. A topological collection is a set of elements structured by a neighborhood relationships that represent a spacial data structure, with optional constraints, manipulatable by

⁴<http://mgs.spatial-computing.org/>

computation, including Delaunay triangulations and multisets [220]. Rules in MGS replace some subcollection of a topological collection with another topological collection. This process is said to offer a unifying view on some computational mechanisms initially inspired by biological processes: L-systems, P-systems and cellular automata. Indeed, MGS has been used as a host language for stochastic P systems [221] in which the authors also implemented a multi-compartmental stochastic simulation algorithm orthogonal to our own (outlined above). This implementation of stochastic P systems were used to model the Lotka–Volterra auto-catalytic system, and the life cycle of the Semliki Forest virus. MGS has also been applied to the modelling of the λ phage genetic switch [222].

The ideas behind MGS will undoubtedly be useful as our ability to map biological systems to computational structures grows. Unfortunately, such generality comes at the cost of much greater complexity in the definition of systems and transformational rules. The functional legacy of the language (the interpreter is written in OCaml) means that we do not feel confident in implementing a moderately complex model in MGS.

3.7 Conclusions

The primary purpose of this review was to examine existing computational modelling formalisms that have been applied to biological systems and decide which, if any, are appropriate specification languages for multicellular biomodels. A secondary consideration, which we address first, is whether a particular specification language restricts or permits certain types of *in silico* experiment. Qualitative approaches, boolean networks for instance, cannot produce timeseries of species quantities comparable to experimental data. On the other hand, boolean networks and some Petri nets can reveal whether a particular state is reachable from another, thereby providing one of the central benefits of model checking.

Of the various computational modelling approaches reviewed, several including stochastic P systems, MGS, κ , Bio-PEPA, stochastic π -calculus (i.e. formalisms that are not systems of mathematical equations), contain sufficient information to permit, with some mapping, each the *in silico* experiments discussed in chapter 2. Stochastic simulation is a commonplace execution strategy for computing their dynamics because the mapping from interactions to bimolecular reactions is often quite direct. Automated verification techniques can be applied to most by reformulating the model or adapting its simulated output to the input format of a particular model checker: a Reactive Modules [223] description in the case of PRISM, sets of timeseries for MC2 [224]. Continuous variables, e.g. concentrations, can be discretized at a chosen granularity [173] facilitating model checking with approximate results. Parameter optimisation requires some formalism-dependent wrapping for model parameterisation, simulation and feedback into a search strategy, but the search and optimisation methods may be taken off-the-shelf. Model

structure optimisation is more challenging because it involves model generation under biological and formalism dependent constraints.

Regarding software support, most formalisms have simulators, some use model checking (P systems and Bio-PEPA can use PRISM, Petri nets MC2), others can optimise parameters (Kinfer for BlenX, MetaPLab log-gain plugin) and a few have experimented with model optimisation. None are covered by simulation, model checking, parameter *and* model structure optimisation. Despite varying degrees of software support we see these approaches as being largely equivalent in how they could be processed and tuned. What they offer in terms of model building and interrogation is much more important. To evaluate this we consider the usability of the language and the tools for working with it: by whom will it be used and for what? Ideally biologists would be both producers and consumers of models (although we realise that in the near term it is more likely that computer scientists will be responsible for formalising biological knowledge and running simulations). For models to be of any value they must be understandable (usable as communication devices), capture the required features and generate answers that are interpretable.

We found that with some of these formalisms the resulting models were less understandable than the systems they were trying to model and therefore too difficult to debug or adapt by biologists. In particular, we consider process algebras to be too abstruse to reach a wide audience.

Others were limited to certain domains, capturing a selection of features in detail. For example, κ effectively models signalling cascades with many possible protein-protein complexes using graph rewriting, but currently has no notion of compartments. BioAmbients and Brane calculi explicitly model compartments but need other process calculi to model biochemical reactions. SBML and CellML are predominantly aimed at continuous mathematical models and consequently an awkward fit for stochastic models that avoid the limitations of macroscopic techniques.

SBML allows many compartments to be specified, it requires that the species and the reactions in each compartment are unique with respect to the entire model, so that each instance of a compartment with identical sets of species and reactions (e.g. individual bacteria of the same strain) must be specified repeatedly. This approach to specification cannot scale for models with increased numbers of the similar compartments such as bacterial colonies. The modular nature of CellML avoids this problem by allowing compartments to be defined once and duplicated many times.

We conclude that none of the reviewed formalisms are perfectly suited to our modelling goals. The main deficiency is, with the notable exception of MGS (which we discount on the issue of usability), the inability to specify a spatial arrangement of compartments, such that neighbourhood relationship would be automatically established. That this was not the case may account

for why multiscale molecular and multicellular models are underrepresented in comparison to single cell models: they are fewer because to build such models implies creating a new modelling framework, incompatible with existing tools.

We see stochastic P systems as the most suitable for formalism-agnostic biologists, as rules incorporating reactions and transport across membranes provide a clear and localised means of expressing intercellular communication. Therefore, to support the modelling of the multicellular systems introduced in the chapter 1 and the modelling principles developed in chapter 2, we have chosen to create our own biomodel specification formalism that extends stochastic P systems with discrete-space using lattices to capture the geometric arrangement of cells population, and develop scalable simulation algorithms, interfaces to model checkers that transparently convert models and timeseries into the appropriate formats. This novel executable biology formalism is presented in next chapter.

Chapter 4

Lattice Population P systems

Chapter abstract

This chapter introduces Lattice Population P systems, the core abstraction of a biological system used by the Infobiotics Workbench to model spatially discrete multicellular systems. We formally define these objects and show how they translate into several machine-readable data formats. Lastly, we look at how they facilitate rapid model building through different kinds of reusability.

4.1 Introduction

We derived the requirements for our modelling approach and software by focusing on two multicellular biological systems of particular interest to our group at the University of Nottingham: the quorum sensing system in pathogenic bacteria *Pseudomonas aeruginosa* and the root node of the plant *Arabidopsis thaliana* (discussed in section 1.1.1). Both systems have a strong spatial component where molecule exchange between *adjacent* cells and the arrangement (in addition to the state) of cells determines the overall phenotypes. However, this structure cannot be captured by our chosen formalism of stochastic P systems, which have only a hierarchical membrane structure of compartments within other compartments. Therefore it was crucial to augment stochastic P systems with an additional level of organisation, a 2-dimensional geometric lattice on which a population of P systems (model cells) could be placed and over which molecules could be translocated. Rules that move objects from one P system (applicable only in the outmost compartment) to another on the lattice are associated a vector that describes how many positions in the x and y dimensions to put that molecule. We call this extension of stochastic P systems **Lattice Population P systems** [225] (LPP systems for short) and, in the tradition of P systems, proceed with their formal definition (published in [102]).

Each cell type with its compartmentalised structure, characteristic molecular species and molecular processes is represented using a *SP system* according to definition 4.1. The rules of each *SP system* are possibly specified in a modular way according to definition 4.4. The spatial distribution of cells in the population is represented using a *finite point lattice*, definition 4.2, and finally

different copies of the corresponding *SP system* representing each cell type are distributed over the points of the lattice according to the spatial distribution of an *LPP systems* in definition 4.3.

4.2 Formal definitions

Definition 4.1. Stochastic P system

A stochastic P system, SP system for short, is a formal rule-based specification of a multicompartmental and discrete dynamical system with stochastic semantics given by a tuple:

$$SP = (M, \mu, L, I_{l_1}, \dots, I_{l_n}, R_{l_1}, \dots, R_{l_n}) \quad (4.1)$$

where:

- M is a finite set (alphabet) of objects specifying the entities involved in the system (genes, RNAs, proteins, etc.).
- μ is membrane structure composed of $n \geq 1$ membranes defining the regions or compartments of the system. Membranes are arranged in a hierarchical manner, i.e. membranes can be inside other membranes. There exists an outermost membrane termed the *skin*, not contained in any other membrane, which defines the system itself acting as the boundary. The membrane structure can be represented graphically using a Venn diagram, a list of matching square brackets or formally as a rooted tree where each node represents a membrane, the root stands for the skin membrane and the relationship of a membrane being inside another one is described by the node representing the first membrane being the descendant of the node for the second membrane (shown in figure 4.1).
- $L = \{l_1, \dots, l_n\}$ is finite set of labels naming compartments in a one-to-one manner (nucleus, cytoplasm, etc.).
- I_{l_k} for each $1 \leq k \leq n$ is the initial condition of the compartment or region defined by membrane k consisting of a multiset of objects over M describing the initial numbers of the various molecular species present in the corresponding compartment.
- $R_{l_k} = \{r_1^{l_k}, \dots, r_{m_{l_k}}^{l_k}\}$ for each $1 \leq k \leq n$ is a set of multiset rewriting rules describing the interactions between the molecules, such as complex formation and gene regulation. Each set of rewriting rules R_{l_k} is specifically associated to the compartment identified by the label l_k . These multiset rewriting rules are of the following form:

$$r_i^{l_k} : o_1 [o_2]_l \xrightarrow{c_i^{l_k}} o'_1 [o'_2]_l \quad (4.2)$$

where o_1, o_2 and o'_1, o'_2 are multisets of objects (possibly empty) over M representing the molecular species consumed and produced in the corresponding molecular interaction. The square brackets and the label l describe the compartment involved in the interaction. An application of a rule of this form changes the content of the membrane with label l by replacing the multiset o_2 with o'_2 and the content of the membrane outside by replacing the objects o_1 with o'_1 . A stochastic constant $c_i^{l_k}$ is specifically associated with each rule in order to determine the probability of applying the rule and the time elapsed between rule applications according to the stochastic semantics of Gillespie's theory of chemical kinetics [226]. This also places a limit on the size of the multisets, that the sum total of members in the multisets o_1 and o_2 be 2 at most.

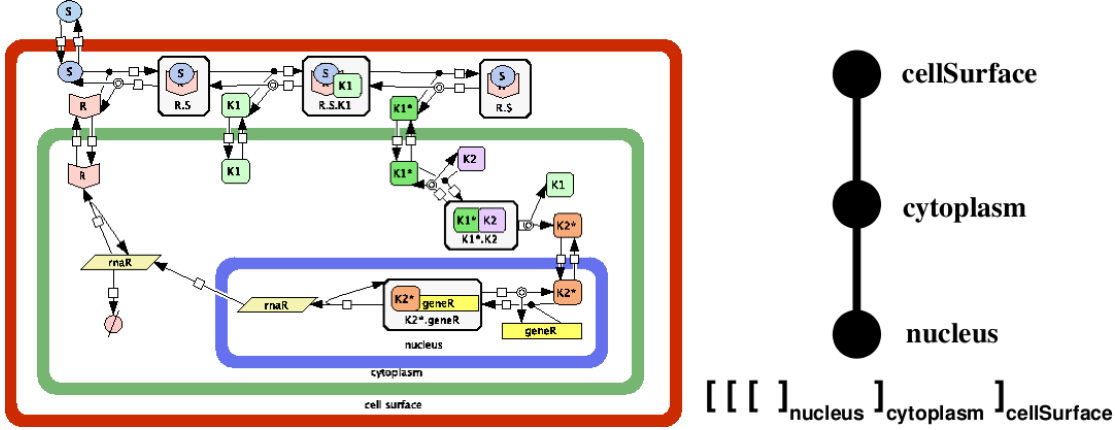


Figure 4.1: A graphical representation using CellDesigner of a *SP system* modelling a simple transduction system where the signal S induces the synthesis or more receptors R . This *SP system* consists of a set of 12 objects representing molecular species; 3 compartments identified with the labels *cellSurface*, *cytoplasm* and *nucleus*; and 25 rewriting rules represented with arrows describing the molecular interactions like active nuclear translocation and gene transcription. The three different possible representations of the membrane structure, namely, as a Venn diagram, list of matching square brackets and rooted tree, are also presented.

Definition 4.1 provides the formalism needed for the specification of an individual cell as shown in figure 4.1. To specify the possible spatial distribution of cells assembled into colonies and tissues we define an array of regularly distributed points according to a *finite point lattice* or *grid* [227].

Definition 4.2. Finite Point Lattice

Given $B = \{v_1, \dots, v_n\}$ a list of linearly independent basis vectors, $o \in \mathbb{R}^n$ a point referred to as origin and a list of integer bounds $(\alpha_1^{min}, \alpha_1^{max}, \dots, \alpha_n^{min}, \alpha_n^{max})$ a finite point lattice, lattice for short, Lat in \mathbb{R}^n denoted as:

$$Lat = (B, o, (\alpha_1^{min}, \alpha_1^{max}, \dots, \alpha_n^{min}, \alpha_n^{max})) \quad (4.3)$$

is the collection of regularly distributed points, $P(Lat)$, defined as:

$$P(Lat) = \{o + \sum_{i=1}^n \alpha_i v_i : \forall i = 1, \dots, n (\alpha_i \in \mathbb{Z} \wedge \alpha_i^{min} \leq \alpha_i \leq \alpha_i^{max})\} \quad (4.4)$$

Given a *finite point lattice*, Lat , each point $x = o + \sum_{i=1}^n \alpha_i v_i \in Lat$ is uniquely identified by the coefficients $\{\alpha_i : i = 1, \dots, n\}$ and consequently it will be denoted as $x = (\alpha_1, \dots, \alpha_n)$.

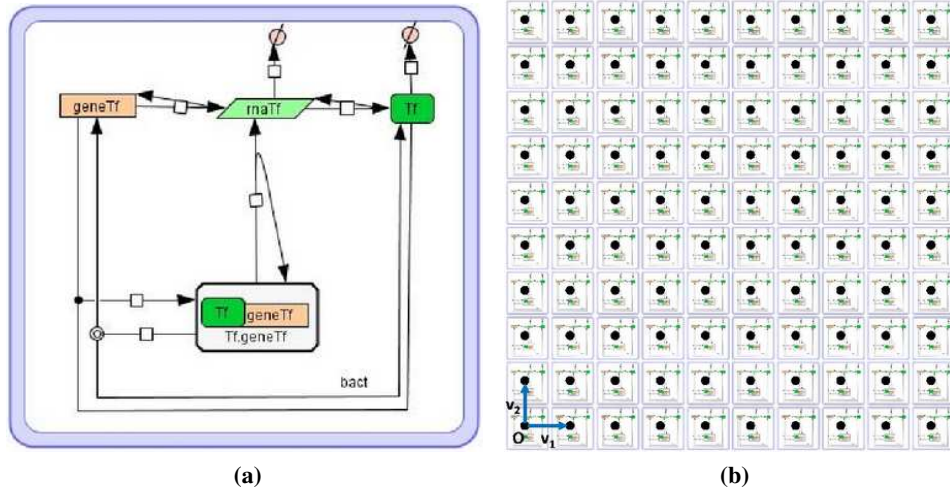


Figure 4.2: SP systems containing reactions of a gene regulatory network, single (a) and distributed over the LPP system lattice (b).

SP systems are distributed on the lattice according to an LPP system, as shown in figure 4.2.

Definition 4.3. Lattice Population P system

A lattice population P system, LPP system for short, is a formal specification of an ensemble of cells distributed according to a specific geometric disposition given by the following tuple:

$$LPP = (Lat, (SP_1, \dots, SP_p), Pos, (\mathcal{T}_1, \dots, \mathcal{T}_p)) \quad (4.5)$$

where

- Lat is a finite point lattice in \mathbb{R}^n (typically $n = 2$) as in definition 4.2 that describes the geometry of cellular population.
- SP_1, \dots, SP_p are SP systems as in definition 4.1 specifying the different cell types in the population.
- $Pos : P(Lat) \rightarrow \{SP_1, \dots, SP_p\}$ is a function distributing different copies of the SP systems SP_1, \dots, SP_p over the points of the lattice.

- $\mathcal{T}_k = \{r_1^k, \dots, r_{n_k}^k\}$ for each $1 \leq k \leq p$ is a finite set of rewriting rules termed *translocation rules* that are added to the skin membrane of the respective SP system \mathcal{SP}_k in order to allow the interchange of objects between SP systems located in different points in the lattice. These rules are of the following form:

$$r_i^k : [obj]_l \bowtie [\]_{l'} \xrightarrow{c_i^k} [\]_l \bowtie [obj]_{l'} \quad (4.6)$$

where obj is a multiset of objects, \mathbf{v} is a vector in \mathbb{R}^n and c_i^k is the stochastic constant used in our algorithm to determine the dynamics of rule applications. The application of a rule of this form in the skin membrane with the label l of the SP system SP_k located in the point \mathbf{p} , $Pos(\mathbf{p}) = SP_k$, removes the objects obj from this membrane and places them in the skin membrane of the SP system $SP_{k'}$ located in the point $\mathbf{p} + \mathbf{v}$, $Pos(\mathbf{p} + \mathbf{v}) = SP_{k'}$. Note that vectors allow for any topology to be encoded in the lattice geometry.

Molecular reaction networks can, to a certain degree, be decomposed into modules acting as discrete entities carrying out particular tasks [8]. It has been shown that there exist specific modules termed *motifs* that appear recurrently in transcriptional networks performing specific functions like response acceleration and noise filtering [4]. Modularisation is also a central technique used in the engineering of synthetic cellular systems by combining well-characterised and standardised cellular models [228] as exemplified in the MIT BioBricks project [37]. In order to represent this conceptual and reified modularity in LPP systems we introduce the concept of a *P system module*.

Definition 4.4. P system Module

A P system module defined as a name, Mod , parameterised with three finite ordered sets of variables $O = \{O_1, \dots, O_x\}$, $C = \{C_1, \dots, C_y\}$ and $Lab = \{L_1, \dots, L_z\}$ (objects, stochastic rate constants and compartment labels respectively), and consists of a finite set of rewriting rules of the form in equation 4.2:

$$Mod(O, C, Lab) = \{r_1, \dots, r_n\} \quad (4.7)$$

The objects, stochastic constants and labels of the rules in module Mod can contain variables from O , C or Lab which are *instantiated* with specific values $o = \{o_1, \dots, o_x\}$, $c = \{c_1, \dots, c_y\}$ and $lab = \{l_1, \dots, l_z\}$ for O , C and Lab respectively as in:

$$Mod(\{o_1, \dots, o_x\}, \{c_1, \dots, c_y\}, \{l_1, \dots, l_z\}) \quad (4.8)$$

the rules are obtained by applying the corresponding substitutions $O_1 = o_1, \dots, O_x = o_x$, $C_1 = c_1, \dots, C_y = c_y$ and $L_1 = l_1, \dots, L_z = l_z$.

Our definition of *P system module* allows the hierarchical description of a complex module, $M(O, C, Lab)$, by obtaining its rules as the set union of simpler modules, $M(O, C, Lab) = M_1(O_1, C_1, Lab_1) \cup \dots \cup M_n(O_n, C_n, Lab_n)$ with $O = O_1 \cup \dots \cup O_n$, $C = C_1 \cup \dots \cup C_n$ and $Lab = Lab_1 \cup \dots \cup Lab_n$.

Finally, the set of rules, R_k , in *SP systems* can be specified in a modular way as the set union of several instantiated *P system modules*, $R_k = M_1(o_1, c_1, lab_1) \cup \dots \cup M_{n_k}(o_{n_k}, c_{n_k}, lab_{n_k})$.

The use of modularity allows us to define libraries or collections of modules:

$$Lib = \{Mod_1(O_1, C_1, Lab_1), \dots, Mod_m(O_m, C_m, Lab_m)\} \quad (4.9)$$

An SP system model may contain instantiations of modules from multiple libraries, and the same module can be instantiated multiple times with different parameters.

An example module library is presented in Table 4.1. This library contains modules modelling the basic regulatory mechanisms in gene transcription and translation. The use of these modules allow us to develop multiple models of typical regulatory mechanisms in transcriptional regulation without having to specify every single interaction in the system.

Table 4.2 presents SP system models of three recurring patterns in gene regulation networks, namely, constitutive expression, positive autoregulation and negative autoregulation of a gene, in which the rewriting rules are specified in a modular manner using modules from the library in Table 4.1 with different instantiations.

4.3 Machine-readable data formats

For LPP system models to be specified and manipulated by computers is necessary that they have a machine-readable equivalent. It is not uncommon for the machine-readable form(s) of a modelling formalism to be somewhat different from the mathematical definition, such as those given in the previous section. This section presents the encodings of LPP systems that are used by the Infobiotics Workbench.

4.3.1 MCSS-SBML

The Systems Biology Markup Language (SBML) [229] is an XML dialect used to store and exchange models of biological systems between different tools (MCSS refers to our simulator for LPP system models for which this format was initially designed). SBML files store information about model compartments, species and reactions, as well as events, units, etc. that are relevant to some models and approaches but not others. Tools for the visual specification of models in

Module	Rules	Description
$Const(\{geneX, rnaX\}, \{c_1\}, \{b\})$	$[geneX]_b \xrightarrow{c_1} [geneX + rnaX]_b$	Constitutive gene expression
$Pos(\{protY, geneX, rnaX\}, \{c_1, c_2, c_3\}, \{b\})$	$[protY + geneX]_b \xrightarrow{c_1} [protY.geneX]_b$ $[protY.geneX]_b \xrightarrow{c_2} [protY + geneX]_b$ $[protY.geneX]_b \xrightarrow{c_3} [protY.geneX + rnaX]_b$	Positive gene regulation
$Neg(\{protY, geneX, rnaX\}, \{c_1, c_2, c_3\}, \{b\})$	$[protY + geneX]_b \xrightarrow{c_1} [protY.geneX]_b$ $[protY.geneX]_b \xrightarrow{c_2} [protY + geneX]_b$ $[geneX]_b \xrightarrow{c_3} [geneX + rnaX]_b$	Negative gene regulation
$Translation(\{rnaX, protX\}, \{c_1, c_2, c_3\}, \{b\})$	$[rnaX]_b \xrightarrow{c_1} []_b$ $[rnaX]_b \xrightarrow{c_2} [rnaX + protX]_b$ $[protX]_b \xrightarrow{c_3} []_b$	Translation and degradation

Table 4.1: Library of P system modules for basic transcriptional regulation and translation.

$SP_{UnReg} = (M_{UnReg}, \mu_{UnReg}, L_{UnReg}, I_{bact}, R_{bact})$	where	$M_{UnReg} = \{geneTf, rnaTf, Tf\}$ $\mu_{UnReg} = []_l$ $L_{UnReg} = \{bact\}$ $I_{bact} = \{geneTf\}$ $R_{bact} = \begin{cases} Const(\{geneTf, rnaTf\}, \{0.025\}, \{bact\}) \cup \\ Translation(\{rnaTf, Tf\}, \{0.07, 3, 0.01\}, \{bact\}) \end{cases}$
$SP_{PAR} = (M_{PAR}, \mu_{PAR}, L_{PAR}, I_{bact}, R_{bact})$	where	$M_{PAR} = \{geneTf, Tf.geneTf, rnaTf, Tf\}$ $\mu_{PAR} = []_l$ $L_{PAR} = \{bact\}$ $I_{bact} = \{geneTf\}$ $R_{bact} = \begin{cases} Const(\{geneTf, rnaTf\}, \{0.0025\}, \{bact\}) \cup \\ Pos(\{Tf, geneTf, rnaTf\}, \{1, 0.6022, 0.025\}, \{bact\}) \cup \\ Translation(\{rnaTf, Tf\}, \{0.07, 3, 0.01\}, \{bact\}) \end{cases}$
$SP_{NAR} = (M_{NAR}, \mu_{NAR}, L_{NAR}, I_{bact}, R_{bact})$	where	$M_{NAR} = \{geneTf, Tf.geneTf, rnaTf, Tf\}$ $\mu_{NAR} = []_l$ $L_{NAR} = \{bact\}$ $I_{NAR} = \{geneTf\}$ $R_{bact} = \begin{cases} Neg(\{Tf, geneTf, rnaTf\}, \{1, 0.6022, 2\}, \{bact\}) \cup \\ Translation(\{rnaTf, Tf\}, \{0.07, 3, 0.01\}, \{bact\}) \end{cases}$

Table 4.2: Three simple models of unregulated (constitutive) expression, positive autoregulation and negative autoregulation of a gene.

SBML, in particular CellDesigner [230], enable the visual creation of models from a collection of symbols for various types of molecular and interactions called Process Diagrams¹.

Despite its ubiquity and demonstrable utility, SBML has a number of issues that have caused people to look for means of generating SBML models through less verbose textual formats such as SBML shorthand² and the modular scripting language Antimony [118], or alternative exchanges formats, of which the major alternative is CellML [233].

The biggest drawback of SBML from the perspective of LPP systems are that it has no means of encoding modular rule sets with undefined parameters, i.e. each potential reactant, including the same species in different compartments, has a unique ID, resulting in $|species| \times |compartment|$ number of IDs in the model. SBML reactions reference species by ID and therefore the same reaction in two different compartments requires two different definitions of the same reaction in the SBML file. Very complex models might have hundreds or even thousands of reactions in each compartment, or fewer reactions but many thousands of compartments. For those models, towards which LPP systems are targeted, SBML is therefore insufficient when used directly or generated as the file sizes of models would simply be too large due to the aforementioned duplication of reactions. Also, editing even medium-sized systems visually with CellDesigner would be impractical as the reaction network within a compartment must be reproduced in its entirety for every single cell. Lastly, SBML does not allow for “next-to” spatial relationships that are useful when the size of the model is subject to change, only containment as per conventional P systems.

Since we wished to incorporate CellDesigner in our modelling pipeline we were driven to find a way to be able to encode spatial-distributed and modularly-defined P system models in SBML. We achieve this by encoding the spatial location of top-level compartments and modules of rules using MCSS-specific naming schema for compartments and reactions. We use the word “template” in lieu of “module” in what follows because templates implement only some of the functionality of P system modules as defined previously.

Compartments must be named as follows: *name* : *t* : *a,b,...* : *x,y* where,

- *name* is a string, not necessarily unique, describing the compartment (and is what is presented in the Infobiotics Workbench simulation results GUI).
- *t* is empty or a non-negative integer identifying the template the compartment *defines*

¹Recently, Process Diagrams have been incorporated into the Systems Biology Graphical Notation (SBGN) [231, 232], which aims to standardise the graphical representation of biological systems and their semantic interpretation, to benefit the communication and teaching of biological models and modelling. Despite their surface similarities there is no formally defined mapping between SBML and SBGN.

²<http://www.staff.ncl.ac.uk/d.j.wilkinson/software/sbml-sh/>

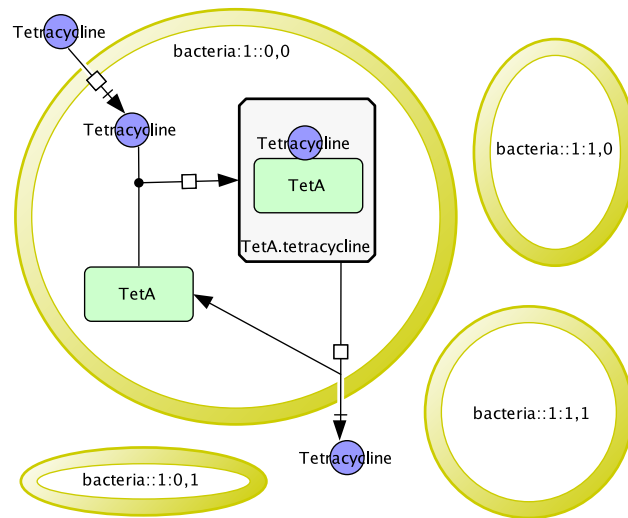


Figure 4.3: MCSS-SBML compartment naming conventions for four adjacent cells

- a, b, \dots is empty or a comma-separated list of non-negative integers giving the identifiers of the templates the compartment uses
- x, y are nothing or two non-negative integers giving the position of the compartment in 2D space

For example, figure 4.3 shows a colony of four bacteria. The bacteria named *bacteria : 1 :: 0,0* defines template 1 for the uptake of the antibiotic tetracycline and its subsequent efflux by the tetracycline resistance protein TetA. It uses no other templates and is located at position 0,0. Its neighbours do not define any other templates but each uses template 1. MCSS interprets the names according to the convention, to ensure that each compartment is endowed with *all* of the reactions it defines and uses.

With this system, replicating a compartment and its reactions becomes a matter of simply adding a compartment entry with the correctly specified name and position to the SBML file, either manually, with a script or by drawing it in CellDesigner. Compartments that define templates can be in one kept to the side of the canvas and instantiated compartments with positions arranged as those positions suggest.

Transport reactions/translocation rules follow a similar naming convention: *name* : x, y where

- *name* is any string including the empty string.
- x, y are either $*$ meaning any neighbour, or a comma-separated pair of integers specifying a vector, usually one of $(0, 1), (1, 0), (0, -1), (-1, 0)$. The x, y integers are added to the those of the compartment to determine the correct compartment to send molecules to.

4.3.2 LPP systems XML

Templates (as defined in section 4.3.1) are useful because they facilitate semi-modular model specification in SBML. However the syntax of the SBML files is essentially at odds with the discrete and stochastic nature of P systems because it was primarily designed to store mathematical models based on differential equations. While SBML is capable of storing species amounts as integer numbers of molecules, specifying stochastic rate constants (which are essentially one number because the propensity calculation to be used can be determined from the number and identities of the species) is overly complex.

It was felt that we needed to develop a custom format that would contain only features necessary for the specification of P system models and the experiments we wish to be able to perform on them, e.g. ranges and precisions for rate constants to be used by optimisation algorithms. We therefore developed an XML schema based on a collaboratively designed BNF grammar for LPP systems.

LPP system XML is a set of machine-readable data formats which closely mirrors our formal definitions. It allows us to define, in a single file or multiple files, modules of stochastic P system rules, P systems with initial multisets and instantiations of modules of rules, a geometric lattice and distribution of P systems over the lattice, which together constitute an LPP system model.

P system module XML

Modules are essentially syntactical sugar for the specification of P system models, they can be reused and composed providing the modeller with a form of hierarchical abstraction that can capture recurrent motifs in biological systems. Reuse is facilitated by the ability to externalise modules from models in *libraries*, from which they can be included into multiple models. Figure 4.4 shows a schematic of the *Const* module from table 4.1, drawn using CellDesigner, which encodes the transcription of a gene into an mRNA and the translation of that mRNA into a protein (the central dogma of molecular biology); the gene products are also degraded. Listing 4.1 shows the XML definition of the *Const* module. The object names *gene*, *mRNA* and *Protein* are substituted when the module is instantiated, as are the constants and compartment labels, to allow the module to be reused within models. Each rule is a *boundary* rule which refers the style of P systems notation in definition 4.2, not an actual boundary (there are no translocation rules in the *Const* module).

Figure 4.5 shows a schematic of a *Neg* module which augments the *Const* module with the complexation and dissociation of a gene with its product protein. This complexation prevents the transcription of the gene, by the first rule in *Const* module, and so the gene is *negatively autoregulated*. Listing 4.2 shows the definition of the *Neg* module later in the same XML file.

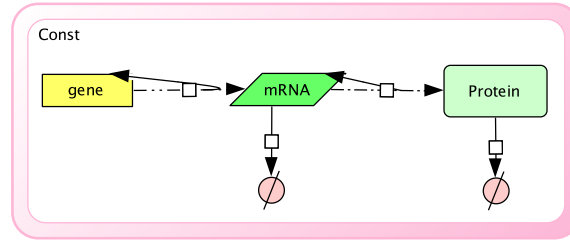


Figure 4.4: Schematic of the *Const* module with 3 species and 4 rules.

```

1  <moduleDefinition name="Const"> <!-- Constitutive expression of a gene into
    protein via mRNA assuming RNA polymerase and nucleotides, ribosomes and
    amino acids are in excess.
    r_1: [ gene ]_1 -> [ gene + mRNA ]_1 ; c_1
    r_2: [ mRNA ]_1 -> [ mRNA + Protein ]_1 ; c_2
    r_3: [ mRNA ]_1 -> [ ]_1 ; c_3
5   r_4: [ Protein ]_1 -> [ ]_1 ; c_4 -->

    <setOfObjects> <!-- substitutive objects -->
      <object name="gene" />
      <object name="mRNA" />
10   <object name="Protein" />
    </setOfObjects>

    <setOfConstants> <!-- substitutive constants -->
      <constant name="c_1" />
15   <constant name="c_2" />
      ...
    </setOfConstants>

    <setOfLabels> <!-- substitutive labels -->
20   <label name="1" />
    </setOfLabels>

    <setOfRules>
      <!-- r1: [ geneX ]_b -c_1> [ geneX + rnaX ]_b -->
25   <rule name="r_1" module_rule="1" type="boundary" constant="c_1">
      <lhs> <!-- reactants -->
        <listOfInsideObjects label="1">
          <object name="geneX"/>
        </listOfInsideObjects>
      </lhs>
      <rhs> <!-- products -->
        <listOfInsideObjects label="1">
          <object name="geneX"/>
          <object name="rnaX"/>
30   </listOfInsideObjects>
      </rhs>
    </rule>
    ... (r_1, r_2, r_3)
    </setOfRules>
40 </moduleDefinition>

```

Listing 4.1: *Const* module XML (rules *r_2*, *r_3* and *r_4* not shown).

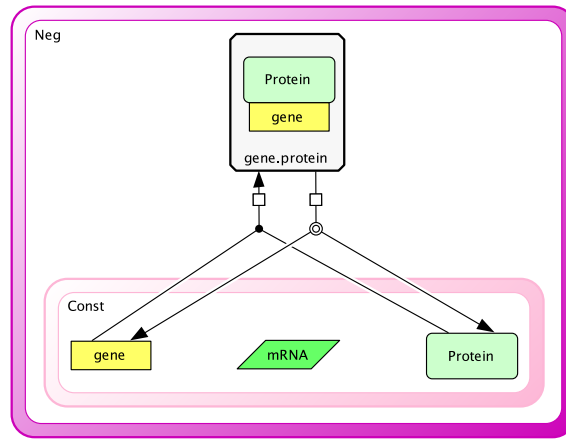


Figure 4.5: Schematic of the *Neg* module composed of a *Const* module and 2 additional rules. Only the identities of the species in the *Const* module are shown.

The `file` attribute (`file="this"` in the example) is important because it allows modules to be defined outside models and therefore be reused by multiple models. SBML does not have that facility.

Module library XML

Listing 4.3 shows a (compressed) module library containing three modules. Stochastic rate constants in library modules can be declared with additional parameters that define a range of possible values, as shown in listing 4.4. This enables the modeller to introduce a known amount of uncertainty to a rule's rate constant, which can be exploited by optimisation algorithms (see section 5.4).

Individual P system XML

Figure 4.6 shows a schematic of a compartment labelled *bacteria* that contains the *Neg* module from figure 4.5 and listing 4.2. It also contains a single rule for the translocation of the protein *Tf* outside of the compartment. Because the compartment is not contained within another compartment we called it a top-level compartment, which is analogous to a cell. Listing 4.5 shows the XML definition of the P system in figure 4.6 where the *Neg* module is imported from a module library and instantiated with the correct object names, constants and the label *bacteria*. The alphabet of objects, compartment labels, membrane structure and initial multisets are specified in the manner as in the formal definition of a stochastic P system (4.1). Note how the translocation rule r_1 has a vector value denoting the direction of translocation on a 2-dimensional lattice.

Figure 4.7 demonstrates that modules are purely intended for the specification of models and play no rule in the actual execution by showing the *bacteria* compartment as it would look from

```

1  <moduleDefinition name="Neg"> <!-- Negative autoregulation of a gene by its
    protein through complexation.
    Reuses Const module for constitutive expression. -->

    <setOfRules>
5     <module name="Const"
        objects="gene,mRNA,protein"
        constants="c_1,c_2,c_3,c_4"
        labels="1"
        file="this" />

10    <!-- r_1: [ gene + Protein ]_1 -> [ gene.Protein ]_1 ; c_5 -->
    <rule name="r_1" module_rule="1" type="boundary" constant="c_5">
        <lhs>
            <listOfInsideObjects label="1">
15                <object name="gene"/>
                <object name="Protein"/>
            </listOfInsideObjects>
        </lhs>
        <rhs>
20            <listOfInsideObjects label="1">
                <object name="gene.Protein"/>
            </listOfInsideObjects>
        </rhs>
    </rule>

25    ... (dissociation rule)

    </setOfRules>
</moduleDefinition>

```

Listing 4.2: *Neg* module XML.

```

1  <libraryOfModules name="example module library">

    <moduleDefinition name="Const">
        <!-- Const module definition from above -->
5        ...
    </moduleDefinition>
    <moduleDefinition name="Neg"> ... </moduleDefinition>
    <moduleDefinition name="Pos"> ... </moduleDefinition>
</libraryOfModules>

```

Listing 4.3: Module library XML

```

1  <constant name="c_1" lowerBound="0.001" upperBound="0.1" precision="1"
    scale="logarithmic" />

```

Listing 4.4: Rule constant with additional parameters used by POPTIMIZER.

```

1 <PSystem name="NAR">
    <alphabetOfObjects>
        <object name="geneTf" />
5        <object name="mRNATf" />
        <object name="Tf" />
        <object name="geneTf.Tf" />
        <!-- geneTf.Tf is created by string substitution in complexation
            rule of the
            Neg module, mcss will not prompt to add it the alphabet
            if not declared here -->
10    </alphabetOfObjects>

    <compartmentLabels>
        <compartmentLabel label="bacteria" />
15    </compartmentLabels>

    <membraneStructure>
        <compartment name="bacteria" label="bacteria" />
    </membraneStructure>

20    <initialMultisets>
        <initialMultiset label="bacteria">
            <object name="geneTf" multiplicity="1" />
        </initialMultiset>
25    </initialMultisets>

    <ruleSets>
        <ruleSet label="bacteria">

30        <!-- instantiating a Neg module from the library file
            module_library.xml -->
            <module name="Neg"
                objects="geneTf,rnaTf,Tf"
                constants="1,0.6022,2,3,0.07,0.01"
                labels="bacteria"
35                file="module_library.xml" />

            <!-- adding a translocation rule outside of a module -->
            <rule name="r_1" constant="c_1" type="translocation" value="1"
                vector="0,1">
                <!-- [ Tf ]_bacteria =(0,1)=[ ]_bacteria -> [ ]_bacteria =(0,1)=[
                    Tf ]_bacteria -->
40                <lhs>
                    <listOfTranslocatedObjects label="bacteria">
                        <object name="Tf"/>
                    </listOfTranslocatedObjects>
                </lhs>
                <rhs>
45                <listOfTranslocatedObjects label="bacteria">
                    <object name="Tf"/>
                </listOfTranslocatedObjects>
                </rhs>
50            </rule>

        </ruleSet>
    </ruleSets>
</PSystem>

```

Listing 4.5: Individual P system XML

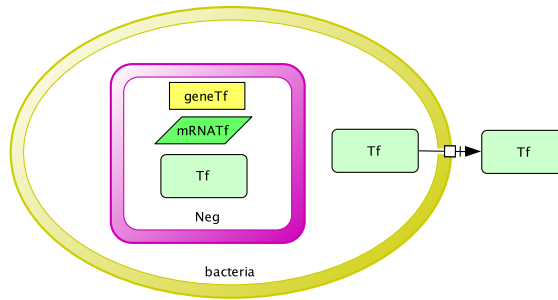


Figure 4.6: Schematic of the *bacteria* P system composed of a *Neg* module and a translocation rule.

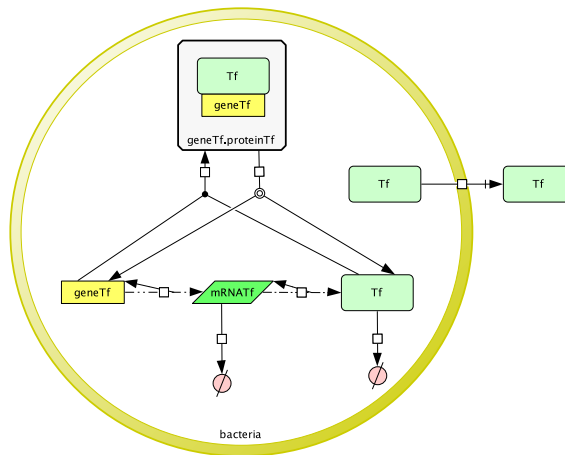


Figure 4.7: Schematic of a *bacteria* P system from the simulator's perspective with all modules flattened into one set of rules.

the point of view of the simulator, with all modules flattened to leave a simple set of rules.

Lattice Population P system XML

Our XML for LPP systems compactly describes the arrangement of individual P systems on a 2D lattice. Listing 4.6 demonstrates the distribution of 10,000 individual P systems in a 100x100 lattice. The lattice is specified using a pair of basis vectors and the substitutive parameter a , with maximum x , y sizes of 100. An individual P system definition is imported from a separate file (listing 4.5) and assigned the name *NAR*. Instantiations of that definition are distributed over the grid by creating SP systems with X and Y coordinates in the ranges 1 to 100. Ranges allow the compact instantiation of many P systems over rectangular areas on the lattice. Instances of other SP systems, modelling cells with different molecular interaction networks, can be added in the same manner.


```

1 <LPPSystemModel name="NARModel">
  <lattice name="rectangular" dimension="2" units="microns"
    scale="1" xmax="100" ymax="100">
    <parameter name="a" value="1" />
5    <basis>
      <vector x="a" y="0" />
      <vector x="0" y="a" />
    </basis>
  </lattice>
10 <listOfPSystems>
  <PSystem name="NAR" file="NAR.xml"/>
</listOfPSystems>
<SPsystems name="distribution" distribution="fixed">
  <SPsystem pSystem="NAR">
15    <Xcoordinate lowerBound="1" upperBound="100" />
    <Ycoordinate lowerBound="1" upperBound="100" />
  </SPsystem>
</SPsystems>
</LPPSystemModel>

```

Listing 4.6: Example LPP system model XML file

4.3.3 LPP systems DSLs

The LPP XML formats were well suited to software development with LPP systems, but it became increasingly clear that without a graphical tool like CellDesigner to specify reactions, writing models in XML by hand and reading them back was an arduous process with syntax obscuring information. In response Infobiotics team member Francisco Romero Campero developed a parser for an LPP system DSL (domain-specific language) that is essentially the XML formats without the angle-brackets, quotes and some closing tags. The parser is used to read DSL files directly but it also silently converts them into XML. The following listings demonstrate this format being used to specify a model of a synthetic pulse propagation circuit³. This model is built on a library of reusable promoters with DNA sequences.

³www.infobiotics.org/models/pulseGenerator/pulseGenerator.html

```

1  # Author: Francisco J. Romero-Campero #
   # Date: April 2010 #
   # Description: This library collects modules describing the regulation of
   well known promoters #

5  libraryOfModules promoterLibrary

   PLtetO1({X},{c_1,c_2,c_3,c_4,c_5,c_6},{1}) = # This module represents the
   regulation of the P(LtetO-1) promoter #
   {
   type: promoter # This is a promoter #

10  sequence: TCCCTATCAGTGATAGAGATTGACATCCCTATCAGTGATAGAGATACTGAGCAC # This
   sequence was obtained from the MIT registry of parts, part BBa_R0040
   #

   rules: # These rules describe the regulation of the promoter. #

15  # In the absence of the repressor TetR transcription initiation takes
   place at a rate of c_1 PoPs #
   r1: [ PLtetO1_X ]_1 -c_1-> [ PLtetO1_X + RNAP_X ]_1

   # The repressor TetR binds cooperatively to the promoter with an
   affinity determined by the forward constants c_2 and c_4 and the
   reverse constants c_3 and c_5 #
   r2: [ PLtetO1_X + TetR ]_1 -c_2-> [ PLtetO1_TetR_X ]_1
20  r3: [ PLtetO1_TetR_X ]_1 -c_3-> [ PLtetO1_X + TetR ]_1
   r4: [ PLtetO1_TetR_X + TetR ]_1 -c_4-> [ PLtetO1_TetR2_X ]_1
   r5: [ PLtetO1_TetR2_X ]_1 -c_5-> [ PLtetO1_TetR_X + TetR ]_1

   # Anhydrotetracycline (ATc), a tetracycline analog, binds the TetR
   repressor with an affinity determined by the constant c_6. This
   makes the repressor to drop from the promoter #
25  r6: [ PLtetO1_TetR_X + ATc ]_1 -c_6-> [ PLtetO1_X + TetR_ATc ]_1
   r7: [ PLtetO1_TetR2_X + ATc ]_1 -c_6-> [ PLtetO1_TetR_X + TetR_ATc ]_1
   }

```

Listing 4.7: Library of promoter modules used in synthetic biology models.

```

1 SPsystem signalCell

   alphabet # Molecular species involved in the regulation and expression of
           the synthetic gene circuit #

5   PtetO102_RBSI_luxR_LVA # The construct regulating the transcription,
           translation and degradation of the luxR gene and its corresponding
           protein LuxR is represented in the following string-object. It
           represents the recombinant DNA fusing the promoter PtetO102 with the
           RBSI developed by Ron Weiss, with the luxR gene from Vibrio fischeri
           and the degradation tag LVA #

   PluxR_ToppRibo_gfp_LVA_luxI_LVA # The construct regulating the
           transcription, translation and degradation of the gfp and luxI genes
           and their corresponding proteins is represented in the following
           string-object. It represents the recombinant DNA fusing the promoter
           PluxR from Vibrio fischeri with the riboswitch ToppRibo developed by
           Topp et al, with the gfp gene and the degradation tag LVA and with
           the gene luxI with the degradation tag LVA #

10  # The rest of the molecular species involved in the system #
    3OC6
    ATc
    GFP
    LuxI
15  LuxR
    LuxR2
    LuxR_3OC6

```

Listing 4.8: Pulse propagation SP system DSL alphabet.

```

1 compartments
  DH5alpha # E. coli DH5alpha is used as the chassis. This cell consists
           of a single compartment. #
endCompartments

```

Listing 4.9: Individual SP system with a single top-level compartment (single cell).

```

1 initialMultisets
  initialMultiset DH5alpha # Initial conditions stating the initial
           number of molecules in the compartment representing the bacterium #

           # The only objects initially present in the cell compartment are the
           ones representing the three different constructs regulating the
           genes luxR, cI and gfp #
5   PtetO102_RBSI_luxR_LVA 1
   PluxR_ToppRibo_gfp_LVA_luxI_LVA 1
   theop 300
   3OC6 300

10 endInitialMultiset
endInitialMultisets

```

Listing 4.10: Initial multiset of objects in the DH5alpha compartment.

```

1 # The following rules represent the dimerisation of LuxR in the
   presence of 3OC6 #

r1: [ LuxR + 3OC6 ]_DH5alpha -c1-> [ LuxR_3OC6 ]_DH5alpha c1 = 1
r2: [ LuxR_3OC6 + LuxR_3OC6 ]_DH5alpha -c2-> [ LuxR2 ]_DH5alpha c2 = 1

```

Listing 4.11: Boundary rules modelling complexation of quorum sensing effectors.

```

1      #PtetO102 RBSI luxR LVA#

      PtetO102({RBSI_luxR_LVA},{5,1,0.1,1,0.1,1},{DH5alpha}) from
          promoter_library_June.plb
      RBSI({luxR_LVA},{2.98,0.14,1},{DH5alpha}) from RBS_library.plb
5      luxR({LVA},{2.74},{DH5alpha}) from gene_library.plb
      LVA({LuxR},{0.14},{DH5alpha}) from degradation_library.plb
      LVA({LuxR2},{0.14},{DH5alpha}) from degradation_library.plb

```

Listing 4.12: Promoter module instantiation in DH5alpha cell rule set.

```

1      # The following four rules represent the diffusion of the signal 3OC6 #
      r1: [ 3OC6 ]_DH5alpha =(1, 0)=[ ] -c1-> [ ]_DH5alpha =(1, 0)=[
          3OC6 ] c1 = 0.1
      r2: [ 3OC6 ]_DH5alpha =(-1,0)=[ ] -c2-> [ ]_DH5alpha =(-1,0)=[
          3OC6 ] c2 = 0.1
      r3: [ 3OC6 ]_DH5alpha =(0, 1)=[ ] -c3-> [ ]_DH5alpha =(0, 1)=[
          3OC6 ] c3 = 0.1
5      r4: [ 3OC6 ]_DH5alpha =(0,-1)=[ ] -c4-> [ ]_DH5alpha =(0,-1)=[
          3OC6 ] c4 = 0.1

      # Alternatively, sending a molecule outside the top-compartment
      # uses the lattice neighbours to determine which directions to
      # send it #
      #r5: [ 3OC6 ]_DH5alpha -c5-> 3OC6 [ ]_DH5alpha          c5 =
          0.1#
10     endRuleSet

```

Listing 4.13: Transport rules relocating a quorum sensing signal.

4.4 Reusability of LPP systems

This LPP formalism enables three types of modelling component reuse:

- Inter-model reuse: Modules (in libraries), SP systems and lattices (encoding neighbourhood relationships between SP systems in 2D space) reside in different files which can be referred to by multiple LPP system models.
- Intra-model reuse: multiple copies of different SP system can be placed within each LPP system, facilitating the building of models of homogeneous or heterogeneous bacterial colonies or tissues.
- Intra-submodel reuse: parameterisable modules of rules can be instantiated multiple times within each compartment of an SP system, using the same or different parameters.

Modules of rules are a means of grouping sets of reactions that repeatedly occur together within a model, and by moving modules into libraries they can be shared between set of models. We use modules as a means of constraining model structure optimisation to biological plausible reaction interaction networks and maintaining a consistent level of detail across models.

Libraries of modules will typically share a level of abstraction at which cellular processes are modelled. For example gene translation can be modelled stochastically with or without the

```

1  # Author: Francisco J. Romero-Campero #
  # Date: April 2010 #
  # Description: This is the specification of a 2D finite point square lattice
    #
5  lattice square
    dimension 2 # This is a 2D lattice #
    xmin      0
    xmax      10
10   ymin      0
    ymax      30
    # The size of the lattice 11x31 = 341 SP systems #
    parameters # List of parameters used in the lattice definition #
15   parameter a value = 1 # A single parameter is used #
    endParameters
    basis # The basis vectors determine the points in the lattice #
      (a,0)
20   (0,a)
    endBasis
    vertices # The vertices vectors determine the shape of each SP-system
      skin membrane #
      (a/2,a/2)
25   (-a/2,a/2)
      (-a/2,-a/2)
      (a/2,-a/2)
    endVertices
30   neighbours # The neighbour vectors determine the neighbourhood in the
      lattice #
      (1,0)
      (-1,0)
      (0,1)
      (0,-1)
35   endNeighbours
endLattice

```

Listing 4.14: Reusable square lattice definition.

```

1  # Author: Francisco J. Romero-Campero #
  # Date: April 2010 #
  # Description: #
  # This model represents a synthetic bacterial colony constituted by two
    different bacterial strains. The colony produces a wave of GFP
    expression as a response to the production of signal 3OC6 by one of the
    bacterial strains #
5
  LPPsystem propagationModel

    SPsystems # representing the different synthetic bacterial strains in the
      colony #
      SPsystem signalCell from signalCell.sps # modelling the bacterial
        strain that produces the signal 3OC6 constitutively #
10     SPsystem relayingCell from relayingCell.sps # modelling the bacterial
        strain that produces a pulse of GFP expression as a response to the
        signal 3OC6 #
      SPsystem boundaryCell from boundaryCell.sps # An empty cell used to
        represent the boundary of the system where the signal 3OC6
        dissapears #
    endSPsystems

    lattice rectangular from square_lattice.lat # A simple rectangular
      lattice is used to represent the geometric distribution of cells in
      the colony #
15
    spatialDistribution # of clones of the different bacterial strains over
      the colony is represented by distributing copies of the corresponding
      SP-systems over the points of the above introduced lattice. #

    positions for boundaryCell # in the upper and lower boundaries of the
      system #
      parameters
        parameter i = 0:1:51
        parameter j = 0:51:51
      endParameters
      coordinates
        x = i
        y = j
25      endCoordinates
    endPositions
    positions for signalCell # in the center of the system #
      parameters
        parameter i = 25:1:26
        parameter j = 25:1:26
      endParameters
      coordinates
        x=i
        y=j
35      endCoordinates
    endPositions

    positions for relayingCell # in the rest of the lattice #
      parameters
        parameter i = 1:1:50
        parameter j = 1:1:24
      endParameters
      coordinates
        x=i
        y=j
45      endCoordinates
    endPositions
    endSpatialDistribution
50 endLPPsystem

```

Listing 4.15: Composed LPP system of colony designed for pulse propagation.

presence of ribosomes. If the number of ribosomes is important to the functioning of the system then a module encoding the production of proteins from a gene with incorporate rules such as $mRNA + ribosome \rightarrow protein + mRNA + ribosome$, and it would be expected that modules of positively or negatively regulated gene expression would also include these rules so that competition for ribosomes is relevant to translation throughout the model. This can be easily achieved in our framework by factoring out translation from transcription (as in table 4.1) and having all gene expression modules instantiate the translation model with the necessary parameters. If ribosomes were not considered a limiting factor but tRNAs were then a more detailed transcription module would be required. The principles of defining the generic case and then instantiating the specific case appropriately remain.

P systems modules can be made more or less abstract by changing the number of components exposed as parameters (species identities and stochastic rate constants). Motifs of biological networks, corresponding to the topology of the underlying reaction network modelled at a particular level of detail, can be captured by fully abstract modules where all components are parameters. In this usage the names of parameters should indicate the role that their values will play in the module, such as `gene`, or `transcription_factor`.

Well-characterised synthetic biological parts and devices can be captured by fully concrete modules (i.e. without parameters because the identity of every species and the stochastic rate constants of each reaction are validated). Our implementation also allows for modules representing bioparts to be annotated with the DNA sequences encoding the genetic components (e.g. the promoter module in listing 4.7), and for those modules to be grouped and ordered sequentially in higher-order modules representing operons. A type is associated with the sequence and the order of these types is validated by a grammar similar to that of GENOCAD [234]. Valid orderings of modules with DNA sequences are outputted in a FASTA format by the LPP DSL parser, ready for the next stage in the biomatter compilation pipeline.

Stochastic simulation of LPP models will produce huge datasets containing many simulation runs of models with many compartments and many species over many timepoints. Purpose-built tools for extracting slices of this data, performing regimented statistical analysis, plotting and visualising the result are prerequisites for their successful utilisation. To this end the next chapter introduces our software suite for *in silico* experimentation with LPP system models: the Infobiotics Workbench.

Chapter 5

The Infobiotics Workbench

Chapter abstract

In this chapter we integrate in “one-pot” the *in silico* experiments components of the CAD pipeline for systems and synthetic biology to deliver the Infobiotics Workbench. We conduct a guided tour of its experimental and analytical capabilities, illustrated by its graphical user interface: the Infobiotics Dashboard. The development of the Infobiotics Dashboard is one of the major software contributions of this thesis, the design and implementation of which are presented in chapter 8.

5.1 Getting started

The Infobiotics Workbench is an integrated software suite of *in silico* experiments for LPP models in Systems and Synthetic Biology [235]. Models are simulated either using stochastic simulation or deterministic numerical integration using MCSS, and visualised in time and space with the *Infobiotics Dashboard*. Model structure and parameters can be optimised with evolutionary algorithms using POPTIMIZER, and properties of a model’s temporospatial behaviour calculated using probabilistic or simulative model checking with PMODELCHECKER.

The Infobiotics Workbench can be started from the installed Infobiotics Workbench application shortcut or from the shell with the command aliases `infobiotics-workbench` or `ibw`. Invoking the program without arguments opens the main window (figure 5.1) which houses the experiment parameterisation and results interfaces.

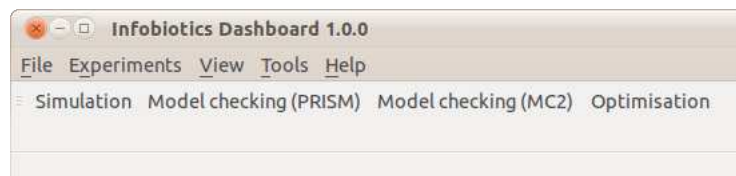


Figure 5.1: Infobiotics Dashboard main window showing the available experiments.

Editing model files

Figure 5.2 shows how the main Infobiotics Dashboard window uses an adjustable tabbed interface to display multiple views on to files, in this case text editors, side-by-side. Textual

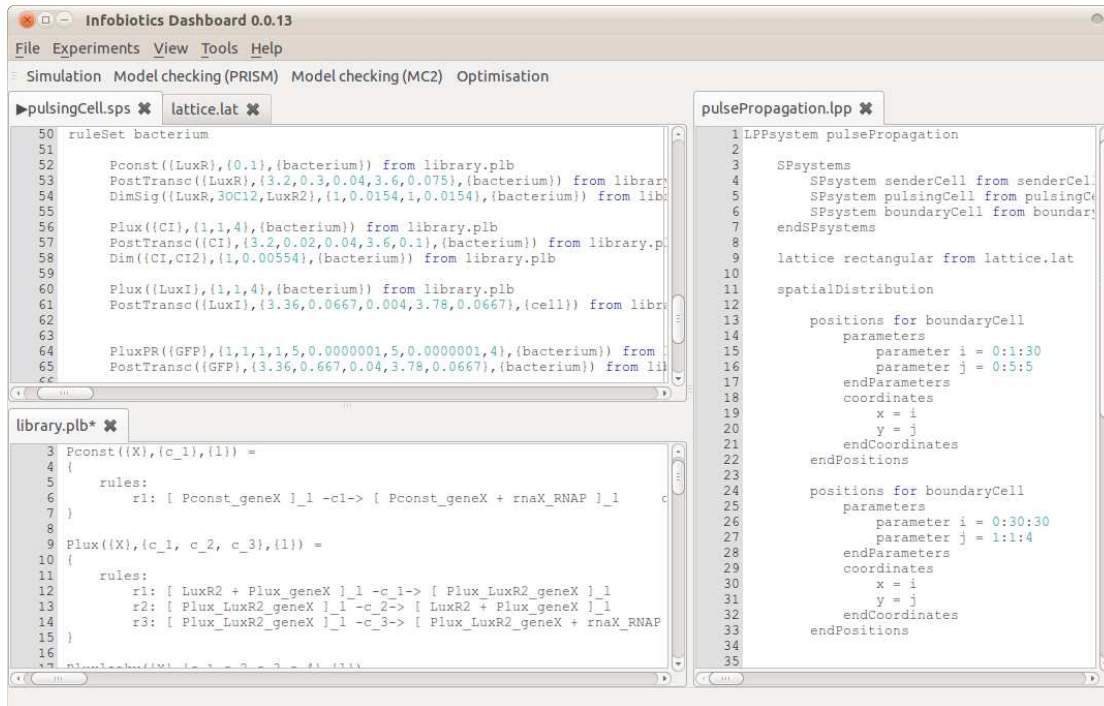


Figure 5.2: The Infobiotics Dashboard with multiple text editors displaying LPP system DSL files for a pulse generating synthetic biology model.

LPP DSL specifications of Infobiotics models can be edited with the simple editor provided the Dashboard or an external editor of the users choosing. Presently there is no widget-based (e.g. CoPaSi [236, 237]) or diagramming interface (e.g. CellDesigner [230]) with which to construct LPP DSL models. This deficiency is addressed in section 9.3.

Performing experiments

The Infobiotics Workbench implements the Polyvalent-Program Pattern [238]: providing several ways to interact with the application logic including a command line interface (CLI), a graphical user interface (GUI) and a scripting interface¹. It's multiple interfaces allow for non-interactive use via shell scripting (CLI) or interactive usage through GUIs designed to assist users in making appropriate parameter choices. In interactive mode the *experiments* can be accessed through the integrated Infobiotics Workbench interface or with individual experiment GUIs outside the Workbench. The GUI components are essentially wrappers over fluent application programming interfaces (APIs) to the various classes implementing experiment setup and results extraction, facilitating automation of experiments and analysis with Python scripts.

Infobiotics experiments are parameterised with XML parameter files (file extension: `.params`) an example of which is shown in figure 5.3. GUIs for parameterising and performing simula-

¹Chapter 11 of Applying Unix Interface-Design Patterns: http://homepage.cs.uri.edu/~thenry/resources/unix_art/ch11s07.html

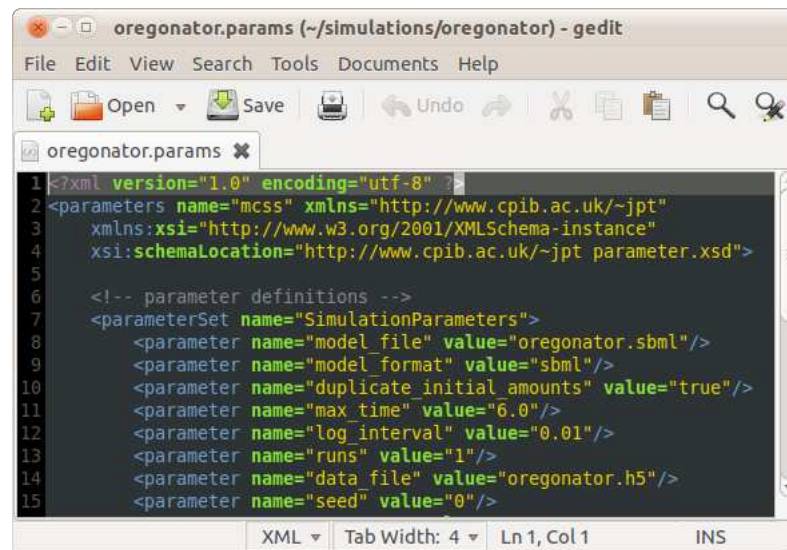


Figure 5.3: Example simulation parameters file.

tion, model checking and optimisation experiments without editing XML are shown in sections 5.2, 5.3, 5.4 respectively. To perform an experiment on the command line the name of the executable must be followed by the name of a parameters file. This may be followed by a series of `parameter_name=value` arguments overriding the values in the parameters file, for example:

```
$ mcss simulation.params max_time=60 log_interval=0.1 runs=100
```

Computationally expensive stochastic simulations can be performed on headless servers using the command line interface (CLI) prior to retrieving and analysing the results in the Dashboard. When compiled from source with the `--mpi` flag, MCSS can load balance simulation runs over HPC clusters to speed up models requiring many runs for statistical significance. Statically-compiled binaries are available for use on Linux servers that do not have the required libraries installed or where the user does not have permissions for installing libraries.

Running Infobiotics Workbench from the command line with any of the subcommands listed in figure 5.4, opens the individual GUIs for any of the experiment parameterisation or results interfaces. For instance:

```
$ ibw mcss
```

opens the dialog shown in figure 5.5 (a), while

```
$ ibw mcss-results simulation.h5
```

opens the dialog in figure 5.5 (b).

```
jvb@weasel:/tmp$ ibw --help
Infobiotics Workbench 1.0.0

ibw [command [file]]

commands:
  mcss
  mcss-results
  pmodelchecker-mc2
  pmodelchecker-prism
  pmodelchecker-results
  poptimizer
  poptimizer-results

file types:
  .params (mcss, pmodelchecker-prism, pmodelchecker-mc2, poptimizer, poptimizer-results)
  .sbml, .lpp, .xml (mcss, pmodelchecker-prism, pmodelchecker-mc2)
  .h5 (mcss-results)
  .mc2, .psm (pmodelchecker-results)
```

Figure 5.4: Infobiotics Dashboard command line interface. Access to individual Infobiotics Dashboard GUIs is achieved through arguments to the main executable.

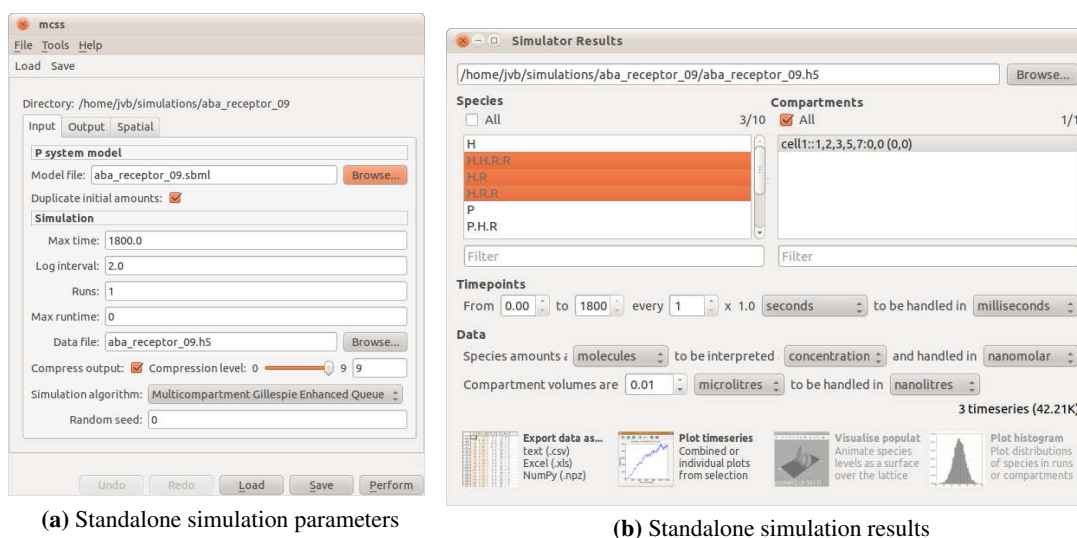


Figure 5.5: Standalone interfaces for Infobiotics Workbench components

Once the parameters of an Infobiotics experiment have been set up, clicking **Perform** will serialize those to a file and call the experimental executable with the file. Experiments output files to a working directory which can be set by saving the parameter file in another place. Experiment progress is interpreted and reported with an estimate of the elapsed time and time to completion. The general aesthetic can be seen in figure 5.6, which shows the input parameters of a simulation experiment and the progress of a running simulation, conducted within the Workbench.

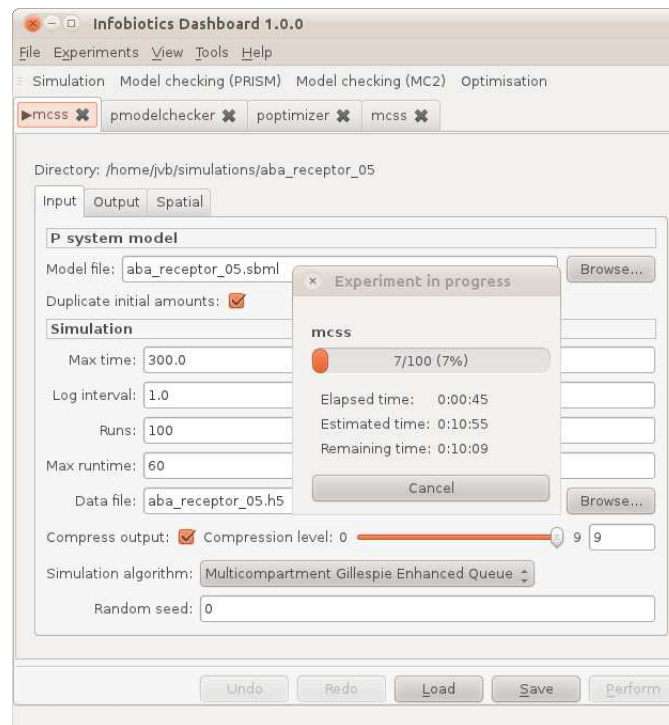


Figure 5.6: The Infobiotics Dashboard with multiple experiment parameterisation interface tabs open. The current tab shows the simulation experiment interface with the progress of a running simulation experiment displayed in the foreground dialog. Cancelling a simulation yields a truncated but usable dataset.

Persisting parameters

Parameter sets for all Infobiotics experiments can be saved to disk and loaded at a later date enabling experiments to be shared between users and reproduced. Attempting to load parameters without previously saving prompts the user to save. To ensure that experiments remain reproducible as the software evolves, older versions of the experiment executables can be specified in the preferences dialogs (**Tools > Preferences** menu) of each experiment or the main Workbench window. The GUI automatically remembers the last used value for each parameter (and selected model components in the simulation results interface) between sessions, a feature that is especially convenient when fine tuning a model file, rerunning the simulation and inspecting the changed results file.

We now proceed to demonstrate the capabilities of the each Infobiotics experiment individually, using the respective Dashboard parameterisation GUI as a guide to the essential inputs and supported options, and the respective Dashboard results interfaces to understand the nature of the experimental results. Figure 5.7 summarises the overall flow of information through the components of the Infobiotics Workbench.

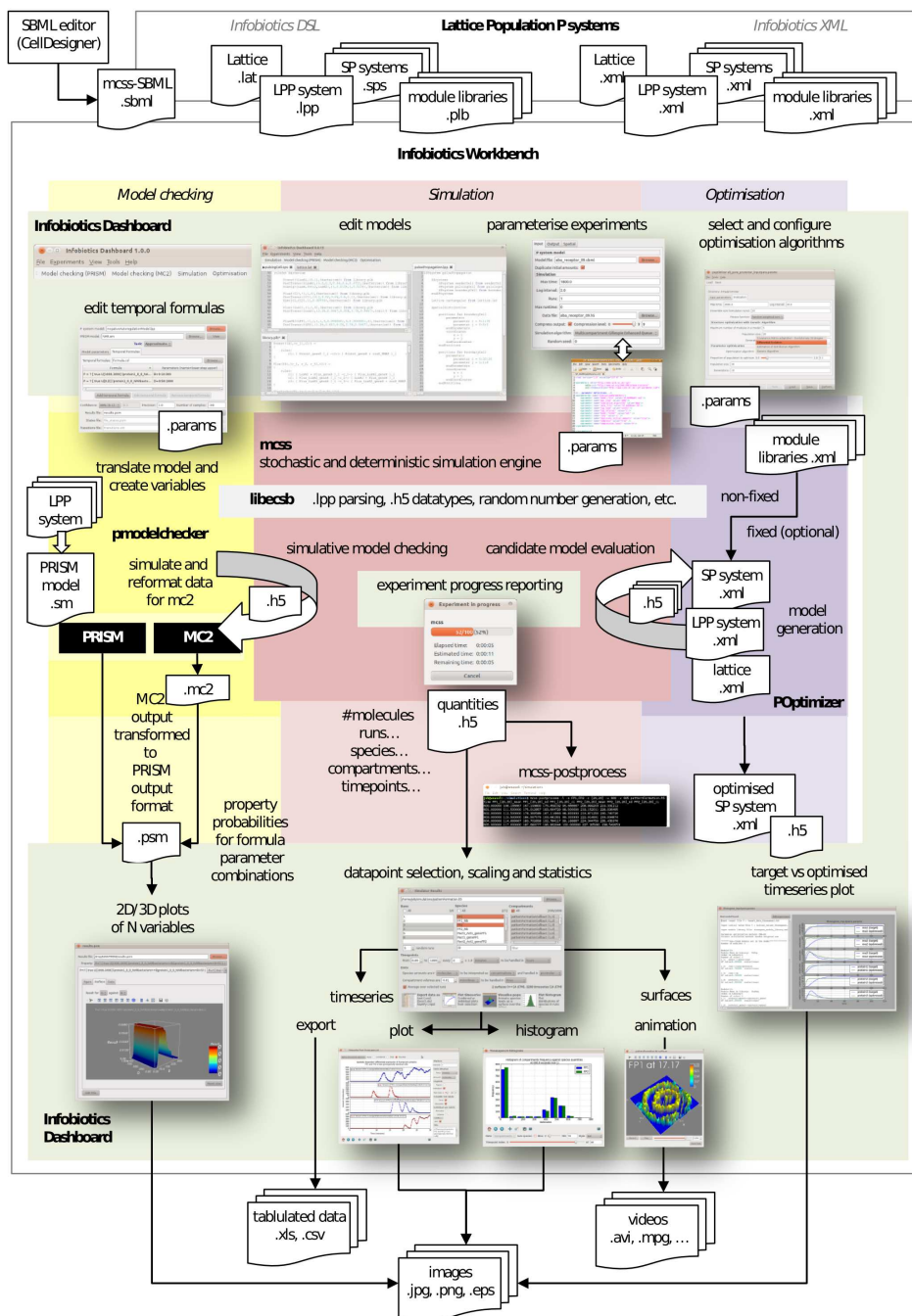


Figure 5.7: Flow of information through the components of the Infobiotics Workbench. Data is passed between components as files. Parameter files (.params), referencing model files (.sbml, .lpp or .xml), are produced by the Infobiotics Dashboard and supplied to the experiment executables for simulation (MCSS), model checking (PModelChecker) and optimisation (POptimizer). Executables communicate progress to stdout which is read and interpreted by the Dashboard to report the percentage completed and estimate time remaining. Files produced by the experiments (.h5 simulation data, .psm model checking property probabilities) are presented by the Dashboard for analysis, and can be exported as tabulated data, images and video files.

5.2 Simulation with MCSS

Simulation recreates the dynamics of a system as described by a model. Quantitative simulations enable measurement of model features changing in time which can be compared with observations of the real system for validation and predictive purposes. The Infobiotics Workbench simulator, MCSS developed by Dr. Jamie Twycross, offers a choice of two types of quantitative simulation: deterministic numerical approximation with standard solvers, and execution of the model with stochastic simulation algorithms (section 2.2). In addition to providing a baseline implementation of the canonical Gillespie Direct Method, MCSS implements an optimised multi-compartmental SSA with queue [239] that takes advantage of the compartmentalised nature of LPP system models by storing the next reaction to fire for each compartment in the heap and only recalculating the propensities of the reactions in the compartments where a reaction occurs, the both compartments involved in a species translocation. This greatly improves performance, decreasing the simulation time of models with tens of thousands of compartments and hundreds of reactions and species per compartment (please see 3.6.2 for a more detailed explanation). Deterministic simulations are performed using algorithms provided by the GNU Scientific Library (GSL) [240], including explicit 4th order Runge-Kutta and implicit ODE solvers.

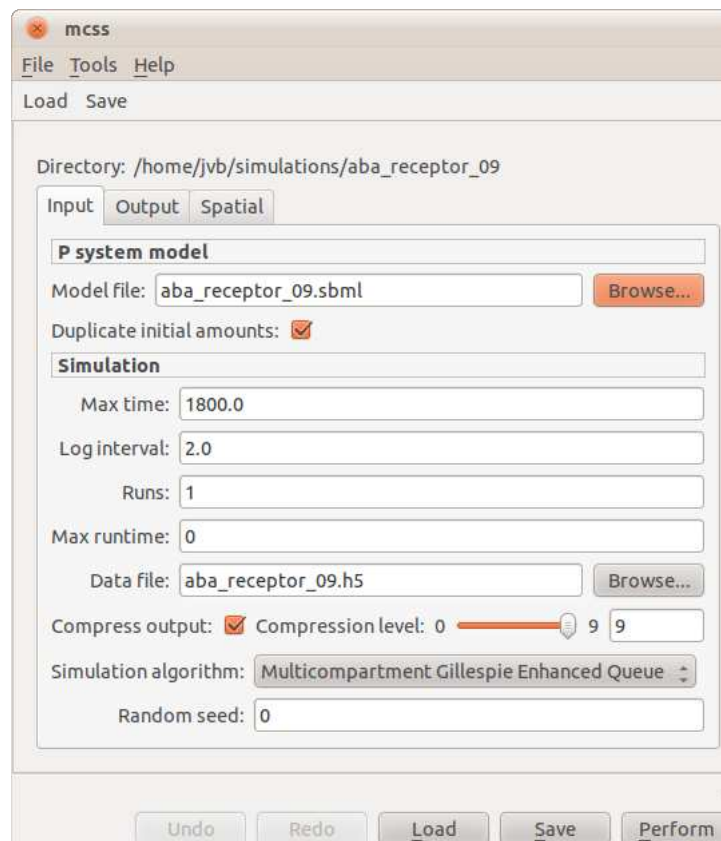
Figure 5.8 shows the possible parameters for a simulation experiment (parameter names as received by the experiments executables are lowercase and underscore separated but for aesthetic reasons the Dashboard displays them as space separated in sentence case). Of those that are not self-explanatory:

duplicate_initial_amounts duplicates the initial species amounts from the first MCSS-SBML template used by a compartment when checked (see 4.3.1; visible only when `model_file` extension is `.sbml`)

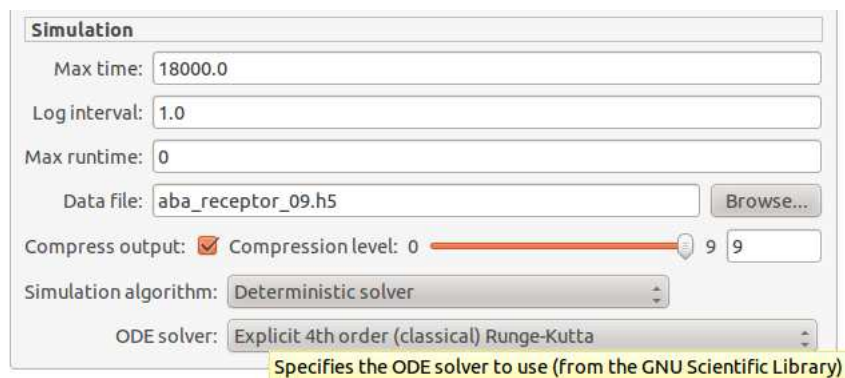
logging_type specifies whether to log species quantities (`levels`) or reactions in the order they are applied (stochastic simulation only).

log_steady_state terminates simulation early when a steady-state has been reached, filling remaining datapoints with the steady-state values.

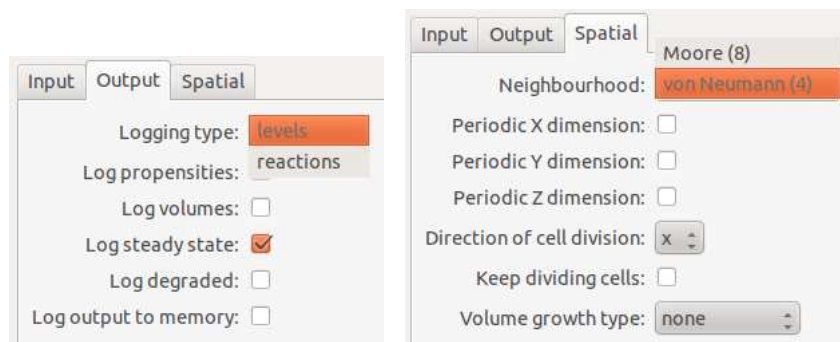
neighbourhood specifies whether to duplicate translocation rules sending molecules out of top-level compartments for just the 4 horizontal and vertical neighbours (Moore neighbourhood) or to include the 4 diagonals also (von Neumann neighbourhood).



(a)



(b)



(c)

(d)

Figure 5.8: Simulation parameters: stochastic input (a), deterministic input (b), output (c) and spatial (d).

periodic_x_dimension species whether the transport rules from on the boundaries should wrap around on the X dimension.

direction_of_cell_division MCSS has in the past supported cell division along one axis only (a restriction of the logging data structures). This feature is currently not supported by the exposed simulation algorithms and therefore is not used. Similarly `keep_dividing_cells` and `volume_growth_type` are also ignored.

During simulation MCSS performs buffered writes of species levels to the specified `data_file` in the Hierarchical Data Format (HDF5) (extension `.H5`) [241], transparently compressed (using the LZO real-time compression library [242]) to the level set in the simulation parameters (highest compression by default) for faster write times. HDF5 is a flexible file format that is well-tested in scientific environments (e.g. CERN) and purposely designed to efficiently manage very large sets of heterogeneous data, organised hierarchically and annotated with metadata. The resultant simulation data file contains a datapoint for every species in every compartment at each logging interval of every run (except those species whose names are prefixed with “_degraded”, if `log_degraded` is set to false which is the default). Often only a few species will be of interest, such as those that are observable in the real system or others that implement a biological mechanism that cannot be observed directly. Similarly, depending on the model, only some cells may be of interest (e.g. those at the interface of two populations or those far from the interface), timepoints observable in the laboratory may have a wider interval than can be logged in the simulation, or runs are to be examined independently instead of as an ensemble. In each of these cases, and due to memory constraints, it is necessary to provide a means of selecting only those datapoints that are important and to extract them from the simulated data file on demand.

Simulation results

When a model is simulated via the GUI, the output data file of a completed simulation is auto-loaded into the simulation results interface under a new tab, as shown in figure 5.9. The purpose of this interface is to enable the user to select a subset of the datapoints logged during a simulation, which can then be visualised using the provided timeseries, histogram or surface plotting functions (explained in detail below), or exported in various data formats for manipulation by third party software.

The path to the opened simulation data file is displayed in a read-only line edit widget at the top of the interface, allowing the user to copy the file path to the clipboard. An alternative simulation

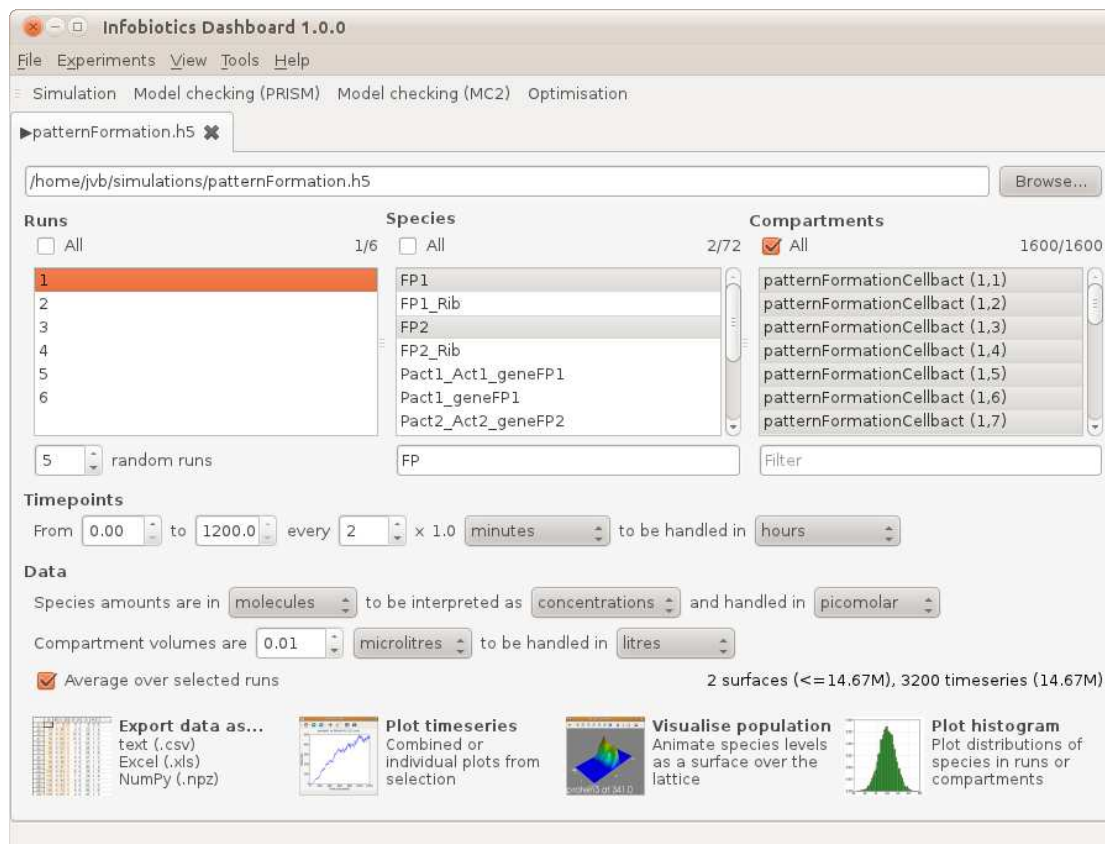
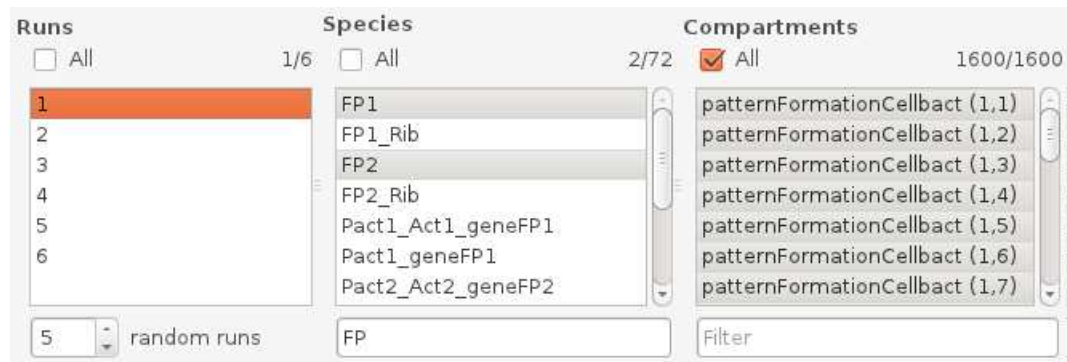


Figure 5.9: Main stochastic simulation results interface.

file can be loaded using the **Browse...** button to its right, which creates a native open file dialog for .H5 files.

Simulation datapoints selection

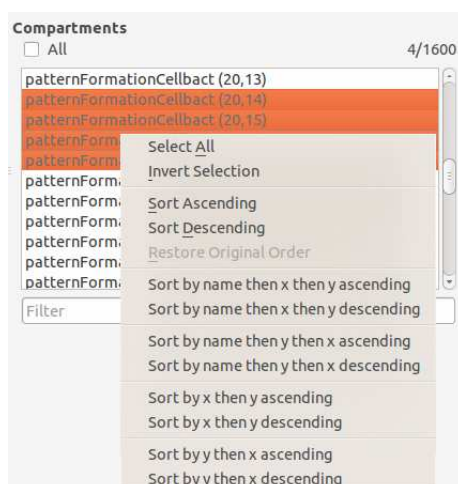
On loading a simulation the interface is populated with the available simulation data, from which the user can select some or all of the runs, species, compartments, and logged timepoints.



The lists of run, species and compartment entries are each connected to an **All** checkbox and a label showing the number of selected entries and the total number of entries. When checking **All** the previously selection is remembered and can be restored by unchecking, helpful when making complex selections.

Multiple runs, species and compartments can be selected by the usual methods of the operating system, i.e. clicking and dragging up or down the list. Under Linux, Ctrl-click selects additional individual entries and Shift-click selects the range starting from the previously selected entry to the clicked entry. Deselection works similarly. Selection can be inverted by right-clicking (Linux/Windows, Cmd-click on Mac) the list and selecting that option. An unbiased subset of runs can be made by changing the value of the **random runs** spinner.

Sorting and filtering species and compartment lists The list of species names can be sorted in either ascending or descending alphabetical order, and filtered by typing a partial name into the line edit below the list. The list of compartments can similarly be sorted or filtered by name, and compartments can additionally be sorted by their X and Y positions on the lattice. Filtering and the All checkbox work in conjunction so that selecting all entries in the list while it is filtered will only select the visible entries.



Sampling timepoints selectively The number of timepoints to use can be reduced by changing the values of the *from*, *to* and *every* spinboxes. The values of *from* and *to* are constrained by their natural relationship so that *from* may not be larger than *to* or *to* smaller than *from*. *Every* ranges from 1 to the total number of timepoints between *from* to *to*. The *logging interval* (*log_interval* parameter) of the simulation is shown as a multiple of *every*.



Setting and changing units

The simulation data file does not store the units of the model components at present; units are absent from the LPP DSLs and optional in SBML. Comboboxes for selecting the *data units* of model components as they were simulated, and the *display units* in which simulation results are to be handled and presented in plots, are provided for timepoints, species quantities and compartment volumes.



All stochastic rate constants in the model should have the same units, e.g. $\text{molecules} \cdot \text{seconds}^{-1}$ (it is the modellers responsibility to ensure that this is the case). The data units of the timepoints should be in the same time units as the rate constants; both are set by changing the value in the combo box immediately following the logging interval value and the default is **seconds**. The display units of time can be set to pico-, nano-, micro-, milliseconds, seconds, minutes, hours, and days.

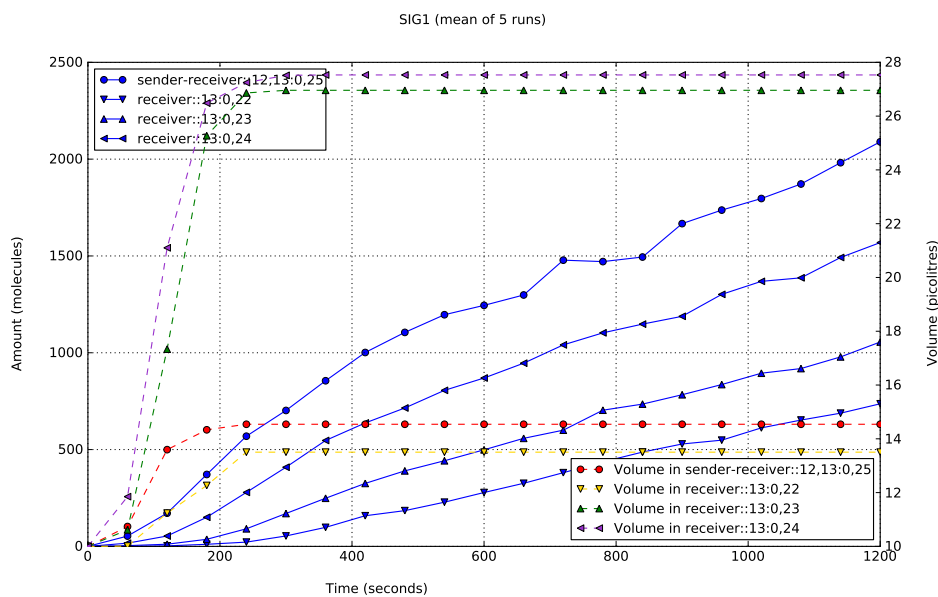
The assumed data units of species amounts are *molecules*, but this may be changed to any SI prefix of moles such as nanomoles. Species amounts may be interpreted as either molecules,

moles or concentrations, the choice determines which display units are available. If molecules of species are to be interpreted as concentrations and displayed in nanomolar (nM) for instance, then the volume of each compartment must be known in order to perform the conversion. Volumes are also missing from the LPP DSLs, so to handle this case a universal compartment volume may be set in units of the users choosing. The volume spinner is hidden for data sets with volumes.

From version 1.0.2 compartment volume defaults to 0.01 picolitres, an estimate of the volume of a bacterial cell volume [243]. This is explained in the widget's tooltip, which also suggests 1 picolitre is a reasonable estimation of plant cell volume [244].

Handling growth and division

MCSS was capable of simulating cell growth and division along one axis, where the compartments new volume is calculated as a function of time given in the name attribute of its element in MCSS-SBML, but this capability has been withdrawn. The simulation results part of the Infobiotics Dashboard retains the capacity to work with dividing compartments and changing compartment volumes. If a volumes dataset is found to be present in the simulation data file then species concentrations are calculated using the volume of the compartment at each time-point and the number of the molecules. Timeseries of changing compartment volumes can be plotted by selecting an additional *Volumes* item which appears in the *species* list. If species are also selected then volumes are plotted on a secondary axis of the timeseries plot, for example:



At some point in the future when simulation of cell growth and division is reinstated in MCSS the Dashboard is ready to utilise this additional information, provided the data structures do not change.

Additional options



Lastly, the user can choose whether or not to average the amounts of each species in each compartment over the set of selected runs (default for stochastic simulations, hidden along with the list of runs for deterministic simulations). Averaging over many runs can approximate the average behaviour of a stochastic system. The greater the number of runs used to calculate the average the more likely it is the converge on the solution of the equivalent deterministic model, providing stochasticity does not lead to a wide variety of different outcomes. The variability of species amounts across runs is summarised by the standard deviation and a derived estimated confidence interval statistics which in timeseries plots are overlaid on timeseries of the mean as shown in figure 5.13.

Estimated memory usage

2 surfaces (<=14.67M), 3200 timeseries (14.67M)

Before proceeding to export or plot the selected data it is useful to have an estimate of the total amount of data that has been selected, as this will determine how quickly the action can be performed and whether the results will be comprehensible. Extracting more data than the available memory of the computer will cause virtual memory requiring disk access to be invoked and leave no memory free for other programs, slowing the computer down considerably. Attempting to plot hundreds of timeseries will take some time to render and the resulting jumble of graphs will likely be unusable.

Rather than place an arbitrary limit on how many datapoints, timeseries or how many megabytes the user can extract and plot, the Dashboard estimates and displays the number of timeseries and surfaces, and the approximate memory requirements of each action, allowing the user to decide what is too much. Averaging over runs reduces the number of timeseries to handle from $runs \times species \times compartments \times timepoints$ to $3 \times species \times compartments \times timepoints$ as the mean, standard deviation and estimate confidence interval for each timepoint is calculated over the runs.

Exporting data



The selected and rescaled datapoints can be exported from the Infobiotics Dashboard by clicking the **Export data as...** button to open a save file dialog limited to files with the extensions `.csv`



Figure 5.10: CSV options

(comma-separated value), `.xls` (Microsoft Excel) and `.npz` (NumPy). All formats contain the same information: timeseries of the species amounts (and compartment volumes if available) at the selected datapoints, converted into the chosen units.

Columns of the tabular formats (CSV and XLS) are ordered lexicographically, first by run (if not averaging), then compartment and then species, with time always as the first column. This permits calculated traversal of columns by software reading the files. If a volumes dataset is available and/or species amounts are being displayed as concentrations either a `<filename>_volumes.csv` file is also saved or a volumes sheet added to the Excel file. Columns of the volumes data are also ordered lexicographically, first by run (if not averaging) and then compartment, with time as the first column. The headers of the columns are identical for both formats, except CSV column headers are escaped with double quotes in line with the established conventions².

Saving to CSV format triggers the modal dialog shown in figure 5.10 to pop up, allowing the user to set the delimiter and number of decimal places for floating point numbers.

Saving to NPZ format enables users to work with extracted timeseries in Python. `.npz` files are *uncompressed* zip files containing the same data CSV and XLS exports but stored as NumPy arrays. These arrays retain their multi-dimensionality, so in order to enable users to access them correctly an array containing the names of the axes is also written to the file, as are arrays of the data and model file names, timepoints, runs/species/compartment names and their indices along the respective axes. The arrays are named accordingly and can be extracted from the loaded file like a regular Python dictionary. The array names can be listed using the `keys()` method of the NumPy file object.

Listing 5.1 shows an interactive Python shell session where a `.npz` file is loaded and the various arrays and metadata contained within are inspected. Finally, the quantities of the species FP2 in every 16th compartment (of 1600) at all timepoints of run 1 are extracted and the shape of the resulting array shown.

²Y. Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. Network Working Group <http://tools.ietf.org/html/rfc4180standard>

```

1 >>> import numpy as np
>>> npz_file = np.load('extracted_timeseries.npz')
>>> sorted(npz_file.keys())
['amounts', 'amounts_axes', 'compartment_indices',
 'compartment_labels_and_positions', 'data_filename', 'model_filename',
 'run_indices', 'run_numbers', 'species_indices', 'species_names',
 'timepoints']
5 >>> str(npz_file['data_filename'])
'patternFormation.h5'
>>> str(npz_file['model_filename'])
'patternFormation.lpp'
>>> npz_file['amounts'].shape
10 (5, 2, 1600, 121)
>>> tuple(npz_file['amounts_axes'])
('runs', 'species', 'compartment', 'timepoint')
>>> len(npz_file['timepoints'])
121
15 >>> list(npz_file['species_names'])
['FP1', 'FP2']
>>> list(npz_file['species_indices'])
[2, 4]
>>> npz_file['compartment_labels_and_positions'].shape
20 (1600,)
>>> npz_file['compartment_labels_and_positions']
array(['patternFormationCellbact at (1,1)',
 'patternFormationCellbact at (1,2)',
 'patternFormationCellbact at (1,3)', ...,
25 'patternFormationCellbact at (40,38)',
 'patternFormationCellbact at (40,39)',
 'patternFormationCellbact at (40,40)'],
      dtype='<S35')
>>> npz_file['compartment_indices']
30 array([ 0,  1,  2, ..., 1597, 1598, 1599], dtype=uint64)
>>> npz_file['run_numbers']
array([1, 2, 3, 4, 5])
>>> npz_file['run_indices']
array([0, 1, 2, 3, 4])
35 >>>
>>> npz_file['amounts'][0][1][::16].shape
(100, 121)

```

Listing 5.1: Interactive Python session working with data exported in NPZ format.

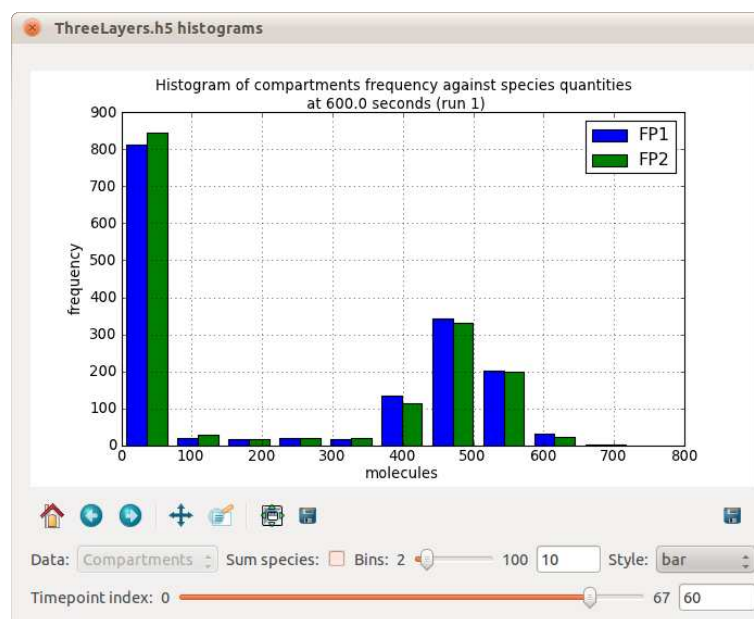


Figure 5.11: Histogram plotting interface.

Plotting histograms



Distributions of the average quantity of each selected species at a single timepoint can be plotted as histograms for either:

1. each selected compartment over all selected runs, or
2. each selected run over all selected compartments.

Whether the distributions over runs or compartments is plotted or not is determined by the value of the **data** combo box below the plot, shown in figure 5.11. Species can also be summed to, for instance, group all complexes containing a particular protein if only those complex species are selected. The number of bins from 2 to 100 (default 10) can be set using the **bins** slider. The appearance of the histogram can be changed to any of the styles available in Matplotlib [245] available styles: **bar**, **barstacked**, **step** or **stepfilled**, where **bar** means groups of bars per bin, one for each species; **barstacked** is a single bar per bin, stacked on top of each other; **stepfilled** is the standard adjacent rectangles of a histogram and **step** is the line joining the top edge of each rectangle. Finally, the timepoint currently being plotted can be changed using the **timepoint index** slider. As with timeseries plots, the histogram plot is “live”, so that as changes to the options are effected immediately. Similarly images of the histograms can be saved with the enhanced sizing functionality described in section 5.2.

Plotting timeseries

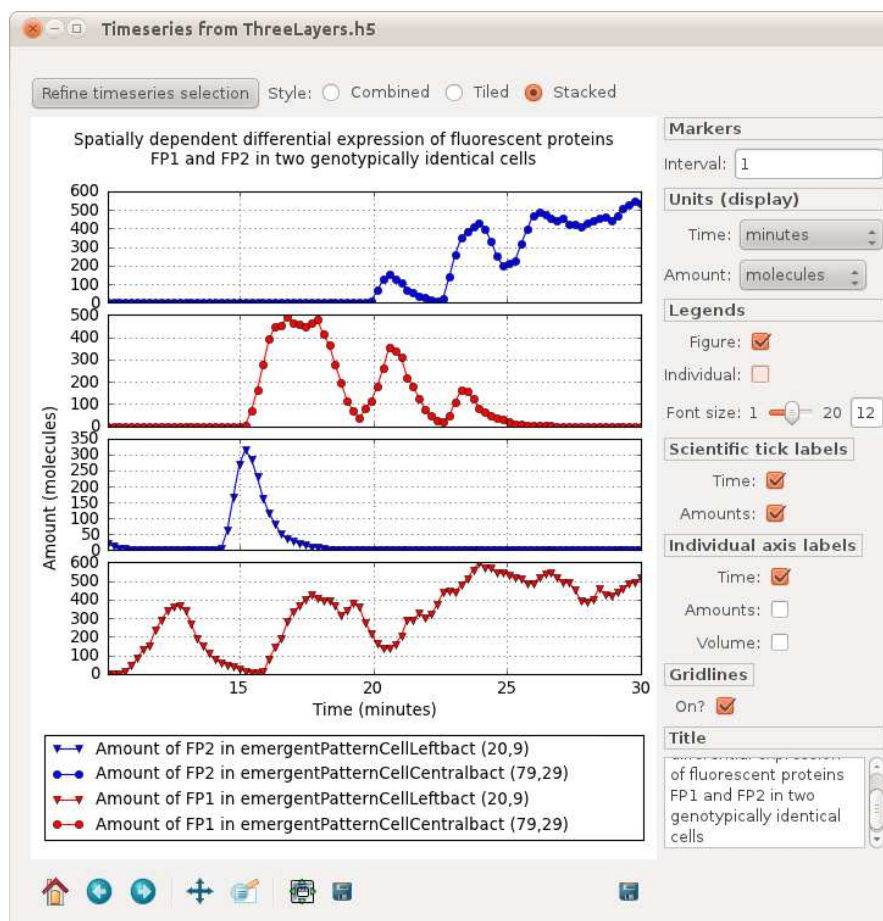


Timeseries are plotted from the selected data, either **combined** in one plot or **stacked/tiled** with individual amounts axes for better comparison of species whose timeseries are orders of magnitude different in scale. Figure 5.12 shows the timeseries plotting interface for each style. When working on a stacked or tiled plot, **Refine timeseries selection** will open a dialog in which the order and visibility of subplots can be adjusted (figure 5.12c).

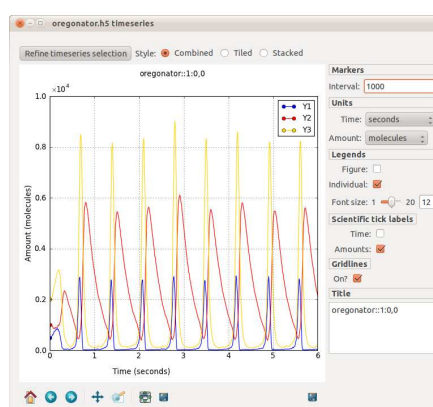
There are some restrictions on the plot elements items which can be customised using the interface. For instance, line and marker colours are determined automatically on per species basis, modulo a small number to discourage overcrowding (colours are eventually reused for other species). Multiple timeseries of the same species in different compartments can be distinguished by the shape of the markers, also chosen on the users behalf. Taken together these restrictions place a natural limit on the number of timeseries that can reasonably be plotted before meaning is lost. This is intentional as we believe it maintains clarity and consistency between plots.

The right-hand side of the interface groups the settable options. Axes values are initially rendered in the display units of the datapoint selection interface, but these can be readjusted and the data will scale accordingly. Timeseries of species quantities averaged over multiple runs are augmented with markers at a default interval of 10 timepoints. The interval between markers can be adjusted without affecting the resolution of the line, useful to declutter plots of many timepoints. Markers can be hidden by setting an interval larger than the number of selected timepoints. A (draggable) figure legend can be overlaid, instead of and in addition to the individual plot legends. The legend's font size can be also adjusted, as legends with too many entries can become unwieldy. The text of each legend entry is *computed* from the set of species/compartment names and run numbers of the timeseries so that repetitive information is factored out of legend text and into the default title text as demonstrated in figure 5.12 (a).

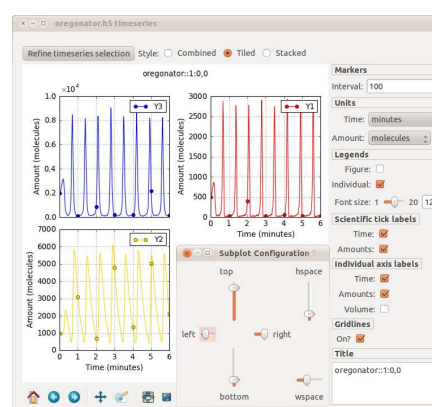
When averaging over multiple runs, each line is the *sample mean* and each marker is overlaid with error bars of either the **standard deviation of the sample** (SD) or the **confidence interval** (CI) describing the accuracy of the standard deviation. The mean timeseries from 100 runs of a negative autoregulation motif model shown in figure 5.13 demonstrates that even when the SD error bars are large (a), if the CI error bars are small (b) then we can be confident that the SD error bars (for a particular marker) are close to the actual standard deviation, because the sample size is sufficiently large. Figure 5.13 (c) shows 3 individual runs of the same model with a high degree of variation after the peak. The **degree of confidence** with which the confidence interval is calculated can be changed from the default of 0.95 to any real value in the range



(a) Stacked.



(b) Combined.



(c) Tiled (shown with Subplot dialog).

Figure 5.12: Timeseries plot styles.

0.5 to 0.999, and the errors will be updated accordingly. The function to calculate CI is: $CI = \frac{SD \times \text{InverseStudentT}(n-1, 1-(1-d)/2)}{\sqrt{n-1}}$ where n is the sample size and d is the degree of confidence.

The figure toolbar provided by Matplotlib enables zooming, panning, *Subplot configuration*: adjustment of the spacing between multiple plots and the figure boundary (dialog shown in figure 5.12c) and exporting plot image, as it appears for publication in bitmap and vector formats. Because it may be necessary to export larger or smaller images than displayed, we augmented this functionality by adding a secondary save button (far right) which in addition to setting height and width in inches with a certain DPI, allows pixel width and height to be configured reflexively with inches measurements via the DPI.

With the timeseries plotting functionality, users can make exact (combined) or relative (stacked/tiled) quantitative comparisons of the temporal behaviour of multiple molecular species in multiple compartments, between several, or averaged over many, simulation runs. These plots can be exported as images for further comparison with experimental observations.

Visualization of species quantities over the model lattice



One observable that timeseries plots cannot effectively visualize is how overall species quantities change over time *in space*, for instance when the population model is 100×100 grid of cells models. The Infobiotics Dashboard enables users to visualise how species quantities change in time and 2D space by using 3D heat-mapped meshes or *surface* (where the vertices of the mesh correspond to model lattice points and the height of the peaks to the species quantities), to capture the distribution of each selected species over the model at a single timepoint. Multiple surfaces, one per species, each corresponding to particular species, can be visualized simultaneously side-by-side for qualitative comparison. The overlaid scalar bars map heat as colour to quantity.

Two points should be noted about the data being visualised as a surface. Firstly, if multiple simulation runs are selected then the quantities visualised will be the mean over those runs. Secondly, when some of the selected compartments share a lattice point (i.e. some are enclosed within another or are subcompartments of the same deselected compartment) then the species quantities in those compartment are summed.³

Figure 5.14 shows an example in which two surface plots of 1600 compartments (40×40) are rendered. Time is progressed manually by dragging the timepoint index slider or automatically using the **Play/Pause** button. Multiple surfaces are synchronised to the same timepoint. The

³Averages could also be appropriate but are not implemented in version 1.0.

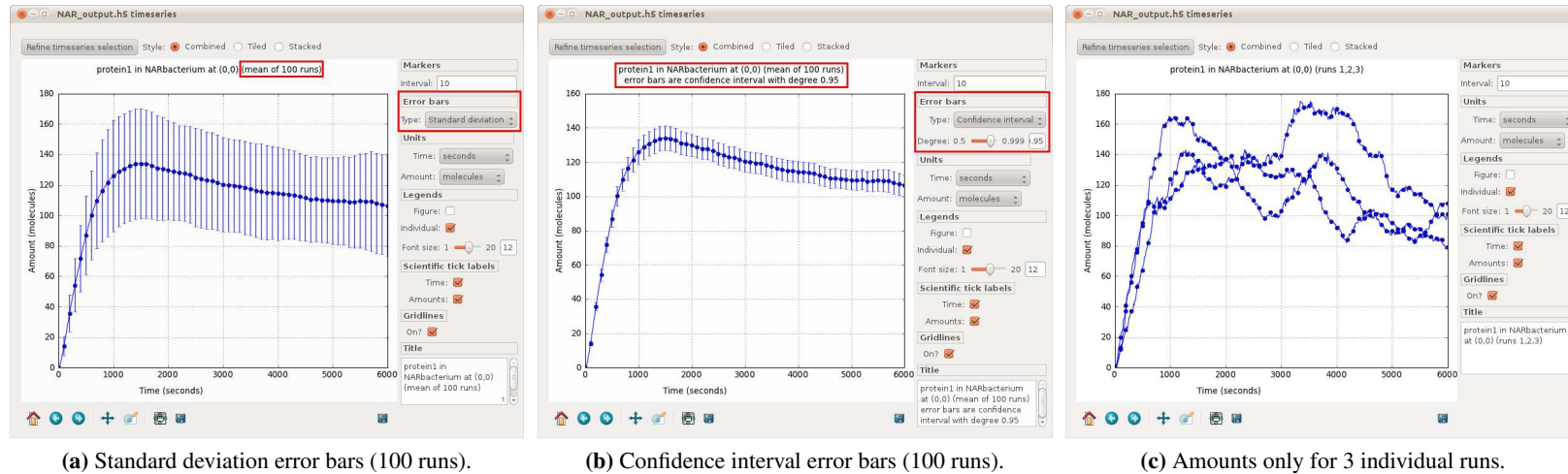


Figure 5.13: Comparison of error bar alternatives: standard deviation (a) and confidence interval (b) for 100 runs of a negative autoregulation motif model. (c) shows 3/100 runs individually.

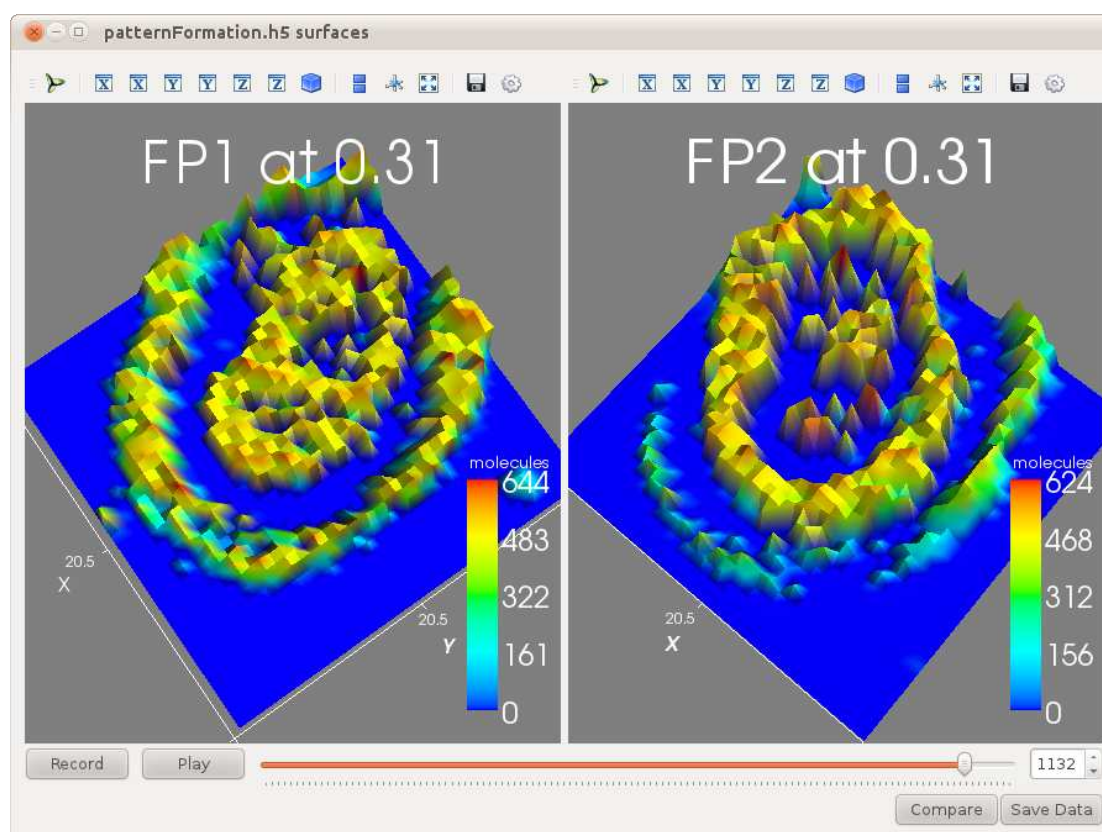


Figure 5.14: Paired surface plots showing expression patterns of fluorescent proteins.

perspective of an individual surface can be spun, tilted, zoomed and panned using the mouse and keyboard; the Mayavi [246] toolbar allows users to reset the perspective to isometric (the default), side-on or top-down, and their reverse. The other buttons enable full screen viewing, saving of single images, configuration of titles, colour maps, etc.

If the Dashboard detects that *ffmpeg* is installed it enables the *Record* button which captures a frame for each time change (forward or backward) while recording and processes these into a video file when recording is stopped. Frames from two or more surfaces are stitched together automatically to produce a single movie of both, the results of which are demonstrated in figure 5.15.

Similarly to the *Export data* function (section 5.2), which saves the underlying multi-dimensional arrays used to plot timeseries in NumPy NPZ format, the data used to plot each surface at all timepoints can be saved in NPZ format for further analysis using the *Save Data* button. The underlying functions called by the *Record* and *Save Data* methods can also be used in user scripts to obtain the data and create movies without user intervention. We used these methods together to produce more sophisticated movies. By interpolating values between the saved surface datapoints and then capturing frames of the much larger animations rendered in an off screen buffer we created a smoother, overlaid and differentially coloured version of a pattern

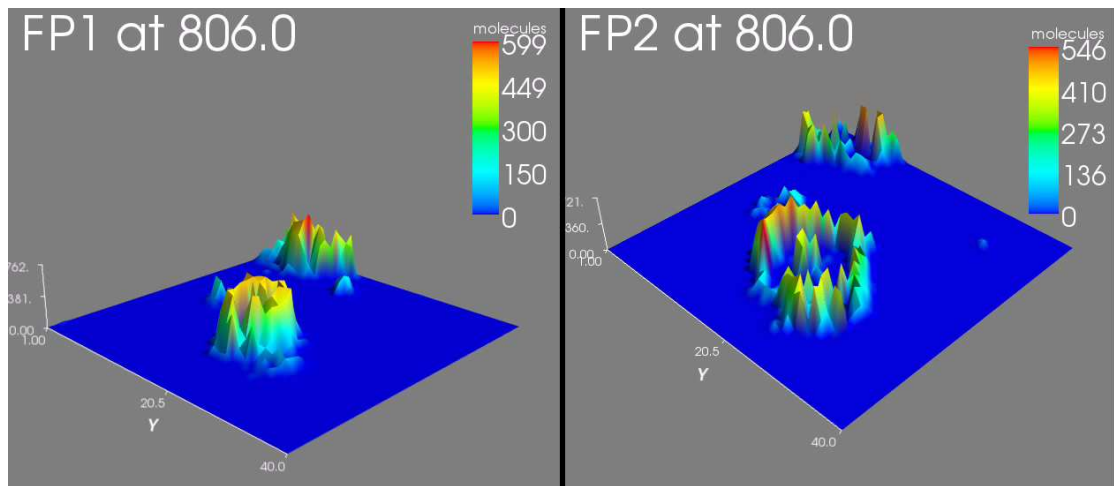


Figure 5.15: A stitched frame from an exported video of a surface plot.

formation model, as shown in figure 5.16 (right) alongside earlier synthetic biology models of pulse propagation (left) and inversion (centre).

A further semi-quantitative comparison can be made between multiple surfaces using the *Compare* feature in which some of the currently visualised surfaces may be overlapped (subtracted) in pairwise fashion (figure 5.17). Subtraction of the top-down heat maps can help to identify patterns where the amount of one species is negatively correlated with another, at the population level.

The majority of figures used in this section to illustrate the surface plotting functionality (5.14 onwards), were obtained from various iterations of the pattern formation circuit design in figure 1.4a. This design was implemented *in vivo* by colleagues in *E. coli* DH5 α cells, producing the phenotypes shown in figure 5.18 and compared in figure 5.19. These images provide an indication that the design functions as expected. A publication is in preparation, pending further controls. In conclusion, surfaces plots provide an intuitive and attractive means of qualitatively gauging the behaviour of population level models, that may (cautiously) be compared to microscopy data.

We now look at how properties of LPP models can be verified, using model checking to methodically sample the initial states and examine the subsequent dynamics.

5.3 Model checking with PMODELCHECKER

Properties of stochastic P system models can be expressed as temporal logic formulae and automatically verified using third party model checking software such as PRISM [34]. PMODELCHECKER, developed by Dr. Francisco Romero Campero, extends this capability to LPP

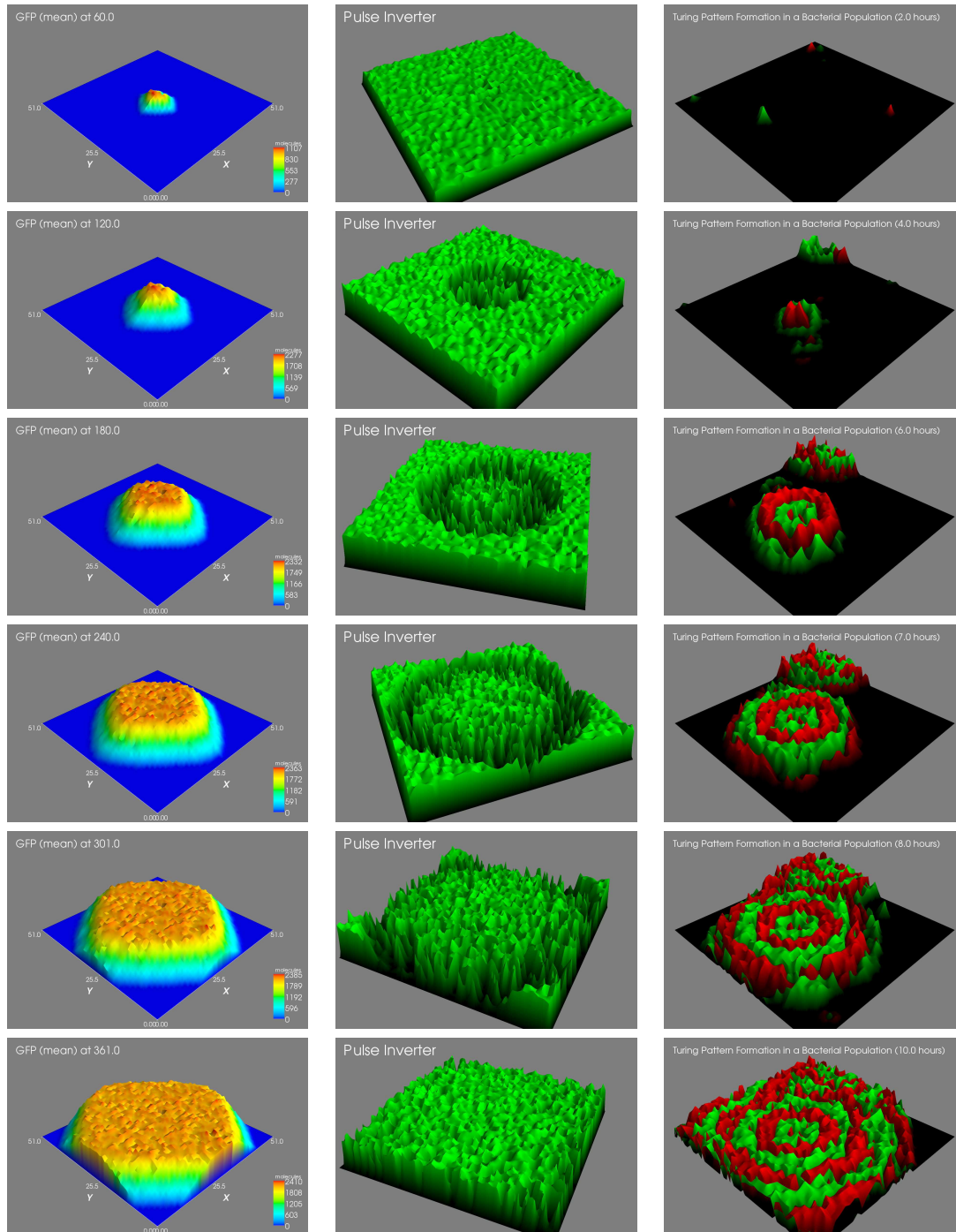
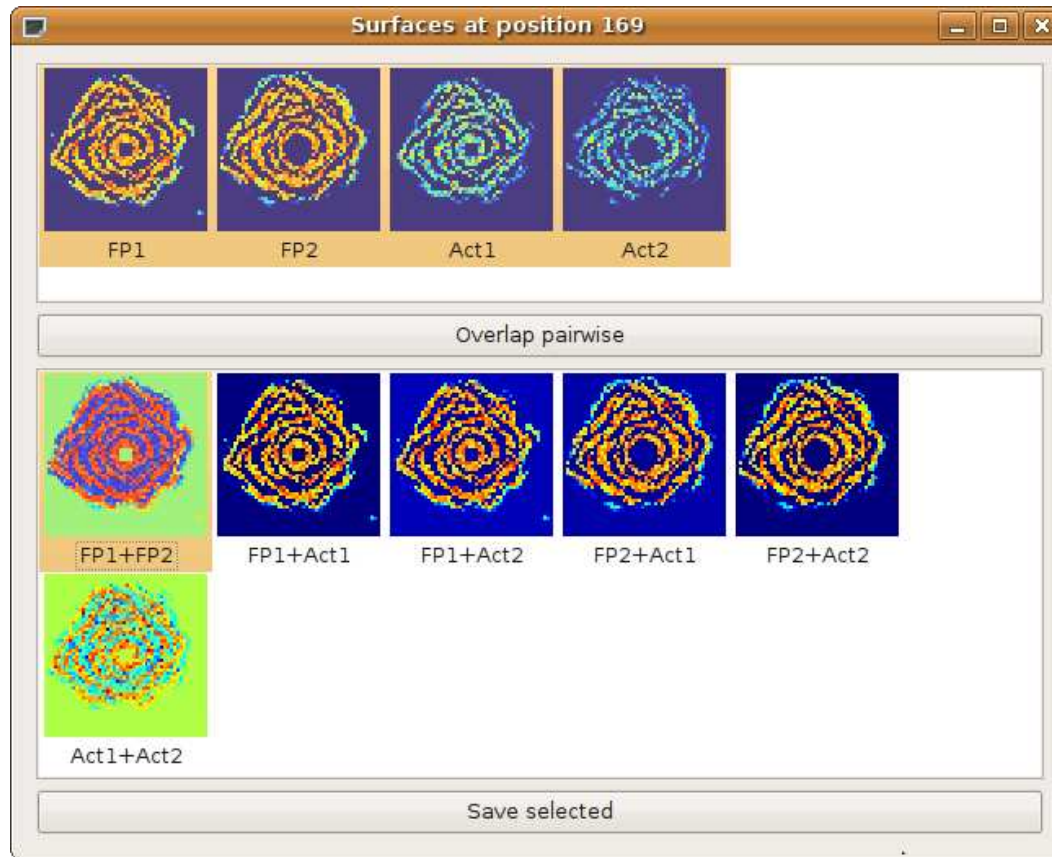
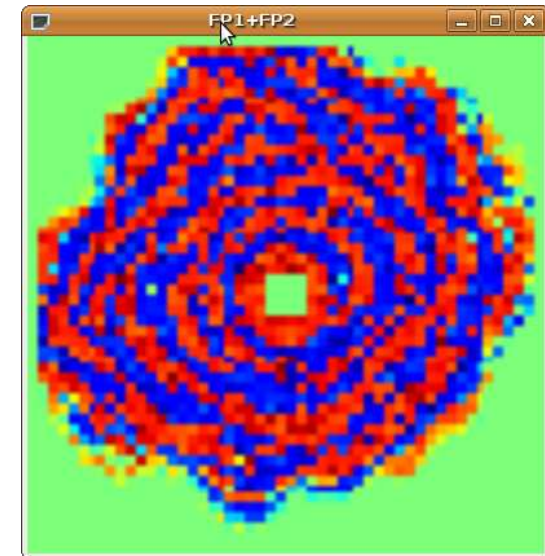


Figure 5.16: Frames from animations of three synthetic biology models. Time is progressing as the frames descend.



(a) Top-down surface images overlapped and subtracted in pairwise fashion to identify negative correlations.



(b) FP1+FP2 subtracted top-down surface image close up.

Figure 5.17: Compare editor-produced images of a pattern formation model.

In (b) the surface plot of one model fluorescent protein, FP2, has been subtracted from that of another, FP1. Green indicates equal quantities in both plots which cancel out to zero (and may both be zero). Blue indicates more FP2 than FP1 and red the opposite. We observe that at each position one or the other either dominates or is expressed exclusively (the outer cells have not begun to express either FP1 or FP2 yet).

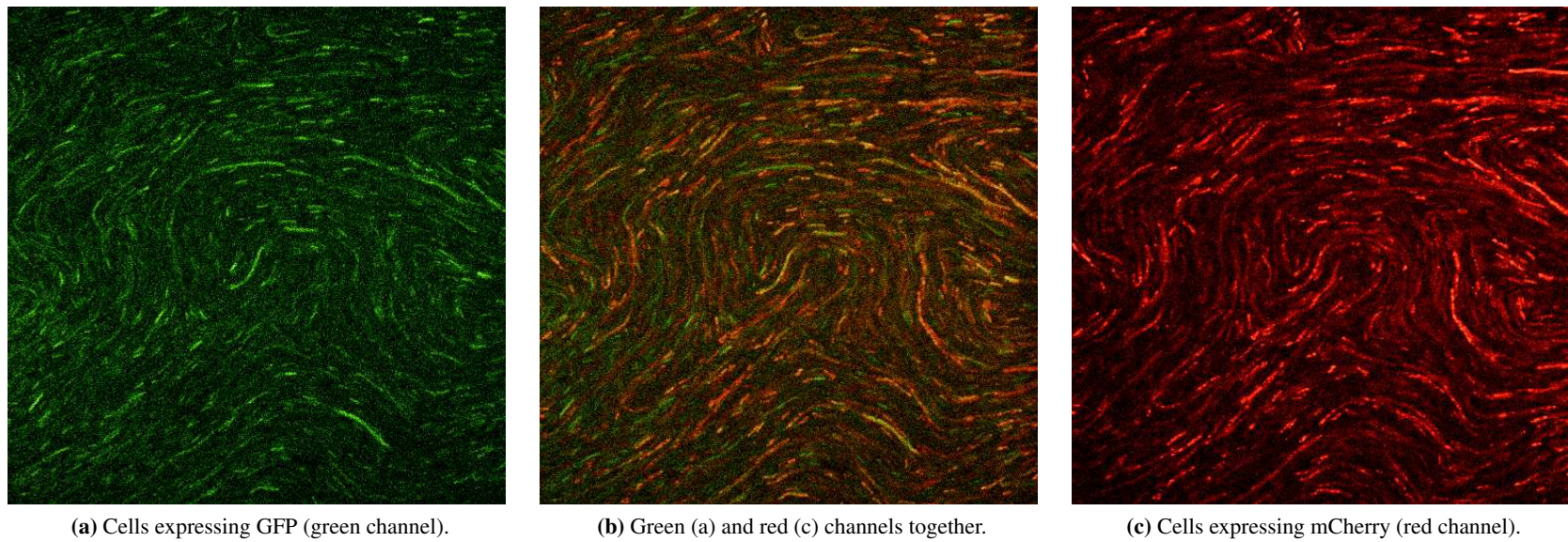
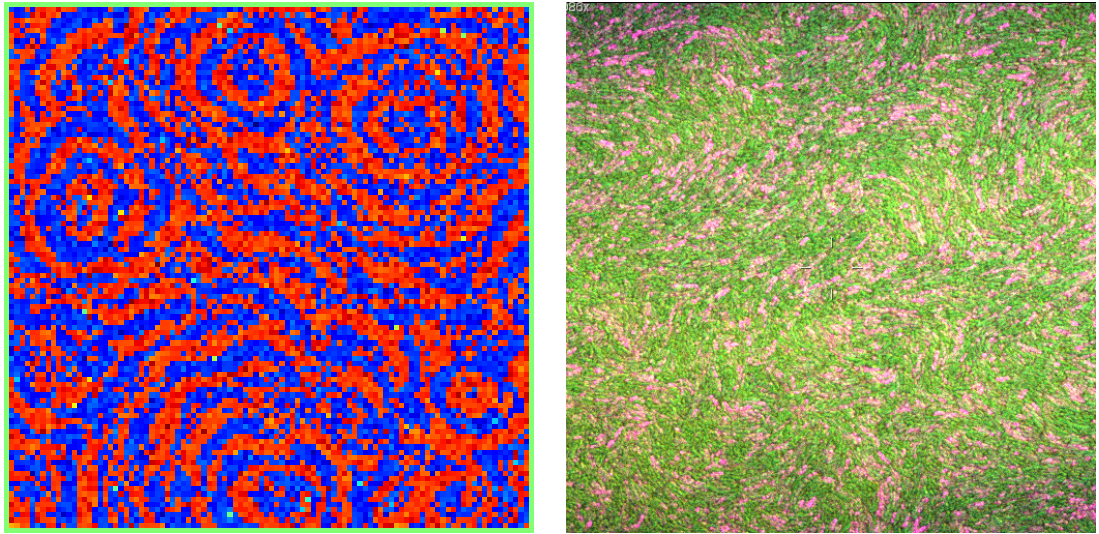


Figure 5.18: Confocal microscopy of bacteria transformed with the pattern formation circuit from figure 1.4a. Images courtesy of Dr. Karima Righetti.



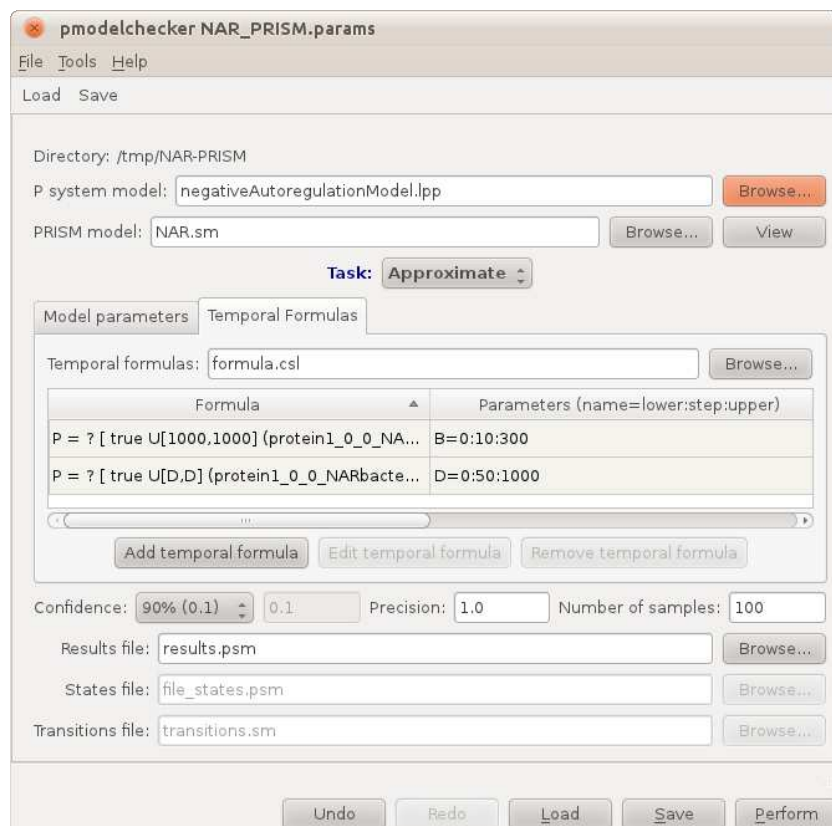
(a) Target pattern produced by model cells equipped with designed circuit. (b) High gain image showing both GFP and mCherry produced alternately by cells with chromosomally integrated circuit DNA.

Figure 5.19: Comparison of *in silico* (a) and *in vivo* (b) functioning of synthetic Turing pattern formation circuit.

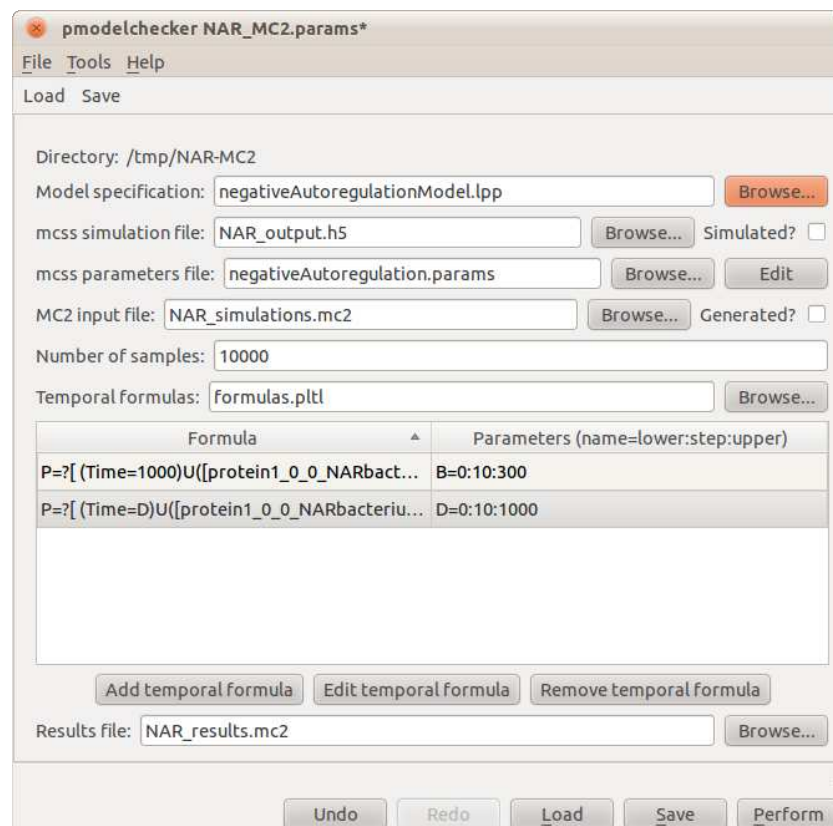
system models by acting as wrapper interface between LPP systems and the model checkers PRISM [35, 247] and MC2 [224, 248, 249].

To perform probabilistic model checking with PRISM, LPP systems are loaded and automatically converted into a Reactive Modules specification [223] that PRISM can accept as input. Parameters are created for the lower and upper bounds of the number of molecules of each species in each compartment: the user defined values of which are used to constrain the potential state space of the PRISM model. PRISM is then called to perform *approximate/statistical* model checking using its own discrete event simulator, performing simulations up to a specified maximum number of runs or confidence threshold. The state space and the generated transitions matrix can also be used to “Build” the complete Markov chain and then “Verify” whether each property is satisfied in all states of the model. Verification is generally infeasible for all but very small models due to the size of the underlying Continuous Time Markov Chain (CTMC), but can be useful for checking critical components of small reaction networks, such as the synthetic bioparts.

To perform *simulative* model checking with MC2, previous simulation results can be reused or a new simulation with a large number of runs to achieve higher confidence in the model checking results can be performed. With model checking, properties such as the probability of a species exceeding a certain threshold after a certain time can be determined to a specified degree of confidence (corresponding to the number of independent simulation runs for simulative model checking).



(a) PRISM interface



(b) MC2 interface

Figure 5.20: PMODELCHECKER parameterisation interfaces

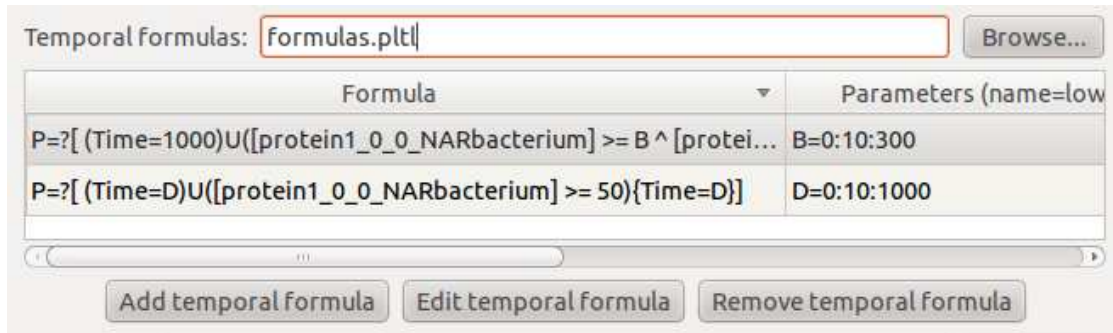


Figure 5.21: Temporal formulas interface.

The Infobiotics Dashboard provides two parameterisation interfaces to PMODELCHECKER, one for each of the model checkers it uses, as some of the parameters are specific to one but not the other. Figure 5.20 contrasts the PRISM and MC2 interfaces showing how the P system model, Temporal Formulas and Results file parameter widgets are common to both.

Temporal formula parameters

PRISM properties are specified in Continuous Stochastic Logic (CSL) - an extension of Probabilistic Continuous Time Logic (PCTL) for CTMCs. MC2 properties are specified in Probabilistic Linear Time Logic (PLTL), the discrete time steps of which correspond to the logging interval of the simulation.

Multiple formulae can be loaded from, and must be saved to, a file. The currently selected formula can be edited or removed, or a new formula added via the respective buttons below the table widget (5.21). Formulae are edited manually and can be parameterised with variables that are finite ranges with equal steps, with the dialog shown in figure 5.22.

Valid model parameters can be chosen from a combo box and inserted, while non-negative real-valued formula parameters can be added or removed and their lower and upper bounds specified, along with a step that determines the number of values a parameter can take and the interval between them. Assistance can be obtained via the Help button which opens a new slimline web browser window on the Property Specification page of the PRISM Manual, as shown in figure 5.23.

PRISM-only parameters

The PRISM model that is produced automatically (by the Dashboard calling PMODELCHECKER when the P system model parameter changes) can be inspected via the **View** button to the right of the PRISM model parameter field (figure 5.24).

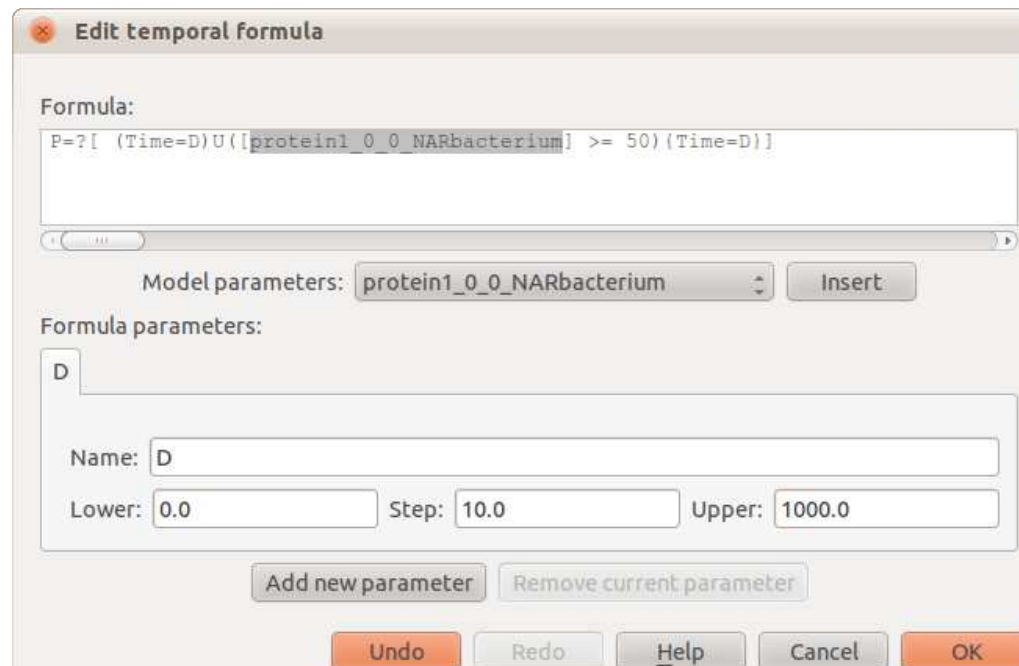


Figure 5.22: Editing a temporal formula.

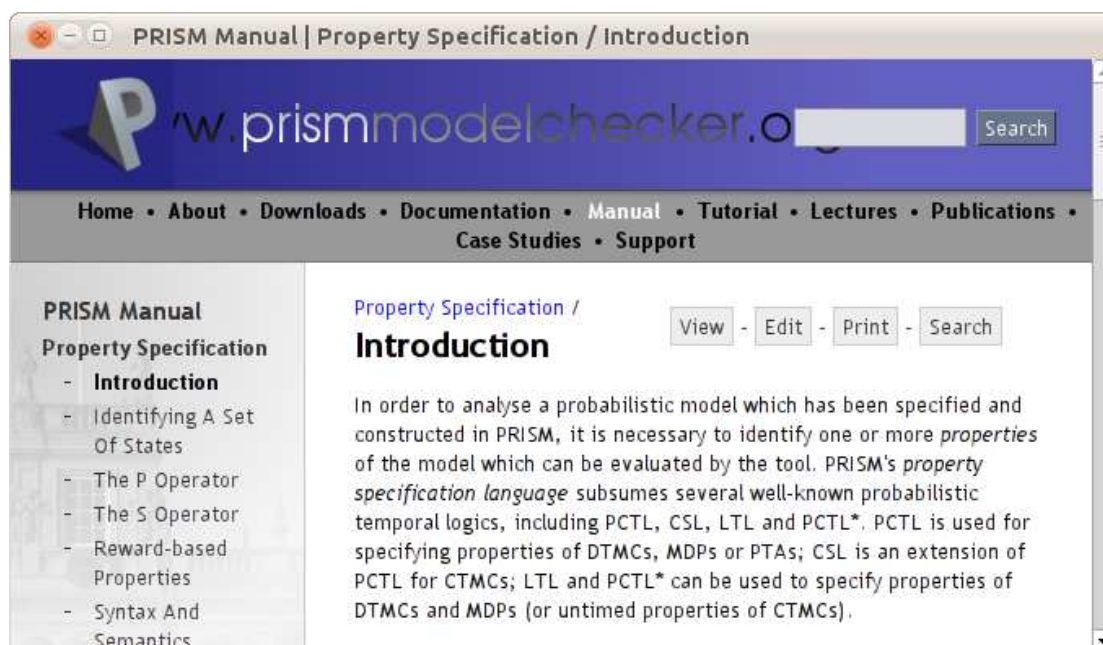


Figure 5.23: Help with temporal formula syntax for property specification is delegated to the PRISM manual.

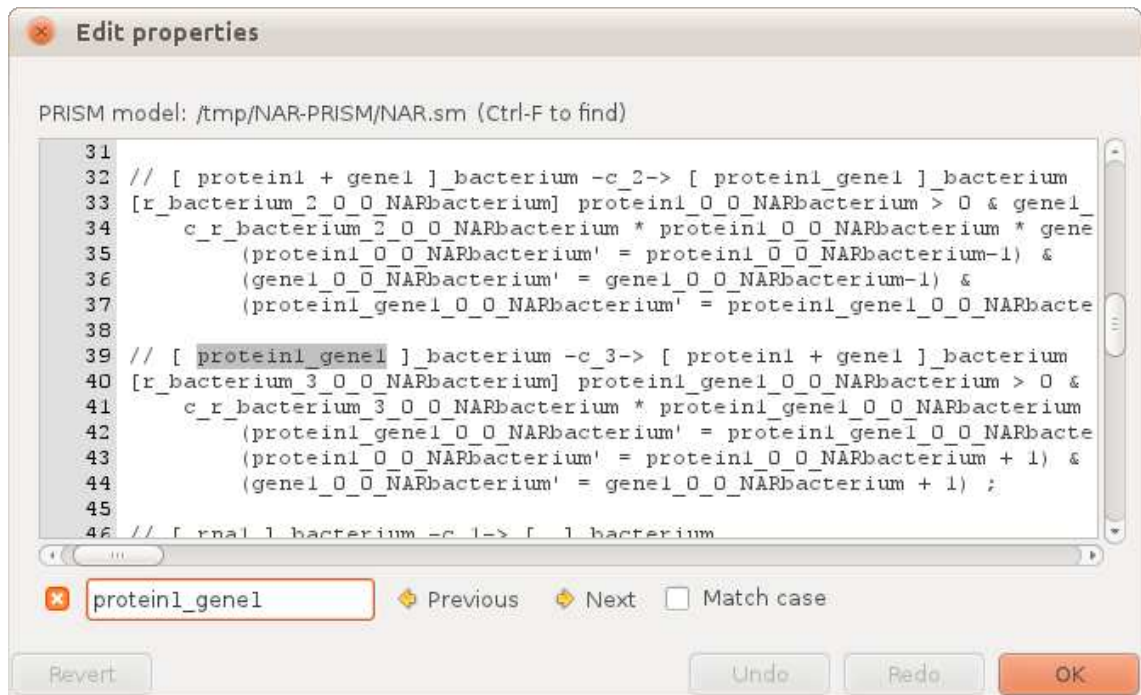


Figure 5.24: Viewing the generated PRISM model

The next step for a PRISM model checking experiment is to choose a task (figure 5.25a) and name the output files relevant to the particular task (figure 5.25d). In order to build construct a state-space lower and upper bounds (non-negative integers) for the number of molecules of each species in each compartment (the **model parameters**) must be specified. Model parameters names are part of the PRISM model computed by PMODELCHECKER, so to clarify their meaning a description field accompanies each one (figure 5.25b). The degree of confidence to obtain for approximate model checking, viewed as a percentage but passed to PRISM as a fraction, can be set to 90%, 95%, 99%, or a *custom* fraction between 0 and 1 (figure 5.25c).

MC2-only parameters

In contrast to PRISM, which uses the bounds of model parameters to construct a state-space and perform a discrete event simulation from which to calculate the probabilities of each formula being satisfied, MC2 accepts suitably transformed pre-simulated timeseries from which it computes probabilities.

The MC2 model checking experiment interface intelligently handles the possible combinations of PMODELCHECKER parameters (shown in figure 5.26) which concern loading previously simulated results, previously generated MC2 input or performing new simulations and regenerating MC2 input, by disabling parameters and changing parameter validation constraints as the usage

PRISM model:

Task: **Build** (highlighted in orange)

Approximate
Verify

Model parameters Temporal Formulas

(a) Available tasks for models checked with PRISM.

Model parameters Temporal Formulas

Molecule constants:

Name	Value	Description
lb_gene1_0_0_NARbacterium	0	Lower bound for gene1 in compart...
ub_gene1_0_0_NARbacterium	1	Upper bound for gene1 in compar...
lb_protein1_0_0_NARbacterium	0	Lower bound for protein1 in comp...
ub_protein1_0_0_NARbacterium	10000	Upper bound for protein1 in comp...

(b) Parameters derived from the model that constrain the state-space to be checked.

Confidence: Precision: Number of samples:

(c) The confidence level to obtain before acceptance, precision to adjust parameters by, and number of samples (executions) to perform.

Results file:

States file:

Transitions file:

(d) Naming of files which will be created by PRISM.

Figure 5.25: PRISM-only PMODELCHECKER parameters.

Figure 5.26: MC2 input parameters. The number of samples is the number of simulation runs to perform and analyse.

(a) Simulated MCSS output files must exist, negating parameterisation of a new simulation.

(b) Generated MC2 input files must exist.

Figure 5.27: Reusing previously simulated results or generated MC2 input places additional constraints on these parameters.

changes. Users must specify an MCSS simulation results file and MC2 input file into which these results are transformed and tell PMODELCHECKER whether or not these have been previously simulated and generated respectively (in which case each file must exist: figures 5.27a and 5.27b).

If not previously simulated, new simulations can be parameterised from an MCSS parameters file and a *subset* of these simulation parameters can be further edited (figure 5.28); while the `data_file` and `runs` parameters are delegated to the `mcss_simulation_file` and `number_of_samples` parameters in the main interface.

Figure 5.29 shows a model checking experiment running under Linux, with the current property displayed above the percentage completion and an estimate of the time remaining. Under Windows, due to unavoidable buffering of standard output, it is not possible to catch the output of the model checkers invoked by PMODELCHECKER, instead a terminal is opened showing this output as the program is running to provide at least some feedback as to its progress. Unlike simulation with MCSS, it is also not possible to preserve a truncated set of results if the experiment is cancelled midway through.

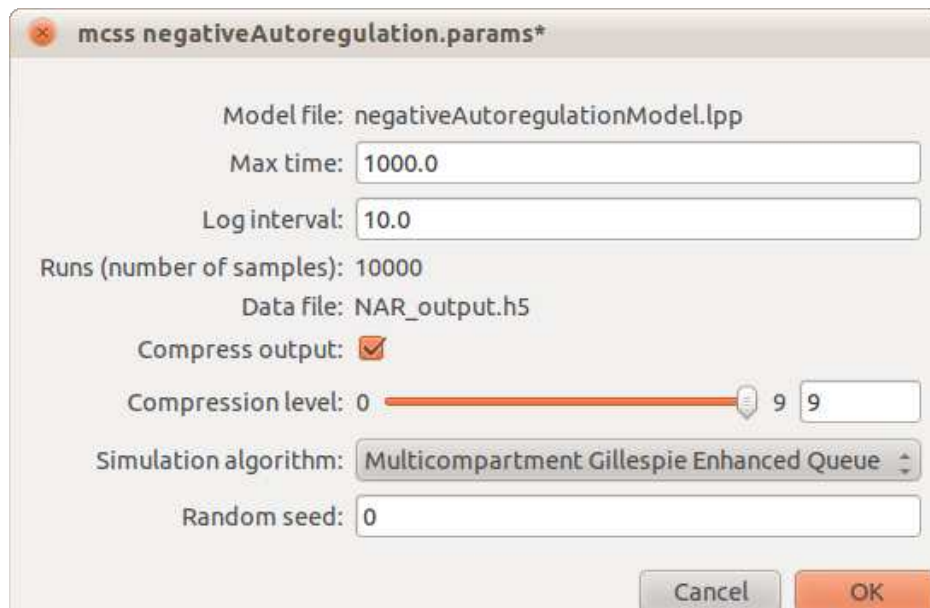


Figure 5.28: Editing the parameters of a new simulation. New simulations are partially parameterised by the MC2 experiment.



Figure 5.29: A running model checking experiment

Model checking results

Once a model checking experiment has completed the results interface is loaded from the file specified by the `results_file` parameter. The output is the same for either model checking experiment: for each formula a list of the probability of each property being fulfilled for each combination of formula parameters, usually time plus several others (e.g. figure 5.30a). The varying probabilities of each property can be plotted in two ways: a 2D plot of the probability that the property is satisfied against all values of one variable (figure 5.30b) or a 3D plot of probability against all values two variables (figure 5.30c), at a single value of each remaining variable. The constant values of the remaining variables can be set using sliders which are dynamically added to the results interface above the plot depending on availability and the currently selected axis variables. In this way both 2D and 3D plots can be used to visualise queries with greater numbers of variables, enabling the results of N-dimensional queries to be interrogated in a consistent manner.

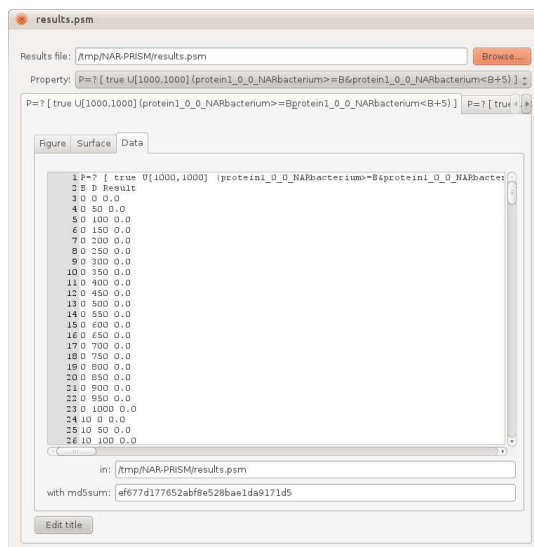
5.4 Parameter and model structure optimisation with POPTIMIZER

POPTIMIZER, developed initially by Dr. Hongqing Cao and later Dr. Claudio Lima, is the model optimisation component of the Infobiotics Workbench. Optimisation is the process of maximising or minimising certain criteria by adjusting variable components of a model, fitting simulated behaviour (quantitative measurements sampled at various time intervals) to observed or desired behaviour in the case of natural or synthetic biological systems respectively. There are two aspects of P system models that can be readily varied to optimise temporal behaviour:

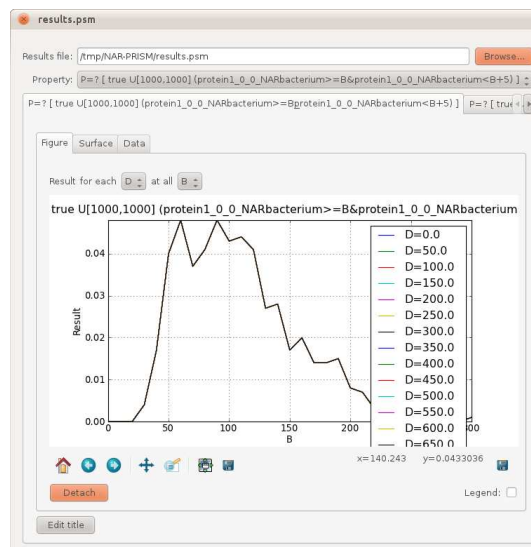
1. numerical model parameters - the values of the stochastic rate constants associated with rules can be tuned to fit the given target
2. model structure - the composition of the rulesets governing the possible state transitions of the compartments can be altered to produce alternative reaction networks that recreate the target dynamics more precisely

Both seek to *minimise* the distance between the stochastically simulated quantities of molecular species and a set of user-provided values of the same species at each target timepoint; a quantitative means of evaluating the fitness of candidate models and discriminating between them in an automated manner.

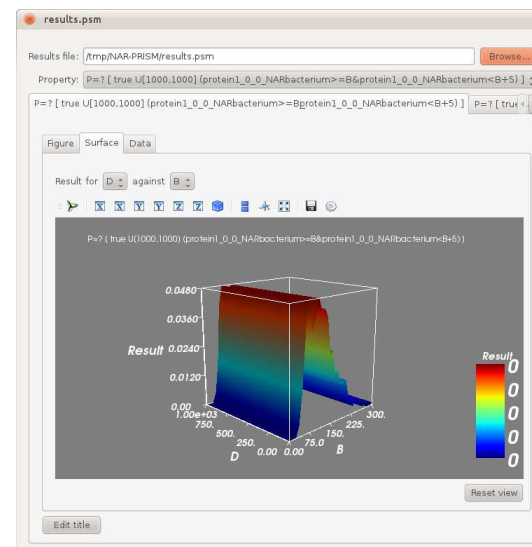
POPTIMIZER search the parameter and structure spaces of *single compartment* stochastic P systems with custom own implementations of state-of-the-art population-based optimisation algorithms: Covariance Matrix Adaptation Evolution Strategies (CMA-ES) [250], Estimation of Distribution Algorithms (EDA), Differential Evolution (DE) [110] and Genetic Algorithms



(a) Model checker results data.



(b) 1 variable 2D plot.



(c) 2 variable 3D plot.

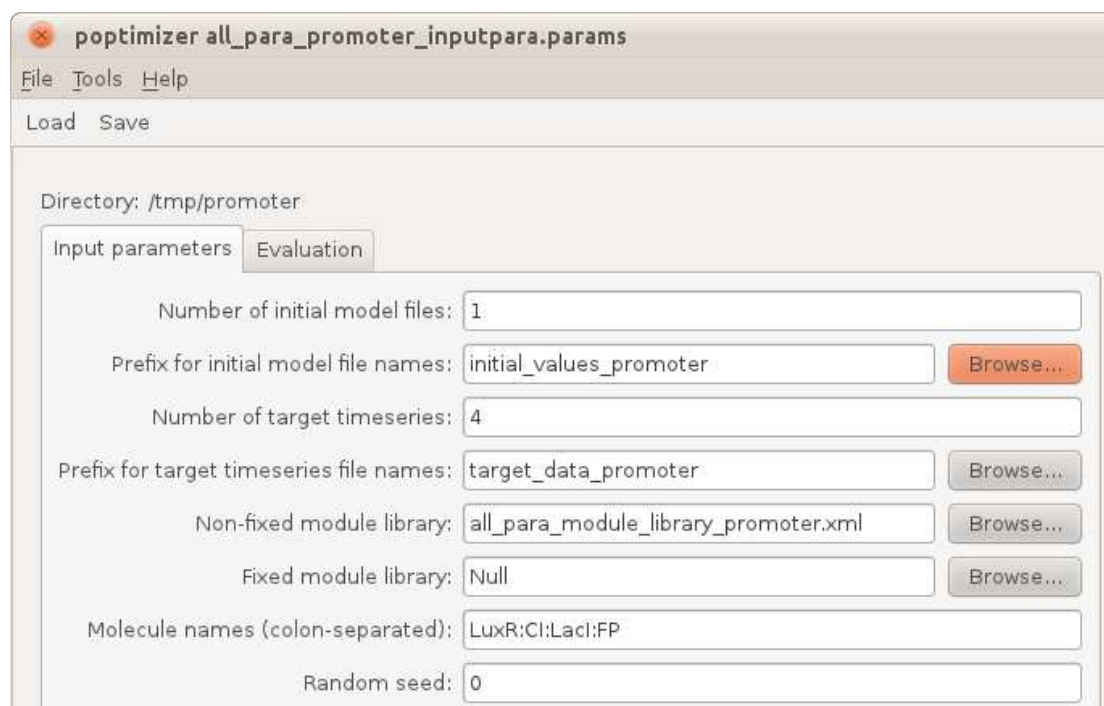
Figure 5.30: Model checking experiments results interface. The same interface is used for both PRISM and MC2 results (in this instance PRISM).

(GA) [108]. Optimisation is limited to single compartment models, partly due to the increased complexity of algorithmically manipulating spatially distributed or hierarchically organised compartmental structures (and the distinction made between these by the LPP formalism), but more pragmatically because repeated stochastic simulation of each individual in a population of (potentially unfeasible) single compartment models (with suboptimal rate constants) is very computationally expensive. Simulating many copies of those compartments, interacting on a 2D lattice would multiply the cost and providing suitable or accurate target data would be difficult also. Thus model optimisation is generally only tractable with smaller models (as with model verification). However, submodels can be optimised in isolation and then reintegrated, provided they can be decoupled: the assumption made by the modularised, engineering approach to synthetic biology.

Model structure optimisation is an automated means of generating alternative hypotheses about the network of molecular interactions underlying real biological system for which the actual structure is unknown, or suggesting suitable networks that recreate a desired dynamics. We distinguish between known or *fixed* components of the system that are included in all candidate models, and unknown or *non-fixed* components that can vary between models. For the non-fixed components, instead of generating new P systems rules without constraint, POPTIMIZER utilises the knowledge captured in a library of P system modules to add or remove biologically plausible sets of rules representing reactions as one. By recombining and mutating modules that capture realistic network motifs the space of models is somewhat constrained, significantly improving the quality and realism of the models produced.

POptimizer uses a *nested genetic algorithm* [251, 252] to generate a set of candidate models, initially by random choice and thereafter by mutating the fittest individuals of the previous generation, and performs several rounds of parameter optimisation on each individual to ensure that the candidates are given a fair chance of fitting the desired behaviour (as previous rate constants may be unsuited to the updated reaction network) before using the final fitness to select the next generation. Parameter optimisation can of course be performed without structure optimisation if the user is confident in their model's structure.

Figure 5.31 shows the input parameters of an optimisation experiment. When performing structure optimisation a library of modules from which the non-fixed part of the model structure can be constructed must be specified. A set of fixed modules, that will always be part of the model, may also be specified, enabling the modeller to embed information about known/hypothesised model structure around which optimisation can take place. To facilitate an effective exploration of the parameter spaces of each rate constant, users can augment (in the module library XML/.plb input files, see section 4.3.2) the module definition of rate constants with lower and upper bounds, precisions and linear or logarithmic scales, which POPTIMIZER then uses to



The screenshot shows a web-based application window titled "poptimizer all_para_promoter_inputpara.params". It has a menu bar with "File", "Tools", and "Help". Below the menu bar are "Load" and "Save" buttons. The main content area has a "Directory: /tmp/promoter" label. There are two tabs: "Input parameters" (selected) and "Evaluation". Under the "Input parameters" tab, there are several input fields and buttons:

- "Number of initial model files:" with a text input containing "1".
- "Prefix for initial model file names:" with a text input containing "initial_values_promoter" and a "Browse..." button.
- "Number of target timeseries:" with a text input containing "4".
- "Prefix for target timeseries file names:" with a text input containing "target_data_promoter" and a "Browse..." button.
- "Non-fixed module library:" with a text input containing "all_para_module_library_promoter.xml" and a "Browse..." button.
- "Fixed module library:" with a text input containing "Null" and a "Browse..." button.
- "Molecule names (colon-separated):" with a text input containing "LuxR:Cl:LacI:FP".
- "Random seed:" with a text input containing "0".

Figure 5.31: POPTIMIZER input parameters.

constraint the values of parameters and thereby reduce the size, if not the dimensionality, of the search space.

Because the output of a model can be highly dependent on the input, multiple sets of initial conditions (the molecule counts of each species) can be specified as independent models, as can multiple sets of target timeseries, so that the behaviour of the final model produced is robust over a range of inputs and desired outputs. Figure 5.32 shows how the Infobiotics Dashboard communicates the required format of target timeseries input files to users with them needing to consult the documentation, although this is easily accessible through the help menu in figure 5.33 (which opens the relevant page of the Infobiotics website in the system web browser).

Figure 5.34 shows the parameters concerned with fitness evaluation, model structure and parameter optimisation. A maximum number of modules per model can be specified to select for more parsimonious, and generally comprehensible, solutions. Parameter optimisation takes place after each round of recombination, to help realise the potential of the new structures that might not be initially apparent. To perform parameter optimisation in isolation only a fixed module library is required and the number of structural optimisation generations should be set to 1.

The fitness of candidate models is calculated using either *random-weighted* or *equal-weighted sum* functions of the distance from the produced timeseries to the target timeseries for each species (figure 5.35). In experiments [252] using POPTIMIZER the random-weighted sum fitness function was found to produce a better convergence to designed target models when some

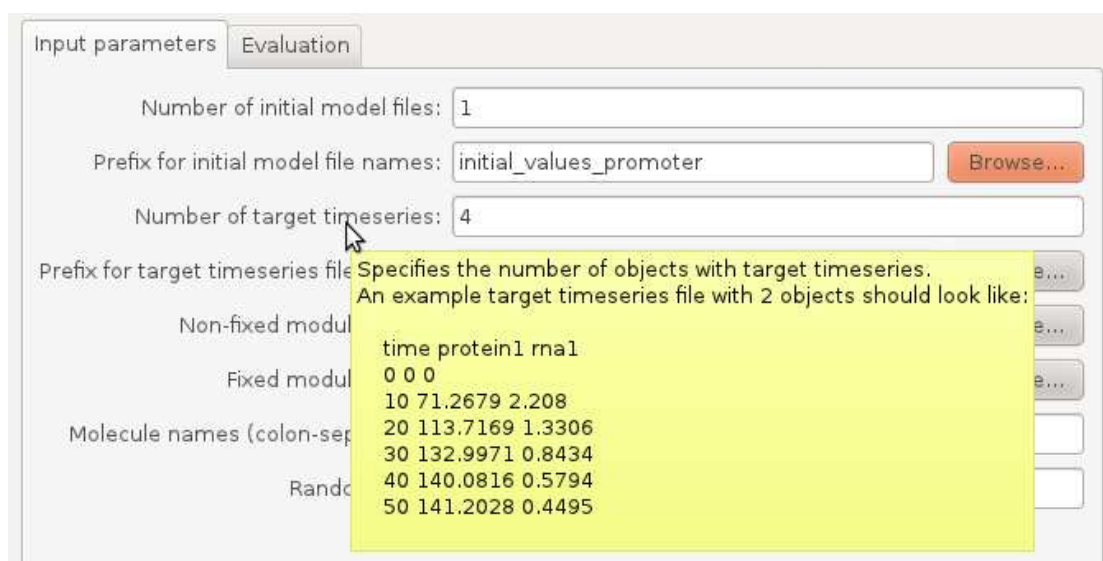


Figure 5.32: Target timeseries tooltip.



Figure 5.33: Accessing POPTIMIZER documentation via the Help menu.

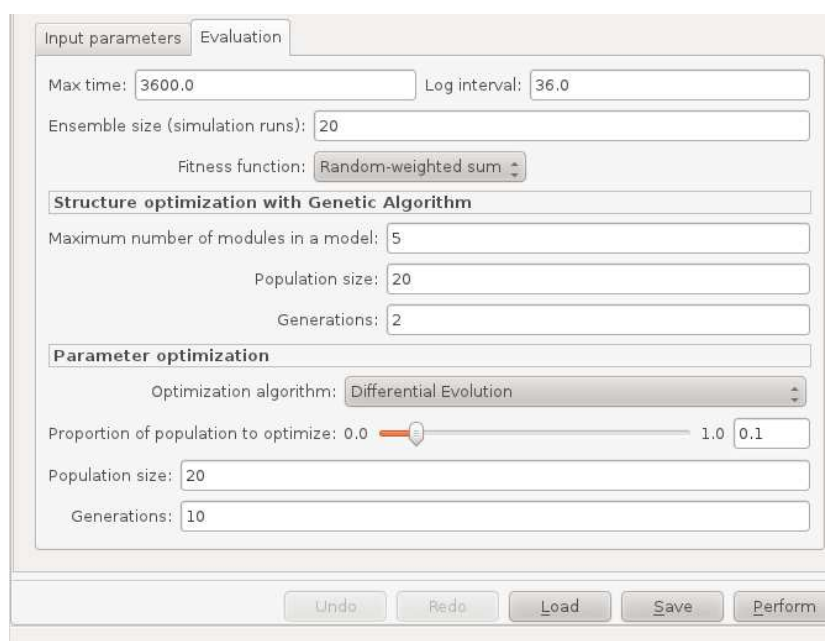


Figure 5.34: POPTIMIZER evaluation and optimisation algorithm parameters.



Figure 5.35: A choice of two fitness functions is offered to discriminate between candidate models.

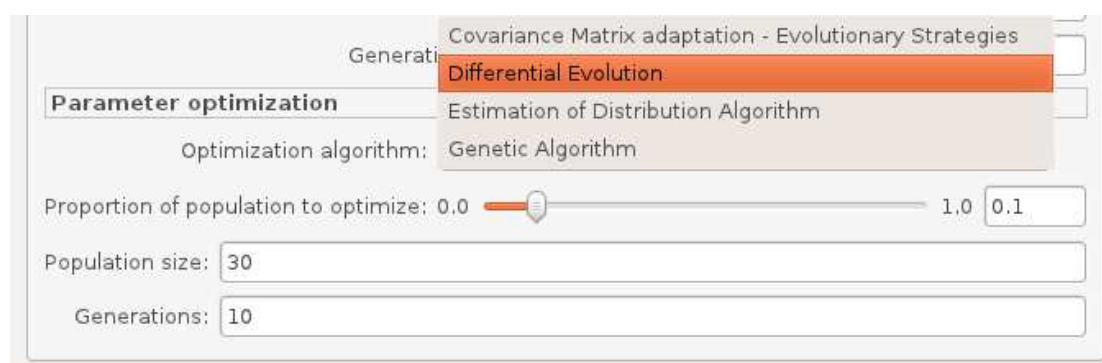


Figure 5.36: The four parameter optimisation algorithms provided by POPTIMIZER.

of the target output timeseries of the models had quantities orders of magnitude larger than others. This is particularly relevant to cellular systems where the number of transcribed messenger RNAs does not necessarily correlate with the number of proteins translated from them. With the equally-weighted sum fitness function, the search can be biased towards optimising the timeseries of higher concentrations, when those in lower concentrations could be of equal or greater importance.

Figure 5.36 shows the four parameter optimisation algorithms available. Sensible defaults are provided for population size and generations (not applicable for CMA-ES, hidden if selected), taking into account that the algorithm will be run for each generation of the structure optimisation GA.

The output of an optimisation experiment is the fittest model produced. For a visual comparison of the output models suitability and the optimisation algorithms success, timeseries of the target and the optimised output are plotted for each species, as shown in figure 5.37. A summary of the experiments inputs and the modules that comprise the optimised model are captured from POPTIMIZER and displayed alongside the timeseries.

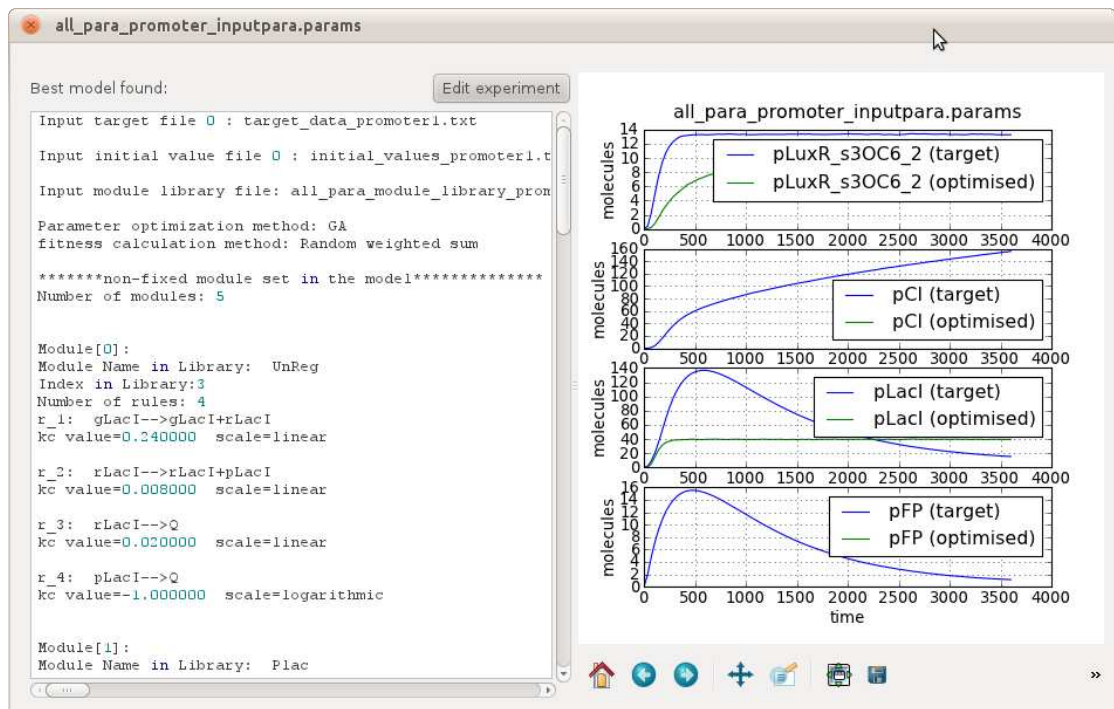


Figure 5.37: POPTIMIZER results interface.

5.5 Summary

The Infobiotics Workbench is an integrated *in silico* platform for computer-aided modelling and design of large-scale biological systems incorporating simulation, formal verification and optimisation algorithms. The optimisation components of the Workbench enables designs of synthetic circuits matching a set of desired temporal dynamics (specified as timeseries of molecular species) to be automatically composed from modules of abstract networks motifs and/or completely specified bioparts (with corresponding DNA sequences) drawn from libraries of reusable model components. These circuits are simulated stochastically and tuned against a variety of initial conditions. The availability of deterministic and stochastic simulation of population models enables comparisons between macroscopic and mesoscopic interpretations of molecular interaction networks. Model checking can be used to increase confidence in simulated observations by quantifying the probability of reaching definable states for all possible trajectories.

The computational expense of using population-based evolutionary optimisation techniques, which require multiple runs of stochastic simulation to evaluate each individual at each generation, places practical limits the size of the systems which can be automatically designed and optimised; currently to a single cellular compartment. Therefore simulations of candidate circuits optimised in this context may not accurately reflect their performance when placed in the context of a population of cells. For instance, when diffusible molecules produced by other instances of those circuits - or other synthetic devices in the same compartment or other cells

- in some way regulate the circuits operation, then emergence may begin to exert its effects on the dynamics of the system at large. To evaluate these effects, users can simply add the module invocations defining the optimised circuits to existing stochastic P system models of cellular chassis, then use these to build LPP system models with specific spatial arrangements of cells, and perform simulations or model checking.

The simulation results capabilities of the Infobiotics Dashboard enables molecular populations to be animated as a surface over the cellular population on the lattice for a visually rich semi-quantitative analysis of behaviour in space as well as time. Timeseries of molecular quantities (as concentrations or number of molecules) in individual or averaged simulation runs can be plotted for any combination of species, compartments and timepoints, enabling a fine-grained quantitative comparison of expected and simulated temporal dynamics at multiple locations in spatial models. Histograms can be used to estimate the distributions of molecular species in cells or runs at different timepoints, possibly revealing differentiation of cell states as initially homogeneous populations become heterogeneous through emergence.

Further information, tutorials and examples are available at the Infobiotics website (<http://www.infobiotics.org/>). The community can report bugs or request features at Infobiotics Dashboard Bitbucket repostory issue tracker (<http://bit.ly/qn9pUA>).

In the next chapter we move from the design of individual models of synthetic biological devices to families of related devices exploring biological problem-spaces with a scalable new approach to the specification and synthesis of combinatorial DNA libraries: the DNALD language.

Chapter 6

DNALD: a language for DNA Library Design

Chapter abstract

The aim of this chapter is to elucidate the requirement for and specification of the DNALD language that will enable the formulation of a library of output DNA molecules as a function of combinatorial assemblages of the input and intermediate sequences DNA sequences. This work is ongoing within the CADMAD project, involving research groups from 7 European partner institutions, which aims to develop a novel synthesis platform based on DNA reuse. The research in Chapters 6, 7 and the latter half of Chapter 8 represents one year of work by the author as part of this three year project.

6.1 Background

Artificial or *de novo* DNA synthesis is the process of synthesising genes or other gene length sequences of nucleotides *in vitro* without a template strand. Gene synthesis can save time and money compared to conventional techniques, relieving skilled researchers from the manual labour of DNA editing with restriction enzymes and PCR. In addition to the planned incorporation (or avoidance) of subsequences required for further manipulation (affinity tags, antibiotic selection markers, multiple cloning/restriction sites) [253], synthesis provides access to sequences that have proven difficult to clone or are not found in nature, allowing rational redesign of sequences for improved protein expression and promoter binding, as well as novel uses in other fields such as DNA origami nanotechnology [254].

6.1.1 Artificial DNA synthesis

The primary method is ligation of chemically synthesised oligonucleotides 15-25 bases in length¹, of the kind used as primers for DNA amplification, antisense sequences and probes. DNA synthesis companies such as DNA 2.0 or GENEART use proprietary protocols to assemble single-stranded oligonucleotides, which are then cloned, complemented and the sequences verified.

¹The error rate for oligo synthesis grows linearly with sequence length, with 200 base Ultramers (from IDT: <https://www.idtdna.com/pages/products/dna-rna/ultramer-oligos>) being the longest presently available.

Synthesised DNA sequences are delivered in 2 – 5 μ g (microgram) quantities which researchers amplify using PCR to maintain a constant supply. Quality is certified by the full nucleotide sequence of the delivery plasmid and insert, which can be compared against downloadable chromatogram traces covering both strands.

Synthetic DNA lengths can range from hundreds of base pairs through tens of kilobases, to just under half a megabase at the upper limit (<https://www.dna20.com/index.php?pageID=17>). Beyond these sizes stitching is required [44].

6.1.2 DNA libraries

The ability to synthesise arbitrary sequences of DNA enables the construction of a library of DNA sequences sampling a biological problem space. The effects of each sequence on the system under study can be screened in parallel, reducing the likelihood of small scale, undetectable processing errors that can occur when repeating a protocol on different occasions, and thereby improving reproducibility and debuggability of the experiment. This approach enables the systematic investigation of biological problems at a larger scale and less ad hoc basis.

Constructing a DNA library requires a specification of the set of required sequences to synthesise, that is complete, correct and compact. Both the specification and the synthesis should scale efficiently, enabling libraries of increasing size and complexity to be manufactured. The scalability of a library, and its specification, can be viewed as amount of overlap between its sequences, because each identical shared subsequence (down to a reasonable minimum length) can be synthesised first and then reused in the construction of the other complete sequences. Where multiple variant sequences differ at the same point - from a single nucleotide substitution to being composed of this or that synthetic biopart - the library can be termed *combinatorial*. Going deeper, it is possible to consider the series of operations that would be required to transform one sequence into another, and determining the starting sequence that represents the shortest path to the remaining variants. Factoring out common subsequences and editing previously synthesised sequences are both ways of producing, in several steps, DNA libraries in a scalable manner.

The chemical DNA synthesis services offered by GENEART and DNA2.0 manufacture each sequence sequentially oligo-by-oligo, base-by-base, which does not take advantage of the combinatorial nature of DNA libraries. By this method, the production of two almost identical sequences proceeds in parallel but without cross-talk, and costs scale linearly as a result.

The CADMAD platform (<http://www.cadmad.eu/vision>) aims to maximise DNA reuse in the synthesis of combinatorial DNA libraries, by optimising the construction plan around the most reusable intermediate sequences and, where errors occur, recursive construction of perfect DNA molecules from imperfect oligonucleotides [22]. The key to CADMAD's approach to DNA reuse is the recursive "Y-operation", shown in figure 6.1. Each Y takes a left and a right input

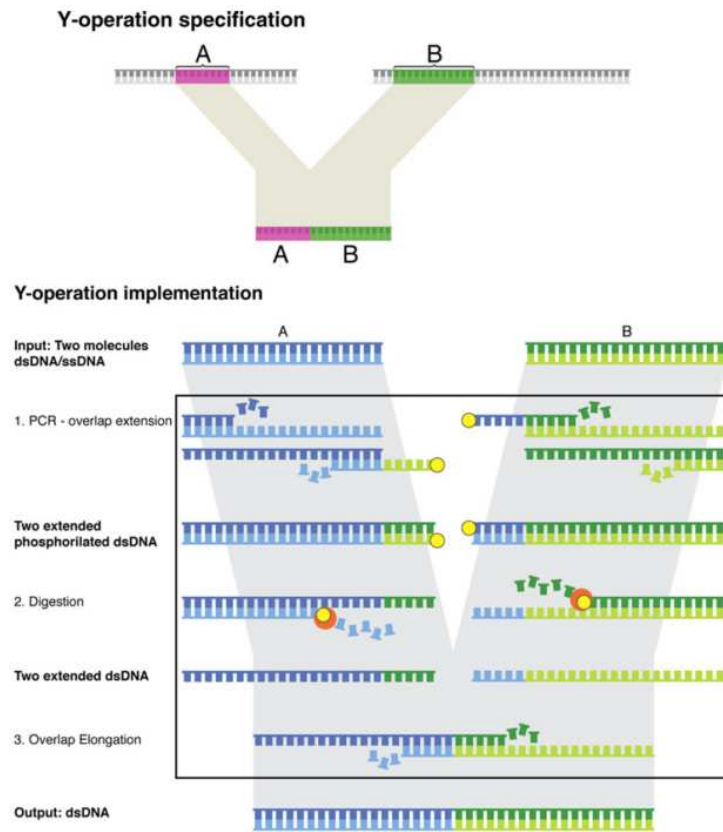


Figure 6.1: A single Y-operation, the basic recursive unit of CADMAD DNA synthesis platform, reproduced from [21].

DNA molecule and outputs a third, the concatenation of right to left, which can be reused as an input to another Y-operation. The choice of primers used to extract the starting sequences enables editing operations to be performed, e.g. a single nucleotide substitution, in a manner analogous to how we edit text in a word processor (cut, copy, paste, etc.). The robotics platform that performs these operations can be thought of as the 21st century, DNA-producing equivalent of Gutenberg's movable type printing press of the 1450s.

Given the set of required sequences, a construction plan can be algorithmically designed in terms of the ordering of Y-operations necessary to produce each required sequence, and then join those to produce all target sequences, maximising the reuse of intermediate sequence fragments between targets. In addition to the scalability advantages of subsequence reuse, there is a reduction in the error rate as following each Y-operation the outputs are sequenced and determined to be correct; sharing these improvement between targets is analogous to the manner in which bug fixes in widely used programming libraries propagate improvements throughout an entire software stack.

6.1.3 Programming combinatorial DNA library specifications

Before a construction plan involving Y or any other operations can be determined, the set of sequences comprising the DNA must first be designed. Assuming a suitable synthesis platform for combinatorial DNA libraries such as CADMAD exists, the problem then is how best to solicit requirements from library consumers, and enable the communication of sequences and intentions from library designer to manufacturer (and the verification of those requirements at both ends of the pipeline).

DNA sequences, being essentially one long word in an alphabet of four letters, are notoriously inscrutable and cumbersome to work with as the length and number of sequences grows. For large, highly combinatorial libraries computer aided design is essential. A large body of algorithms and software exists to handle, annotate, align and compare sets of sequences, but to the best of our knowledge no software exists that describes libraries of sequences in terms of the combinatorics of sequence fragments for library synthesis. While it is possible for programming-literate biologists to use general purpose programming languages (Java, Python, Perl or R) supplemented with bio-oriented libraries (BioJava, Biopython, BioPerl or Bioconductor) to generate sets of sequences, there is no established method for doing so, resulting in many, varied, one time (i.e. not reusable) efforts.

We believe that in order to be able to efficiently and correctly design DNA libraries it is necessary to abstract away from raw sequences to a representation that enables reasoning about and composition of fragments through operations on **sets of sequences**. A programming language specifically for defining DNA libraries could provide a *lingua franca* to the synthetic systems biology community and avoid error-prone reimplementations of sequence generating scripts.

6.2 The DNALD language

The first stage towards the creation of a DNA library is the formal specification of the DNA molecules that comprise it. This process must be user friendly, easy to debug and yet it has to provide the user with enough expressive power to specify non-trivial notions such as complex combinatorics and certain degrees of freedoms which the manufacturer may have with respect to producing the requested DNA sequences.

Certain tools for editing and manipulating DNA strings were developed long ago. In recent years a new system called GENOCAD [234] was developed, offering a GUI based on attribute grammars, to design biologically plausible DNA molecules based on known building blocks. High-end closed-source/commercial packages, e.g. Gene Designer 2.0² from DNA 2.0, provide advanced features for designing cloning sequences and plasmids. These features include

²Gene Designer 2.0 <http://www.dna20.com/genedesigner2/>

sophisticated GUI, project management, codon-usage optimizations, restriction site handling, etc. However, similarly to GENOCAD, they are intrinsically geared towards building high-level structured entities (e.g. plasmids) rather than multiple, combinatorially dependent DNA sequences, which are more likely to advance research and development in biotechnology and nanotechnology. In addition, these tools are limited in their expressive power, without a notion of degrees of freedom, such as using ambiguous nucleotide codes to specify a set of alternatives for a particular base to dramatically simplify the specification of multiple closely related constructs.

We have recognised and addressed this need by creating a DNA programming language (DNALD - DNA Library Design) that will permit the seamless specification of large combinatorial libraries. DNALD will have a major impact on the way scientists, and their programs, specify DNA libraries and we plan to expand its influence through the creation of a visual programming counterpart, providing an illustrative means of reasoning about and communicating DNA library combinatorics, built on a proven textual backend.

To bootstrap the language design process and ensure that is relevant to a broad range of molecular, systems and synthetic biological applications, we have worked in collaboration with European partners to gather application-specific requirements and preliminary DNA library designs. Our current set of use cases includes: protein crystallization, post-transcriptional regulation through RNA secondary structure or DNA-based nanotechnology. Section 6.3 details the design of a DNA library using DNALD which investigates the role of secondary structures in post-transcriptional regulation of the azurin gene in *Pseudomonas aeruginosa*.

6.2.1 Origins

Although we say DNALD is a new language for the specification of combinatorial DNA libraries, in fact it has a short but important history. DNALD is a refinement and extension of DNAPL prototype language developed by Yair Mazor, Uri Shabi and Ehud Shapiro at the Weizmann Institute, from whom we have inherited the responsibility of bringing the idea to its fullest fruition. DNALD inherits some of the structure and intentions of DNAPL whilst rationalising, refining and extending the operations, syntax and semantics to make it more capable, consistent, efficient to parse and evaluate, and amenable to future extension. To evince the extent of our contribution, and as a specification for the core of DNALD, we present a brief overview DNAPL's structure and syntax. We then analyze its features and explain how these have informed the development of DNALD.

A DNAPL library is a single file containing definitions of sequences as expressions editing and recombining previously defined sequences. These definitions are divided between three *sections* to form three disjoint subsets: *inputs*, *intermediates* and *outputs*. Each section has a slightly

different meaning which in term places constraints on the content of the definition expressions permitted:

Inputs are existing fragments of DNA that the library design can provide: a single unambiguous DNA sequence such as a sequenced plasmid. Input definitions cannot therefore contain DNAPL operations that would yield a new sequence that does not exist. They can be referenced by name in intermediate or output definitions.

Intermediates are definitions that can be referenced by name in other intermediate or output definitions but are not directly required as an output of the library. Intermediates are therefore either variations on inputs, outputs, other intermediates or entirely new sequences which must be obtained by synthesis. Intermediates can be used to factor out reusable subsequences derived from series of DNAPL operations, enabling output to be constructed in a more parsimonious and understandable manner.

Outputs are target sequences which the library designer needs to have manufactured by DNA synthesis. Expressions of output definitions may refer to inputs, intermediates or other outputs.

Listing 6.1 gives an example of a basic DNAPL library where two different intermediate sequences are inserted between the same left and right subsequences of a longer input sequence to create two outputs sequences. Figure 6.2 shows a visualisation of the library in listing 6.1.

```

1 INPUT
  pte := GAATTCATCACCAACAGCGGCGATCGGATCAATACCGTGCGGGTCCTATCACAATCACC...;
END

5 # intermediates
  loop1 := GATCTCTCAGATTATTGTTGCACAATCGACTGGGGAAGTGC AAAATATAA...;
  loop2 := GGCGCATGACGCGTCGTGTTTCATCGACTACTTCCCCAGCA...;

OUTPUT
10 T1_SsoPox := pte[1:808].loop1.pte[884:1023];
    T2_AhlA  := pte[1:808].loop2.pte[884:1023];
END

```

Listing 6.1: Example of a simple DNAPL library where two different intermediate sequences (loop1 and loop2) are inserted between the same left and right subsequences of a longer input sequence (pte) to create the outputs T1_SsoPox and T2_AhlA. Example taken from <http://www.cadmad.eu/general-explanation>.

The FASTA equivalent of the output sequences of the DNAPL library in listing 6.1 and visualised in figure 6.2 is given in listing 6.2. What this simple example demonstrates is that the textual representation of a DNA library using a programming language like DNAPL, where the operational semantics of the language *are* the set of target sequences, communicates the intentions behind the library more clearly, succinctly and is more malleable than the sequences alone.

```

1 >T1_SsoPoxparsimonious
  GAATTCATCACCAACAGCGGCGATCGGATCAATACCGTGCGGGTCCTATCACAATCTCTGAAGCGGGTT
  TCACACTGACTCAGCAGCATCTGCGGCAGCTCGGCAGGATTCTTGCGTGCTTGGCCGGAGTTCTT...
  ...TGAGAGTGATCCCATTCCTACGAGAGAAGGGCGTCCACAGGAAACGCTGGCAGGCATCACTGTGAC
5 TAACCCGGCGCGGTTCTTGTCACCGACCTTGCGGGCGTCATGACTGCAG

```

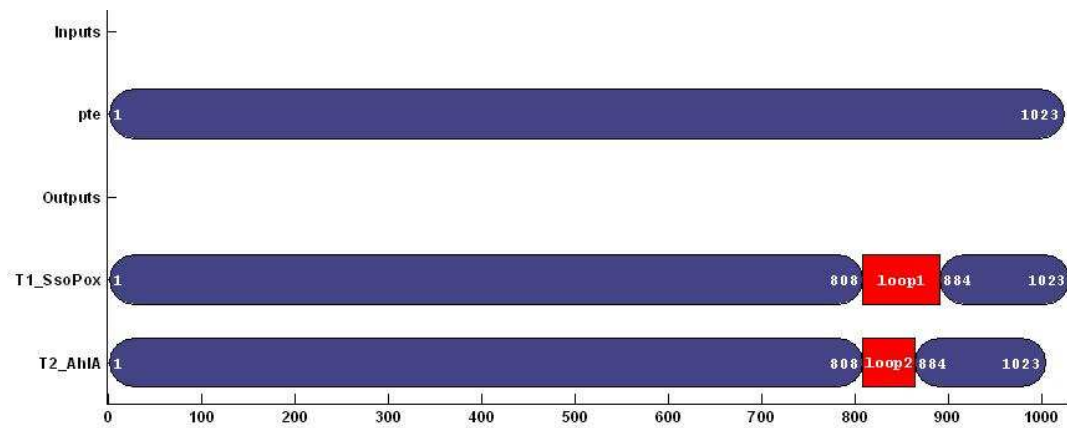


Figure 6.2: Visualisation of basic example DNAPL library in listing 6.1 showing fragments of the input sequence in blue (with start and stop indices to the left and right respectively) and whole inserted fragments in red. The length of the sequences is proportional to the number of nucleotides, measured on the x-axis.

```

>T2_AhlA
GAATTCATCACCAACAGCGGCGATCGGATCAATACCGTGCGGGTCCTATCACAATCTCTGAAGCGGGTT
TCACACTGACTCAGAGCACATCTGCGGCAGCTCGGCAGGATTCTTGCGTGCTTGGCCGAGTTCTT...
...CCCATTCCTACGAGAGAAGGGCGTCCACAGGAAACGCTGGCAGGCATCACTGTGACTAACCCGGCG
10 CGGTTCTGTACCGACCTTGCGGGCGTCATGACTGCAG

```

Listing 6.2: FASTA file of the output sequences in the DNAPL library example.

DNAPL syntax and semantics

The input and output sections are each delineated by the `INPUT` or `OUTPUT` keywords respectively and the `END` keyword (keywords and symbolic names are case-insensitive). Intermediates are any definitions that fall between the end of the input section and beginning of the output section.

The primary innovation of DNAPL was simply to replace long, cumbersome DNA sequences with symbolic names by which they can be referred to. Symbolic names are associated with the results of an expression using the definition operator `:=`. Most expressions return DNA sequences. Unquoted DNA sequences are valid expressions which simply return that sequence. For example the sequence expression `CTCGAG` is assigned to the name `XhoI` by: `XhoI := CTCGAG`; A semi-colon is required to terminate the definition. For clarity in what follows, only the expression on the right hand side of the definition is shown and without a terminating semi-colon.

Subsequences can then be obtained using symbolic names, the subsequence operator and an integer index or range (colon-separated start and stop indices: `[index]` or `[start:stop]` respectively (otherwise known as slicing). Indices are 1-based and stop indices are inclusive. For example, `XhoI[2]` returns `T` the second nucleotide of the sequence referred to by `XhoI`, and `XhoI[2:5]` returns `TCGA` the subsequence from the second to the fifth letter. The last

index symbol `end` can also be used as a stop index to obtain everything from the start index up to the 3' end: `XhoI[2:end]` returns `TCGAG`.

The concatenation operator `.` (dot) joins two sequences returning operations, e.g.: `C.XhoI[5:6]` returns `CAG`. Most expressions will consist of a series of concatenations.

Mutations can be made by using the assignment operator `=` (equals sign) in conjunction with the subsequence operator. Given `XhoI := CTCGAG,XhoI[1=A]` returns `ATCGAG`. Multiple mutations are separated by a comma so that `XhoI[1=A, 3:4=GT]` returns `ATGTAG`.

Sequences can be repeated using the repetition operator `*` (asterisk). `XhoI*2` returns `CTCGAGCTCGAG`.

Variable length repeats of the form `expr * (from:to)` imply a single sequence but give the manufacturer some leeway to produce one that is most feasible given biological and manufacturing constraints around repetitive sequences. From reading the documentation it is not clear what impact this has indexing downstream of the repetition.

Sometimes it is more appropriate to work with sequences of amino acids rather than nucleotides.

To do this DNAPL provides the amino acid function `a_a` which implies back-translation from amino acids to nucleotides given a codon table. For example `gene := a_a(VVPSTQPVTTPPATTPVTTPPTIPPS)`.

The choice of which back-translation to synthesise is at the discretion of the manufacturer, but the designer can have some say by defining a *codon table* which fewer than 64 codons. Codon tables take the form shown in listing 6.3. Only full amino acid names are used so as to avoid clashes between nucleotide and amino acid single letter codes (although single letter codes are used in the `a_a` function to refer to the amino acids). Codon alternatives are separated with the pipe `|` and terminated with a semi-colon. `Start` must be defined in addition to methionine in order to account for the use of alternative start codons in bacteria. Only one codon table is permitted by DNAPL file since they do not have names to distinguish between them. In libraries with a codon table it should appear before the input section.

```

1 CODON TABLE
   Alanine      := GCG | GCC | GCA | GCT ;
   Arginine     := CGC | CGT | CGG | CGA | AGA | AGG ;
   Asparagine   := AAC | AAT ;
5   Aspartic acid := GAT | GAC ;
   Cysteine     := TGC | TGT ;
   Glutamic acid := GAA | GAG ;
   Glutamine    := CAG | CAA ;
   ...
10  Serine      := AGC | TCG | AGT | TCC | TCT | TCA ;
   Stop        := TGA | TAA ;
   Threonine    := ACC | ACG | ACT | ACA ;
   Tryptophan   := TGG ;
   Tyrosine     := TAT | TAC ;
15  Valine      := GTG | GTT | GTC | GTA ;
   Start       := ATG ;
END

```

Listing 6.3: Example DNAPL codon table adapted from E. coli K12 http://openwetware.org/wiki/Escherichia_coli/Codon_usage with 63 codons (the low frequency TAG codon has been excluded).

Finally, DNAPL has three combinatorial operators used to define *degrees of freedom* in the DNA library. The `+` and `++` operators return the set of sequences that is the union of there operands, with the designation that each member be produced in a separate well (of a 96-well plate in which sequences are delivered) or the same well respectively. The pipe `|` operator denotes a choice between its operands which, as with codons, allows the manufacturer to decide which of the set of alternatives to produce.

Visualisations of DNAPL libraries from on the now defunct DNAPL website are reproduced in figure 6.3. They give a flavour of the scale and complexity of the DNA libraries that could be expressed with DNAPL.

Observations

Some DNAPL operations documentation, e.g. variable length repetitions, appear to have been intentions without implementations. The code for the parser and interpreter were unfortunately unavailable to clarify these issues. Without an existing implementation on which to built DNALD we were making a fresh start and took that opportunity to rethink some of the design choices made by DNAPL.

From the above description of the syntax and semantics of DNAPL we can make several observations about the nature of the language in general and in comparison to more orthodox programming languages like Java or Python, and establish the DNALD's positions on these:

- Keywords and symbolic names are case-insensitive. This is reasonable as the intended audience are biologists who may not be used to the case-sensitive restrictions of most programming languages and therefore enforcing case-sensitivity, especially of keywords, might seem pedantic. Additionally, by allowing case-insensitive symbolic names, names must be unique words and will therefore be easier to discriminate and unambiguous in conversation. **DNALD will be a case-insensitive language.**
- Sequence strings are not quoted. This is highly unusual and potentially problematic for parsing. For instance, `TAG` could be a sequence but it is also a valid symbolic name. **DNALD will require all sequences to be quoted.**
- Definitions are terminated by a semi-colon. This is unnecessary as what must follow an expression is either another definition (`symbolic_name :=`) or the section terminator (`END`). **Semi-colons for terminating definitions will initially be optional in DNALD** to be inclusive of users who are experienced with languages which require semi-colons and might miss them. In the future semi-colons may be repurposed, for example, to separate sets of arguments.

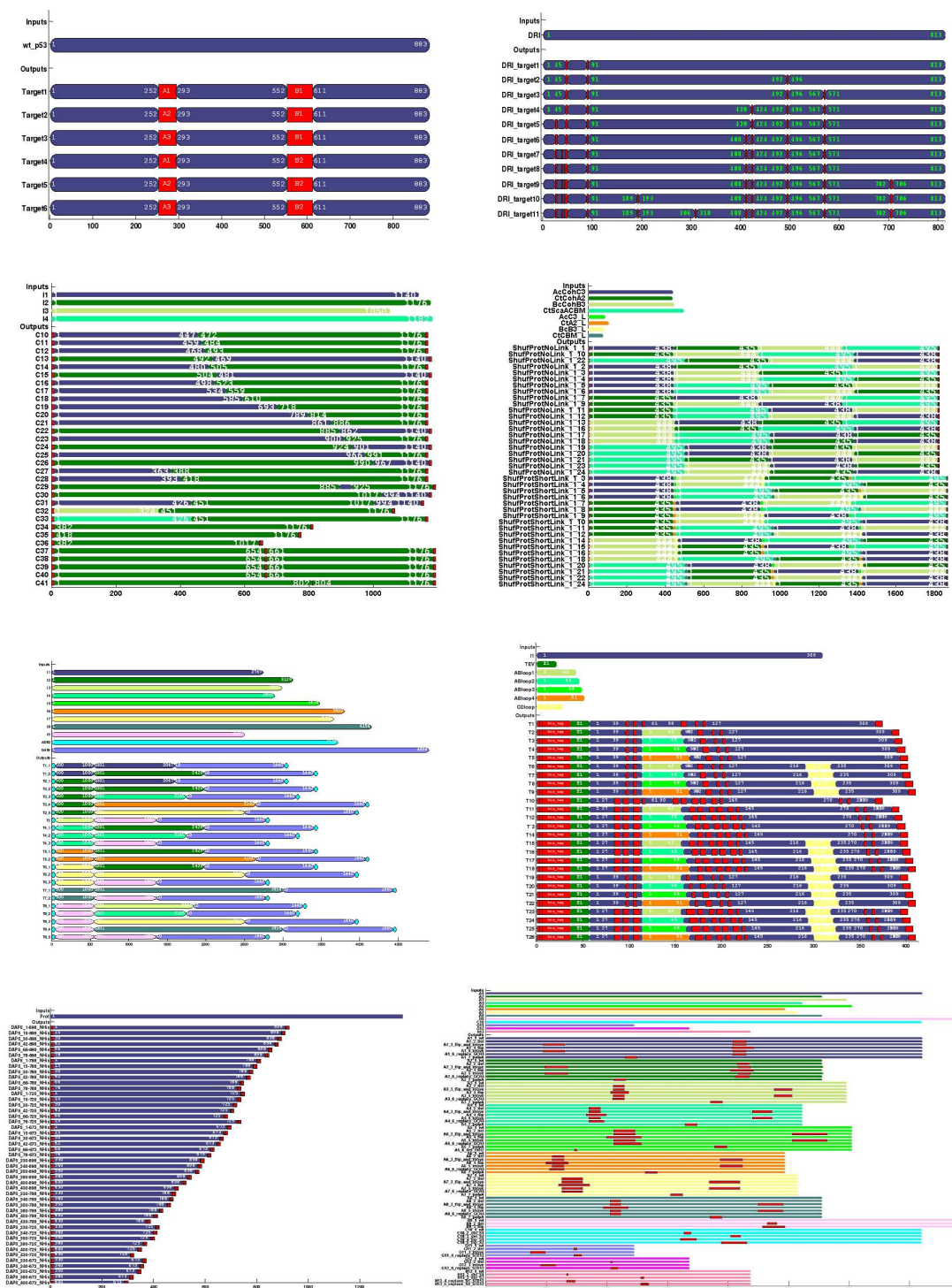


Figure 6.3: Set of example DNAPL library visualisations representative of the size and complexity of DNA libraries produced through DNA reuse. Each bar is a DNA sequence. Each colour represents a different input sequence (topmost sequences) to communicate how these are recombined in the library outputs.

- Subsequence indices start from 1 and go up to the length of the sequence inclusive, whereas array indices in programming languages are usually zero-based and exclusive. This is a sensible departure from Java as DNA sequences are also indexed starting from 1 and therefore the final index must be inclusive. **DNALD will retain 1-based, inclusive indexing of sequences consistent with biological usage.** Library designers more familiar with languages that use zero-based indexing must be careful not to confuse the two systems (although this is a simple error to validate and provide instant feedback on in the textual editor GUI).
- The symbol `end` can be used as a stop index in a subsequence expression to obtain everything up to the 3' end. This is a useful convenience as the length of a sequence returned from several operations may be difficult to determine. Java has no such syntax while Python allows slicing (aka subsequence expression) without a stop index to achieve the same end, e.g. `sequence[3:]`. **DNALD will use the end symbol as it is more explicit** and therefore better conveying the intentions of the library designer to readers who may be unfamiliar with the new language.
- The downstream processing of the sets of sequences defined by amino acids or combinatorial operations is unclear from the explanation of DNAPL given at <http://www.cadmad.eu/general-explanation>. That a symbolic name may refer to a set of sequences would seem to imply therefore that all operations must operate on and return sets of sequences, i.e. map each member of the operand set to a member of the return set by applying the same operation to each input. In which case a concatenation operation would produce the cross-product of the input sets. **All DNALD expressions return a set of sequences and all DNALD operations should accept a set of sequences.**

6.2.2 Specification

File structure

Reflecting the intention to manufacture, a DNALD file is structured similarly to DNAPL, with some additional features. A DNALD file is comprised of *an optional set of named codon tables* and *a set of named libraries*.

- Named libraries
 - Inputs are unambiguous DNA sequences that can be obtained from the library consumer (maybe as output of a previous library). Entirely synthetic libraries will not have inputs and therefore the inputs section is optional.
 - Outputs are always required therefore the outputs section is also strictly required.

```

1 Codon Table DefaultCodonTable {
/* A */ Ala := 'GCA' 0.27 | 'GCC' 0.26 | 'GCG' 0.25 | 'GCT' //0.22
/* R */ Arg := 'CGT' 0.30 | 'CGC' 0.26 | 'CGG' 0.15 | 'AGA' 0.13 | 'CGA'
0.09 | 'AGG' //0.07
/* N */ Asn := 'AAT' 0.59 | 'AAC' //0.41
5 /* D */ Asp := 'GAT' 0.65 | 'GAC' //0.35
/* C */ Cys := 'TGT' 0.52 | 'TGC' //0.48
/* E */ Glu := 'GAA' 0.64 | 'GAG' //0.36
/* Q */ Gln := 'CAG' 0.65 | 'CAA' //0.35
/* G */ Gly := 'GGT' 0.34 | 'GGC' 0.29 | 'GGA' 0.19 | 'GGG' //0.18
10 /* H */ His := 'CAT' 0.63 | 'CAC' 0.37
/* I */ Ile := 'ATT' 0.47 | 'ATC' 0.31 | 'ATA' 0.21
/* L */ Leu := 'CTG' 0.38 | 'TTA' 0.18 | 'CTT' 0.15 | 'TTG' 0.13 | 'CTC'
0.10 | 'CTA' 0.06
/* K */ Lys := 'AAA' 0.71 | 'AAG' //0.29
/* M */ Met := 'ATG' //1.00
15 /* F */ Phe := 'TTT' 0.64 | 'TTC' //0.36
/* P */ Pro := 'CCG' 0.37 | 'CCT' 0.24 | 'CCA' 0.23 | 'CCC' //0.16
/* S */ Ser := 'AGC' 0.20 | 'AGT' 0.18 | 'TCA' 0.18 | 'TCT' 0.18 | 'TCC'
0.14 | 'TCG' //0.11
/* * */ Ter := 'TAA' 0.58 | 'TGA' 0.33 | 'TAG' //0.09
/* T */ Thr := 'ACC' 0.31 | 'ACA' 0.25 | 'ACG' 0.22 | 'ACG' //0.22
20 /* W */ Trp := 'TGG' //1.00
/* Y */ Tyr := 'TAT' 0.65 | 'TAC' //0.35
/* V */ Val := 'GTT' 0.32 | 'GTG' 0.29 | 'GTA' 0.19 | 'GTC' //0.19
}

```

Listing 6.4: DNALD's default codon table targets *E.coli* K12, the most widely used model organism for genetic manipulation.

- Intermediate definitions are any definition not in the input or output sections. Intermediates are useful for sequence sets that are not manufacturing targets but can simplify the expressions of targets. They can provide valuable information about the relationships between targets that share them, ideally that two or more targets are derived from them but cannot be more simply derived from each other.

– Named codon tables

- Codons can have optional usage values to allow for optimisation of back-translation amino acid sequences in a corresponding target organism.
- A default codon table based on *E. coli* K12 will be used for back-translation when no codon table is specified.

Default codon table

DNALD's default codon table comes from *E.coli* K12 and was adapted from the Codon Usage Database (<http://www.kazusa.or.jp/codon/>). It is based on 8087 coding sequences, with a sum total of 2,330,943 codons. *E. coli* was used as it is the most prominent model organism in molecular biology, used extensively in the development of novel genetic constructs.

Listing 6.4 shows the default codon table. Generally, each amino acid (represented by its three letter code) is defined as a set of (optionally weighted) codon alternatives. Here the codons

are weighted as probabilities (fractions of 1). The weighting of the codon with the lowest probability in each set (here the rightmost, commented out) need not be specified as it can be computed as 1.0 - the sum of the others. Weights exceeding of 1 or summing to less than one are also permitted, allowing ratios or frequencies to be expressed, from which probabilities can be calculated. When the sum of the weights exceeds 1 then all weights must be specified as it is not possible to compute a remainder.

The single letter amino acid codes that would be used in an amino acid sequence are commented out inline on the left. Three letter codes are preferred in codon table definitions because the single letter codes would clutter the namespace (also '*' is reserved for the repetition operator) and their meaning is clearer. The three letter codes are reserved, it is an error not to specify all twenty and Ter, and available as literals in expressions corresponding to a (weighted) choice of codons.

Expressions

Expression in the DNALD language are a combination of explicit sequences, operators and functions (e.g. reverse, complement) that evaluate to a set of zero or more sequences.

- Individual sequences are evaluated to a set with that sequence as the lone member, except the empty sequence which evaluates to the empty set.
- All DNALD operators operate on sets of sequences produced from the evaluated expressions of their operands.
- Functions (e.g. reverse, complement) can take zero or more sequence sets as arguments and must return a sequence set.

Operations and rules of precedence

(operation) (example) in descending order of precedence

parenthesised expressions ('aa' + 'ac')

sequence 'acgt'

reference x

function_call reverse(x)

subsequence x[1:3]

mutation x[2:3='TA']

repetition $x * 2$

choice $x | y$

symmetric difference $x \wedge y$

difference $x - y$

intersection $x \& y$

union $x + y$

concatenation $x y$

Concatenation has the lowest precedence but is the most important operation in DNALD. This is emphasised by the lack of an operator: any whitespace separating two expressions will join them, but these rules precedence will ensure that they are both evaluated first.

Associativity

Operations are either unary or binary, operating on one or two expressions respectively (or one expression and an integer in the case of the repetition operation).

1. Unary operations are either function calls or intra-sequence operations. Unary operations operate in a left-associative manner so that the result of the leftmost operation is the input to the next leftmost and so on.
2. Binary operations are two expressions either side of an operator character or keyword. Binary operations can be chained together and operate in a right-associative manner so that the result of the rightmost operation becomes the right operand in the next rightmost and so on.

Definitions

Definitions assign names to the result of evaluating an expression:

$x := 'ATG'$ makes x a proxy for the sequence value ATG.

$y := x$ makes y the value of x .

$y := 'CTG'$ changes y but does not x , whereas $x := 'GTG'$ changes x and y .

Syntax and semantics

Concatenation is simply a series of whitespace separated subexpressions (actually binary expressions that are be chained): `x := 'ATG' 'CC'` defines `x` as `'ATGCC'`.

As with DNAPL subsequences can be extracted with the `[]` operator and either an index `i` or a `start:stop` range where $1 \leq start \leq stop$ and $start \leq stop \leq end$. For example: `x[2]` is `'T'` and `x[2:3]` is `'TG'`.

The operand of the slice operator is the expression to its left, which will be evaluated before slicing, so any expression can be sliced, including slices. For example, copy-pasted sequences can be sliced in situ (as intermediates) without needing to define them previously: `'ATGCCA'[2:5]` is `'TGCC'`. Where sequences in the operand set are of different lengths the `end` keyword can, as in DNAPL, be used to indicate everything up to the length of each sequence in the set. Multiple indices are *not* permitted because concatenation of two fragments is more explicit and amenable to insertion later; this is a design decision to make the language as debuggable as possible, where there is the opportunity for cleverness/laziness it will be abused.

Mutations use a similar syntax to slices but replace the indicated range with a new sequence fragment: `'ATG'[1='T']` is `'TTG'`. Multiple mutations are separated by a comma: `'ATGCCA'[1='T', 3:4='AT']` returns `'TTATCA'`. Mutations can be longer or shorter than the specified range, enabling insertion and deletion. Because deletion and insertion change the length of the sequence, within the same mutation operator the original indices of the sequence will be honoured. As with subsequences multiple indices per mutation are not permitted, although the potential for abuse is smaller and could be perhaps be justified. At present it is not clear whether many mutations would be the same, if that is the case this decision is open to review pending real world examples that demonstrate its utility.

Sequences can be reversed or complemented with functions of the same name:

`reverse('ATG')` returns `'GTA'` and `complement('ATG')` returns `'TAC'`.

These functions can be used together to obtain the reverse complement:

`reverse(complement('ATG'))` returns `'CAT'`.

Both the combinatorial operations union (+) and choice (|) create sets of more than one sequence, except that only one sequence in in choice is required. `'TTA'+'TTG'` returns `'TTA'+'TTG'`.

Concatenating a set returns the cross-product of the operands:

`'ATG' ('TTA'+'TTG')` returns another set of two sequences

`'ATGTT'+'ATGTTG'`.

A concatenation involving two set of sequences of size three and two, e.g.:

`z := ('A'+'CC'+'GGG') 'ATG' ('TTA'+'TTG')`

returns a set of six sequences


```
'AATGTTA' + 'AATGTTG' + 'CCATGTTA' +
'CCATGTTG' + 'GGGATGTTA' + 'GGGATGTTG'.
```

All operations work on sets so that mutation of *z* affects all members, so that *z* [1='T'] returns the set:

```
'TATGTTA' + 'TATGTTG' + 'TCATGTTA' +
'TCATGTTG' + 'TGGATGTTA' + 'TGGATGTTG'.
```

Repetition with '*' works as expected: 'ATG'*2 returns 'ATGATG'.

Variable length repetitions, e.g. 'TTA'*(1:3) returns the set of alternatives

```
'TTA' | 'TTATTA' | 'TTATTATTA'.
```

Listing 6.5 summarises current DNALD operations as a DNA library:

```
1 library Examples {
  inputs {
    Sequence := 'acgt'
  }
5 # intermediates
  5'end := 'atg'
  3'end := 'uua'
  outputs {
    ReferenceWithAssertion := Sequence is 'acgt'
10 Concatenation := 5'end Sequence 'A' 3'end
    Subsequence := Sequence[1:3]
    Mutation := Sequence[1='c']
    Mutations := Sequence[1:2='tg', 3='c', 4='a']
    Repetitions := 2 * Sequence is Sequence * 2
15 VariableLengthRepetitions := "A" * (2:3) is "AA" + "AAA"

    BackTranslation := backtranslate(exampleCodonTable, 'MS*')
    CrossTranslation := backtranslate(DefaultCodonTable,
      translate(exampleCodonTable, 'ATG' 'TCT' 'TAA'))

20 Union := 'a' + 'b' is 'b' + 'a'
    Intersection := ('a' + 'b') & ('b' + 'c') is 'b'
    Difference := ('a' + 'b') - ('b' + 'c') is 'a'
    SymmetricDifference := ('a' + 'b') ^ ('b' + 'c') is 'a' + 'c'

25 Choice := 'a' | 'b' is 'b' | 'a'

    ^Reverse := reverse(Sequence) // '^' escapes keywords in names
    ^Complement := complement(Sequence)
    ReverseComplement := reverse(complement(Sequence)) is
      complement(reverse(Sequence))
30 }
}
```

Listing 6.5: Summary of current DNALD functionality as a DNALD library. The *is* keyword is used to make an assertion of equality between its operands, and when satisfied can be used to document the evaluated sequence set.

Evaluation

DNALD is a declarative language where each definition binds a name to the result of evaluating an expression, i.e. a set of sequences. Names must be unique, cannot be redefined and there are no evaluable operations that can change the value of that name. Consequently, each named set

of sequences is immutable. The only way to change the value of a name is to change its defining expression.

Changing the value of a definition implicitly changes the value of all the other definitions that make reference it by name, just as a changing the value or formula of a cell in a spreadsheet can trigger a wholesale update of the cells that reference it via its coordinates. A DNALD library can similarly be modelled as a directed graph describing the dependency relationship between definitions vertices, where an edge from definition A to definition B exists if A is referred to be B. This property makes DNALD a dataflow language also. If a path from definition A to definition C exists and the value of A changes then C must be re-evaluated.

A definition's expression should not reference itself either directly or indirectly (through any series of other definitions expressions), i.e. create a cycle in the graph. A definition that does so is invalid because the value of the definition is indeterminable due to infinite regression. Any referring definitions are also indeterminable, as are their referrers and so on. A DNALD library is therefore unevaluable if its definitions do not constitute a directed acyclic graph (DAG).

The definitions in a valid DNALD can be evaluated either lazily, evaluating unevaluated definitions as they are encountered, or in an order which guarantees that all referred definitions will have previously been evaluated: a topological sorting of the DAG. Section 8.1.5 details our implementation of the DNALD evaluation process.

This specification implies but does not specify how the relationships between sequences (i.e. which subsequences are shared between sequences and in which definitions they originated) is to be managed. This is a detail of the implementation not the language. It is possible to ignore the provenance of subsequence fragments and produce a DNALD evaluator that still computes the correct output sequences, but that would be of little value. Our solution to the problem of tracing provenance is described in detail in section 8.1.5.

6.3 A real combinatorial DNA library using DNALD

Azurin production in *Pseudomonas aeruginosa* is positively controlled by the RNA-binding protein RsmA. This control has been found not to be exerted upstream of the ATG start codon, and cannot be exerted downstream of the rho-independent transcriptional terminator [255]. It may be possible that RsmA, a known post-transcriptional regulator, enhances the stability of the *azurin* mRNA transcript by binding and blocking degradation, thus increasing the quantity of translated protein translated from it. RsmA usually binds to mRNA at stem-loop structures having the following sequence: (U/A) CANGGANG (A/U). To bind, AGGA or AGGGA have almost always been found on the single-stranded loops. The azurin ORF has three sites that could correspond to RsmA binding sites. Interestingly of the three AGGA sites found, two (the second

and third) are located in the loops of potential short stem-loop structures, as determined using the mFold web server [256].

To investigate the contributions of each potential AGGA stem-loop to azurin transcript stability, we first defined seven parts, three of which could be altered to remove the stem-loop structures with AGGA sequences, and then recombined these to form a library of variants to test whether any combination of these is responsible for positive regulation by RsmA.

The parts to be conserved are 1, 3, 5 and 7. The parts to be altered are 2, 4 and 6:

GGCCTGGACA**AGGA**T (2)

GGCGAGA**AGGA**CTCG (4)

AAGCTGA**AGGA**AGGC (6)

We designed the library of azurin variants with altered parts in 5 steps:

1. translated Parts 2, 4 and 6 to get amino acid sequences to back-translate: GLDKD, GEKDS and KLKEG respectively
2. back-translated each translation to obtain the set of coding sequences that encode the same amino acid sequence: 192 in each case
3. filtered out the coding sequences containing AGGA
4. filtered the remainder to remove stem-loops with stems of length 3 or greater and loops of length 3 or greater, resulting in 52, 41 and 70 plausible variants of Parts 2, 4 and 6 respectively, 149240 ($52 \times 41 \times 70$) potential combinations
5. Due to the practical limitations of screening such a large number of variants, we chose 4 alternatives for each part, with a mix of codons between them. Adding the wildtype parts yielded 125 ($5 \times 5 \times 5$) variants of *azurin*, including the wildtype as a control. The resulting library encoded variants with 0 (wt), 1, 2 or 3 alternative sequences.

Listing 6.6 contains the final azurin DNALD. The library contains 1 input (the azurin ORF), 11 intermediates and 1 output definition. The first intermediate is the 501bp azurin gene sliced from the ORF. The 7 parts are defined as contiguous **in-frame** slices of the gene. The altered parts are defined as sets (using the '+' operator) of 4 synthetic sequences, which were chosen from the set of back-translations without AGGA or significant stem-loops; we did not use the

```

1 library azurin {
  inputs {
    azurin_orf :=
      "ATGCTACGTAAACTCGCTGCGGTATCCCTGCTGTCCCTGCTCAGTGCGCCACTGCTGG..."
  }

5
  azurin := azurin_orf[1:501]
  Part_1 := azurin[1:258] # in frame slices so that no codons
  Part_2 := azurin[259:273] # in 5'3' Frame 1 are truncated
  Part_3 := azurin[274:327]
10 Part_4 := azurin[328:342]
  Part_5 := azurin[343:360]
  Part_6 := azurin[361:375]
  Part_7 := azurin[376:501]

15 Part_2_altered := "GGCCTGGACAAAGAC" + "GGATTAGACAAAGAC" +
  "GGCCTGGACAAAGAT" + "GGACTCGATAAAGAT" # 4/52 variants
  Part_4_altered := "GGCGAGAAAGACAGC" + "GGAGAGAAAGATAGC" +
  "GGGGAAAAAGACTCT" + "GGTGAGAAAGATAGT" # 4/41 variants
  Part_6_altered := "AAGCTGAAAGAGGGC" + "AAGTAAAAAGAGGGG" +
  "AAATTTAAAGAGGGT" + "AAGCTGAAAGAGGGA" # 4/70 variants, last
  contains AGGGA

20 outputs {
  azurin_alternatives := Part_1 (Part_2 + Part_2_altered) Part_3 (Part_4 +
  Part_4_altered) Part_5 (Part_6 + Part_6_altered) azurin_orf[502:end]
}

```

Listing 6.6: DNALD library investigating post-transcriptional regulation of azurin.

back-translation capabilities of the DNALD language directly as those back-translations needed to be filtered for stem-loops and this feature is not yet implemented in the language. Finally, the 125 variants are defined in one expression as the concatenation of each conserved part and the union of each altered wildtype part with its alternatives.

Figures 6.7, 6.4 and 6.5 present three different views on to the outputs computed from the azurin library: the sequences in FASTA format, the composition of the sequences according to the origin of the fragments that compose them, and the minimal Directed Acyclic Word Graph (DAWG) of the outputs with fragments as nodes.

Although the design of the azurin library is relatively simple, with the DAWG corresponding exactly to the lone output expression, it is our intention to use the computation of minimal DAWGs (from any user design or set of related nucleotide sequences) to inform the construction planning stage of library manufacture, specifically regarding the minimal number and order of concatenations that are required to construct the library.

Once the azurin library has been manufactured it will be processed by cloning each azurin variant into *P. aeruginosa* PAO1, characterization of Azurin levels and relating those to contributions by combinations of AGGA stem-loops present.

In this chapter we have introduced the DNALD language and specified its syntax and semantics. The next chapter introduces DNA Library Designer, our integrated development environment for DNALD.

```

1 |>Part_6_altered (3 of 4)
  AAGCTGAAAGAGGGC
  >Part_6_altered (4 of 4)
  AAGTTAAAAGAGGGG
5 |>azurin_alternatives (1 of 125)
  ATGCTACGTAAACTCGCTGCGGTATCCCTGCTGTCCCTGCTCAGTGCGCCACTGCTGGCTGCCGAGTGCTCGGTGGACAT
  CCAGGGTAACGACCAGATGCAGTTCAACACCAATGCCATCACCGTCGACAAGAGCTGCAAGCAGTTACACGTCAACCTGT
  CCCACCCCGGCAACCTGCCGAAGAACGTCATGGGCCCAACTGGGTACTGAGCACCGCCCGCCGACATGCAGGCGTGGTC
  ACCGACGGCATGGCTTCCGGACTCGATAAAGATTACCTGAAGCCCGACGACAGCCGTGTCATCGCCACACCAAGCTGAT
10 |CGGCTCGGGAGAGAAAGATAGCGTGACCTTCGACGTCTCCAAATTAAAAGAGGGTGAGCAGTACATGTTCTTCTGCACCT
  TCCCGGGCCACTCCGCGCTGATGAAGGGCACCTGACCCTGAAGTGATGCGCGAGCGATCCGCTGCATGAAAAAGCCCGG
  CCGCTGCCGGGCTTTTTCATG
  >azurin_alternatives (2 of 125)
  ATGCTACGTAAACTCGCTGCGGTATCCCTGCTGTCCCTGCTCAGTGCGCCACTGCTGGCTGCCGAGTGCTCGGTGGACAT
15 |CCAGGGTAACGACCAGATGCAGTTCAACACCAATGCCATCACCGTCGACAAGAGCTGCAAGCAGTTACACGTCAACCTGT
  CCCACCCCGGCAACCTGCCGAAGAACGTCATGGGCCCAACTGGGTACTGAGCACCGCCCGCCGACATGCAGGCGTGGTC
  ACCGACGGCATGGCTTCCGGACTCGATAAAGATTACCTGAAGCCCGACGACAGCCGTGTCATCGCCACACCAAGCTGAT
  CGGCTCGGGAGAGAAAGATAGCGTGACCTTCGACGTCTCCaaagctgaaagaggaGAGCAGTACATGTTCTTCTGCACCT
20 |TCCCGGGCCACTCCGCGCTGATGAAGGGCACCTGACCCTGAAGTGATGCGCGAGCGATCCGCTGCATGAAAAAGCCCGG
  CCGCTGCCGGGCTTTTTCATG

```

Listing 6.7: Extract from generated FASTA file of azurin library `azurin.fasta`. The final two intermediates of altered part 6 sequences are followed by the first and second outputs: `azurin_alternatives` (1 of 125) is the reconstituted wildtype, (2 of 125) contains one altered part shown in lowercase.

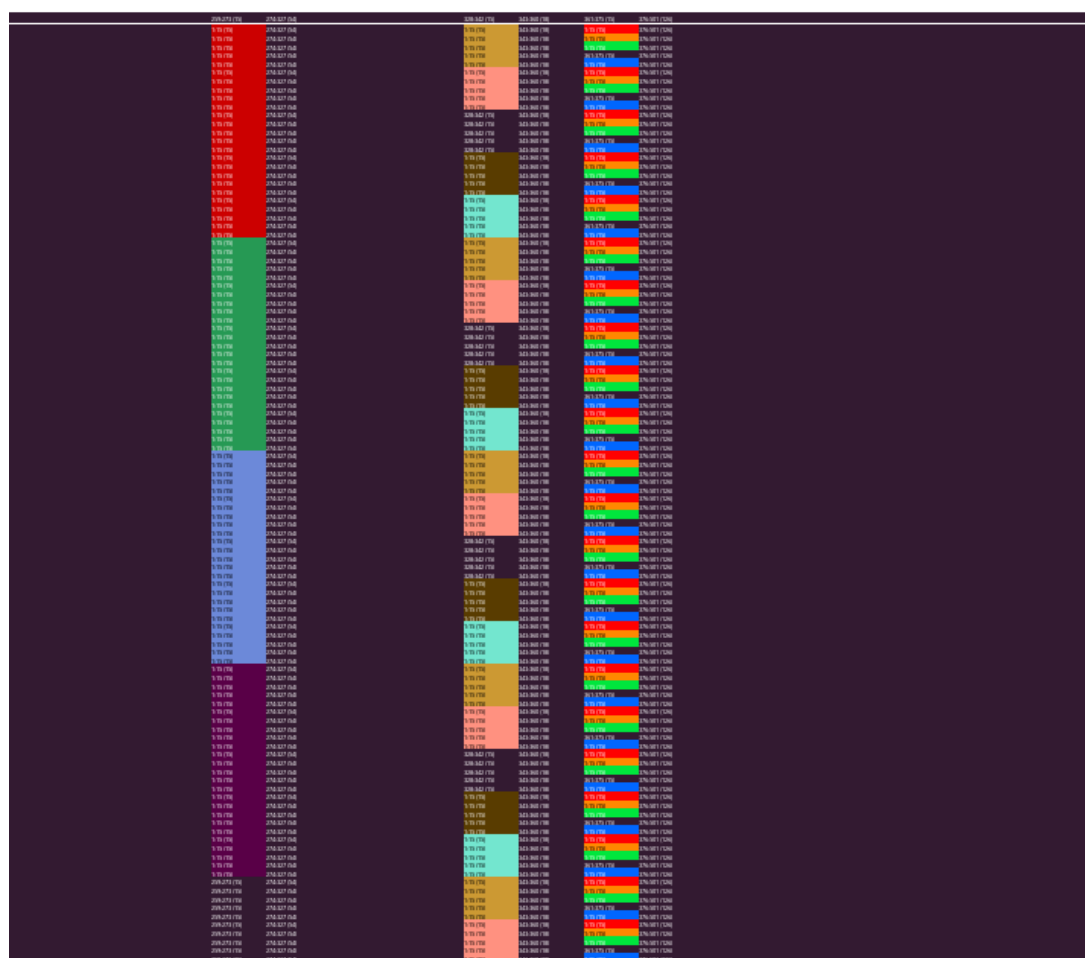


Figure 6.4: Zoomed in view of the output sequences computed from the azurin library. The reconstituted wildtype sequence is the long purely purple line at the top of the outputs stack. The variant parts 2, 4 and 6 are differentially coloured, and the 125 combinations of these can be seen in the odometer-like rotations of colours: part 6 changes most rapidly, then part 4 and then part 2 as the sequences descend.

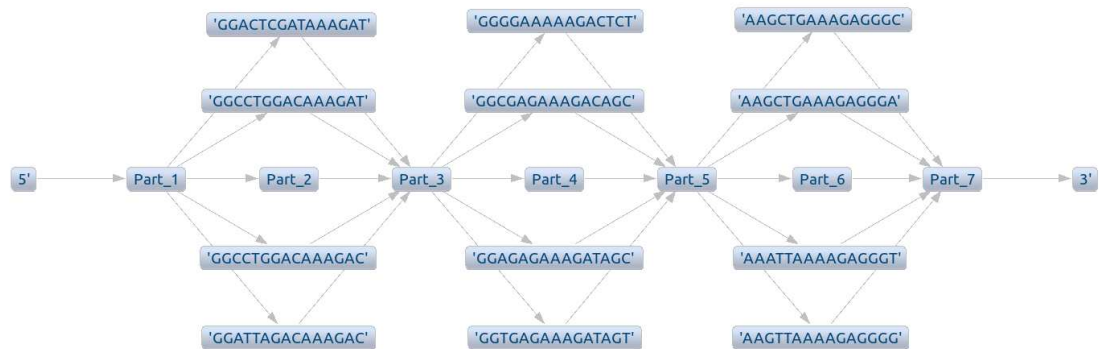


Figure 6.5: Directed Acyclic Word Graph (DAWG) of fragments constituting all outputs of the azurin library. Every path in the DAWG from 5' to 3' is the sequence composed of the nodes in the path. The central path is the reconstituted wildtype, the other paths are all unique output sequences missing either 1, 2 or 3 stem-loop forming and AGGA containing subsequences depending on the number of parts 2, 4 or 6 traversed in that path.

Chapter 7

DNA Library Designer

Chapter abstract

Chapter 6 introduced the DNALD language for combinatorial DNA library design. This chapter introduces the DNA Library Designer integrated development environment (IDE) for designing combinatorial libraries with DNALD. DNA Library Designer is the reference implementation of the DNALD language and evaluator.

7.1 An IDE for DNALD

There are several potential users of the DNALD language introduced in chapter 6:

1. Biologists requiring combinatorial DNA libraries to investigate a biological problem, for whom the language is tailored.
2. Manufacturers of DNA libraries that need to know where common subsequences of the output library originated in order to produce a construction plan that maximises DNA reuse.
3. Programs that produce or consume DNA library specifications as part of their operation to, for example, automate the iterative design and validation of synthetic biological devices.

We aim to cater for each of these different users in the most appropriate manner by providing multiple interfaces to the language:

- a graphical user interface for writing and debugging library designs that transparently handles evaluation
- a command line interface for shell scripts and web servers that evaluates DNALD files and exports a variety of alternative sequence formats
- an application programming interface (API) for reading, writing and evaluating DNALD files

As stated previously, the DNALD language is still evolving and its implementation is likewise. At this stage of development, when the APIs are still fluid, we have focused on supporting human interaction by developing DNA Library Designer.

DNA Library Designer is a sophisticated integrated development environment (IDE) for DNALD that enables biologists to design and explore combinatorial DNA sequence libraries with the support of a real programming editor. The parser and IDE are developed as a set of Eclipse plugins using the Xtext framework for domain-specific language (DSL) development. DNA Library Designer can be installed alongside other plugins in an existing Eclipse installation, and is also available as a standalone product for Linux, Mac and Windows.

7.2 Features

DNA Library Designer fully leverages Xtext and the Eclipse Rich Client Platform (RCP) to provide many of the standard features common to programming language editors: syntax/reference highlighting and validation, code completion, source navigation, outline views and rename refactoring, find-replace (with regular expressions) and a workspace model of project and file management with full text searching. We have added comprehensive validation of DNALD expressions and relevant quick fixes, templates and wizards for DNA libraries including examples. The outputs of evaluated DNALD files are shown in visualisation and several table-based views.

Editing

Figure 7.1 shows a typical arrangement of DNA Library Designer's workbench-style interface. The main window consists of the editor area (right), docked and open views such as the Project Explorer (left) and minimized view bars (bottom right), the *Perspectives* bar (top right) and standard menu, toolbars (top) and status bar (bottom). An *editor* displays the contents of a file and provides advanced editing facilities (discussed later). Views display information either derived from the contents of the active editor or pertaining to the file system or other plugins. Projects (top-level folders in The Project Explorer view) are located on disk in a *workspace*: a directory of the users choosing.

Figure 7.2 demonstrates how multiple editors and collapsible views are easily rearrangement to suit the current situation. Files that are open when the application is closed are reopened on launch. Currently opened (and minimized) views, can be saved as Perspectives and switched between.

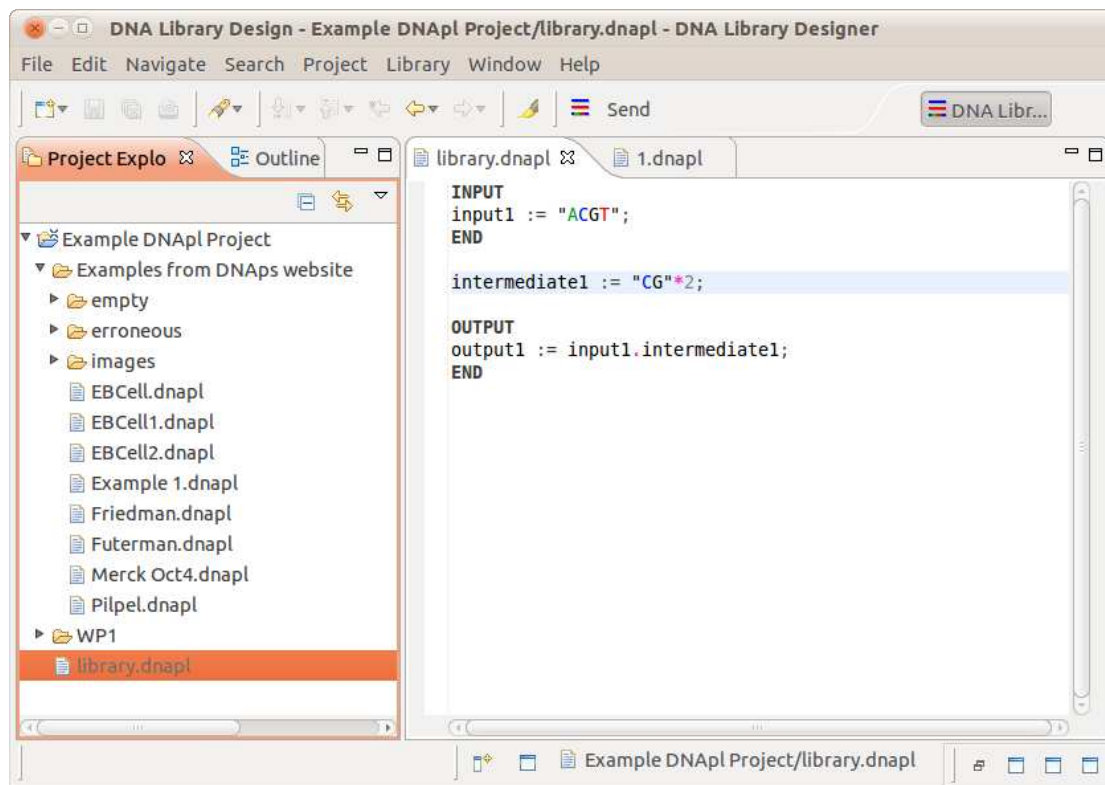
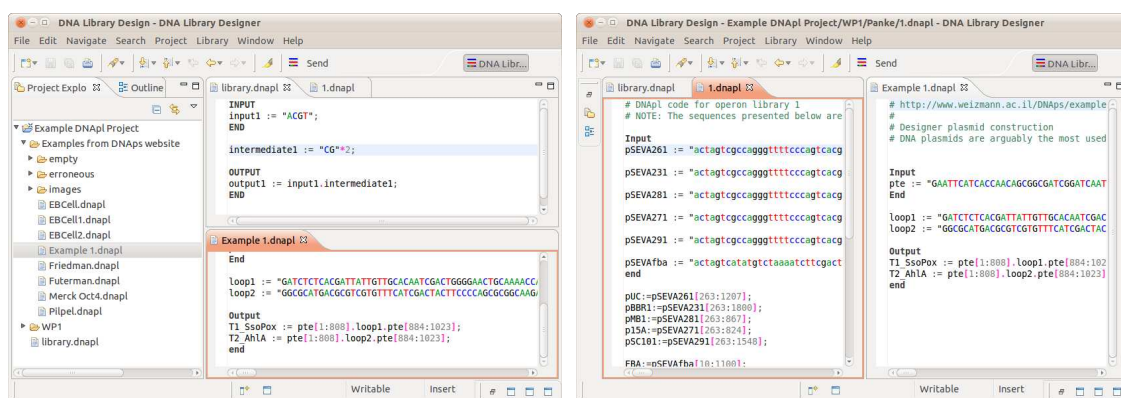


Figure 7.1: DNA Library Designer interface.



(a) Editor pane split vertically

(b) Editor pane split horizontally

Figure 7.2: Multiple editors and collapsible views are easily rearrangement to suit the current situation.

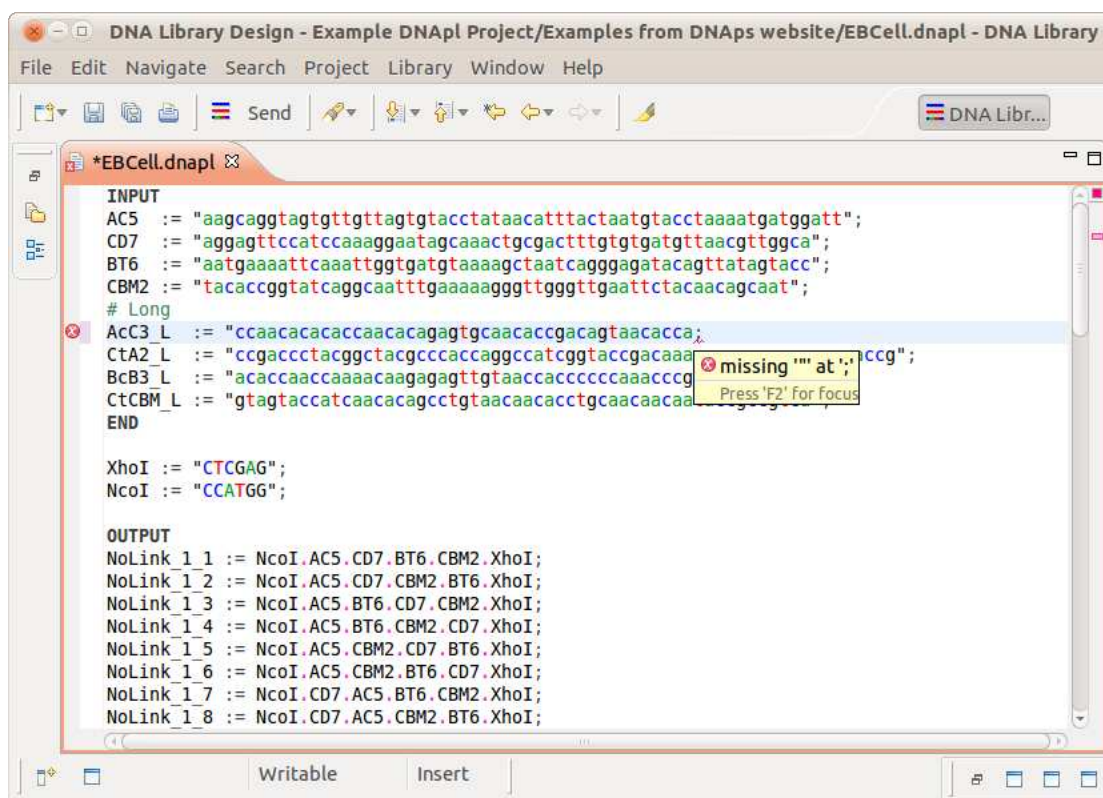


Figure 7.3: Syntactic validation and colouring.

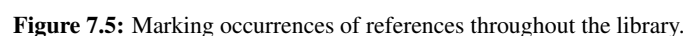
Syntax highlighting

Figure 7.3 shows how DNALD syntax is highlighted with colours. Syntax colouring is fully customisable via the Window > Preferences > DNALD > Syntax Coloring menu choices. By default definition names and references are black, operators are bright pink and nucleotide bases in sequences are coloured according the established convention (adenine is coloured green, cytosine blue, guanine black and thymine/uracil red).

Browsing and navigation

The Outline view shown in figure 7.4 displays a list of definitions in the active editor. Items are displayed in the order they are defined and can also be sorted alphabetically. Clicking an item in the Outline view scrolls to the definition name in the editor for quick navigation.

When the Mark Occurrences button is toggled on, positioning the text cursor on a definition reference highlights other references in the editor with a grey background and the definition with a beige background (figure 7.5), useful for identifying reuse in densely specified libraries.



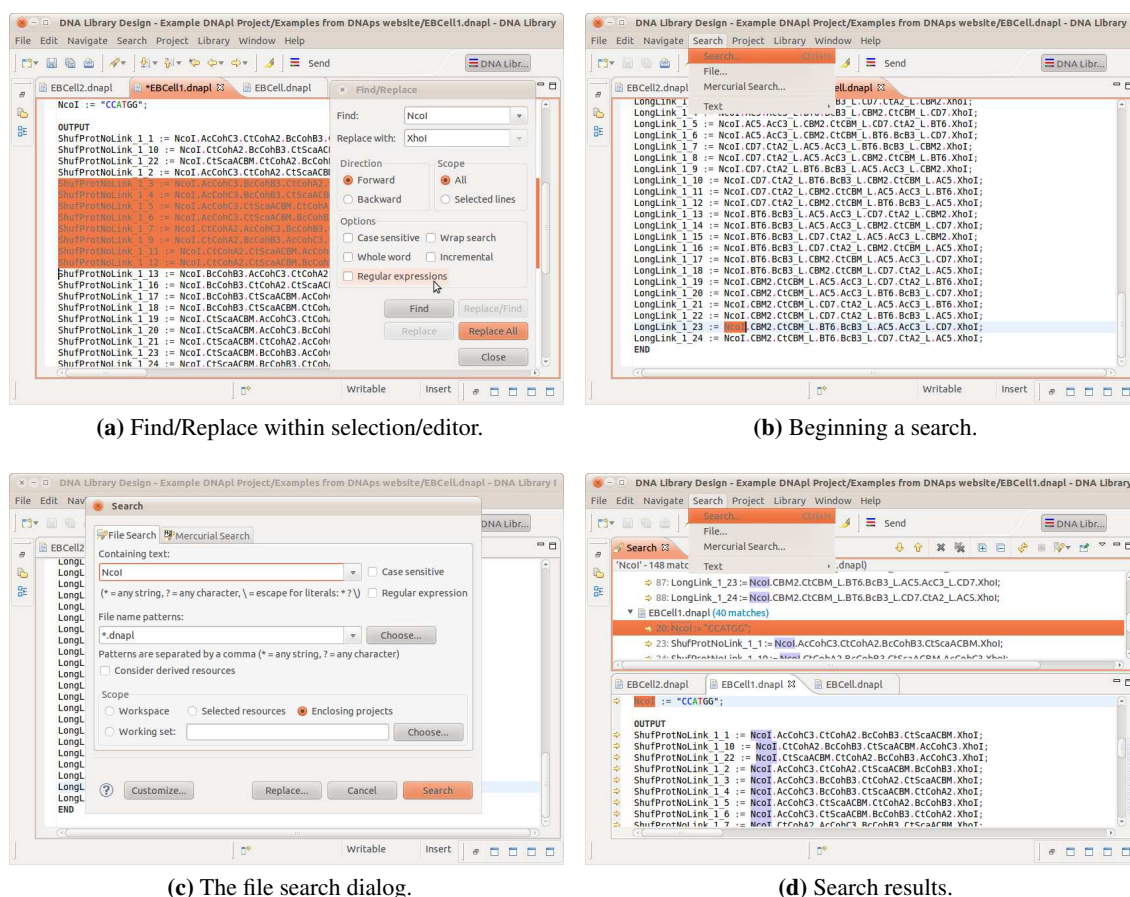


Figure 7.6: Finding and replacing text within and between files.

Searching

The IDE enables library designers to find, and replace, text within the current selection/editor or across files/folders/projects/workspace to locate reused references, sequences and operations; shown in figure 7.6. Searching can be case insensitive, utilise regular expressions to find partial or multi-line matches, and perform batch replacements on multiple files (declinable in each instance). Clicking a Search view result jumps to the relevant in the editor (opening the file in an editor if necessary), providing a quick means of navigating across the workspace.

Version control

Versioning of libraries and projects is deferred to other Eclipse plugins such as MercurialEclipse and EGit. The standalone DNA Library Designer product bundles MercurialEclipse, giving library designers a fully integrated distributed version control system (DVCS). We envisage DVCS as the primary means of communicating DNALD libraries within working groups and back and forth to the manufacturer as designs are refined in line with what is possible to produce.

Evaluation

All DNALD files in open projects are evaluated when DNA Library Designer is started and as when they are changed. evaluation occurs transparently in a background thread, leaving the user interface responsive. When files contain errors, erroneous definitions and their dependents are skipped while the remainder are evaluated. FASTA files are automatically generated in a subdirectory of the relevant project, ready for use in the bioinformatics pipeline or to send to manufacture. Finally, the Definitions, Sequences, Sequence Fragment and Library visualization views (section 7.2) are synchronised with the evaluated outputs of the libraries.

Validation

The DNALD editor reports three classes of errors with DNALD library designs. Syntactic errors are derived from the grammar such as missing quotation marks around sequences, unpaired parentheses, or a missing outputs section. Linking errors are created by misspelled or non-existent references. Semantic errors are discovered by the validator or evaluator and include cyclic references, invalid nucleotide characters, out-of-range subsequence indices, and overlapping mutation indices.

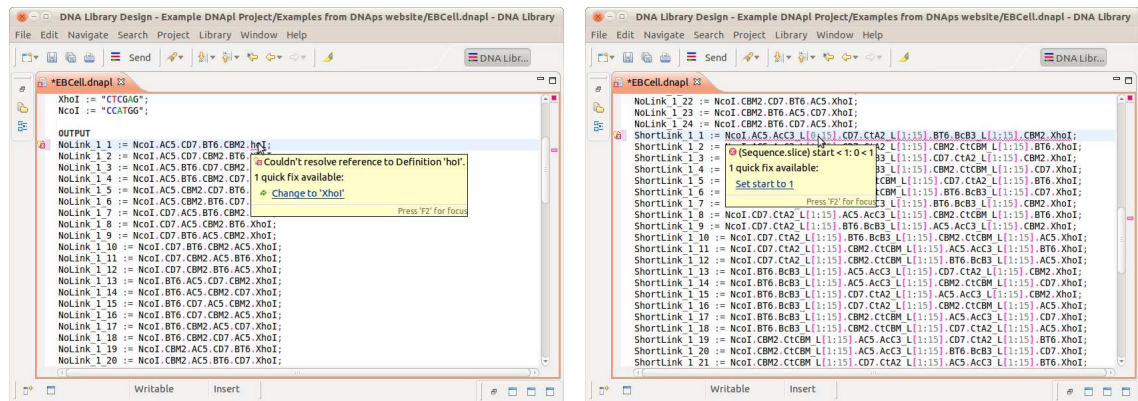
Errors are reported to the user by underlining the error causing expression with a red squiggle, and also by a red symbol on the left hand side of the DNALD file editor. Hovering over either the underline or symbol brings up a tooltip explaining the source of the error. Certain errors can be autocorrected with a quick fixes small user intervention (available only when possible and appropriate from the tooltip or by pressing Ctrl-1 on the corresponding line).

Figure 7.7 demonstrates reporting of validation errors and suggestion of quick fixes. Unresolvable references which are obviously misspelled offer the most likely correction from the pool of valid references (an Xtext feature). Out-of-range indices offer the minimum or maximum value dependent on other valid indices in the expression.

Comparison The compare editor, shown in figure 7.8, highlights the differences between two libraries (currently based on character differences rather than semantics). It provides actions for navigating and copying differences between the libraries, enabling a supervised merging process.

Evaluated library views

The Definitions view (figure 7.9) provides a tabular display of each definition, the expected number of sequences and a reserialization of the parsed expression which users can compare



(a) Unresolvable reference error and quick fix.

(b) Out-of-range index error and quick fix.

Figure 7.7: Validation errors and quick fixes.

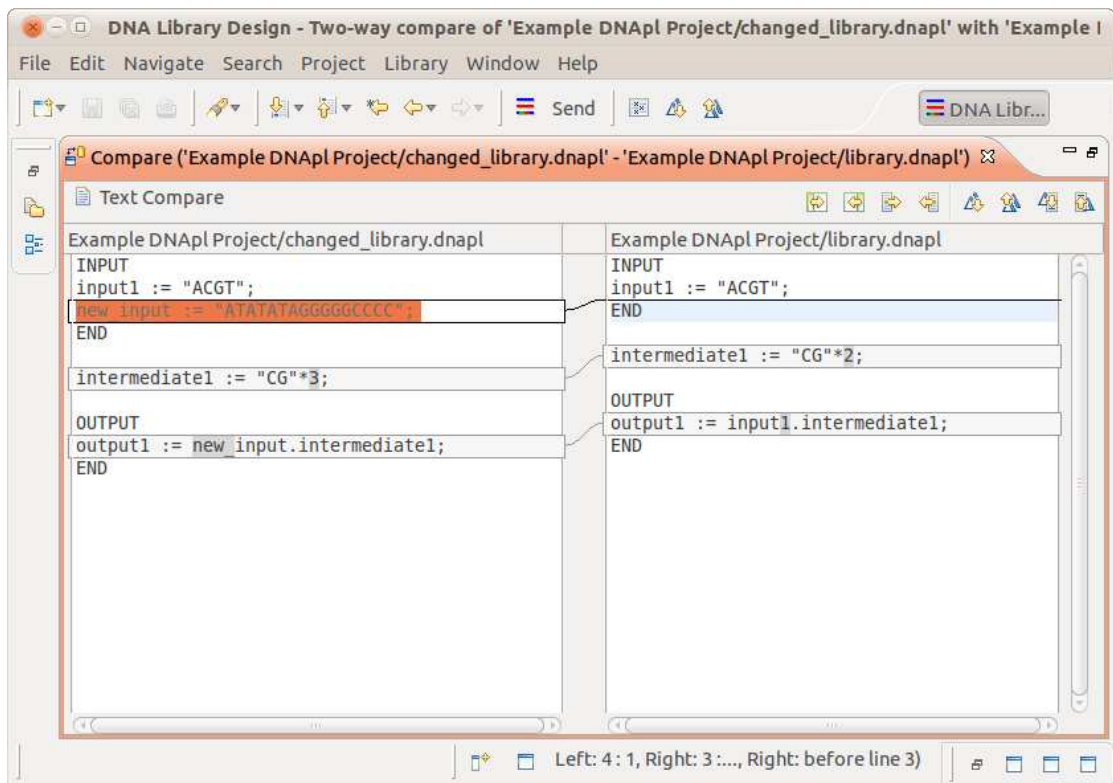


Figure 7.8: Comparing differences between libraries and versions.

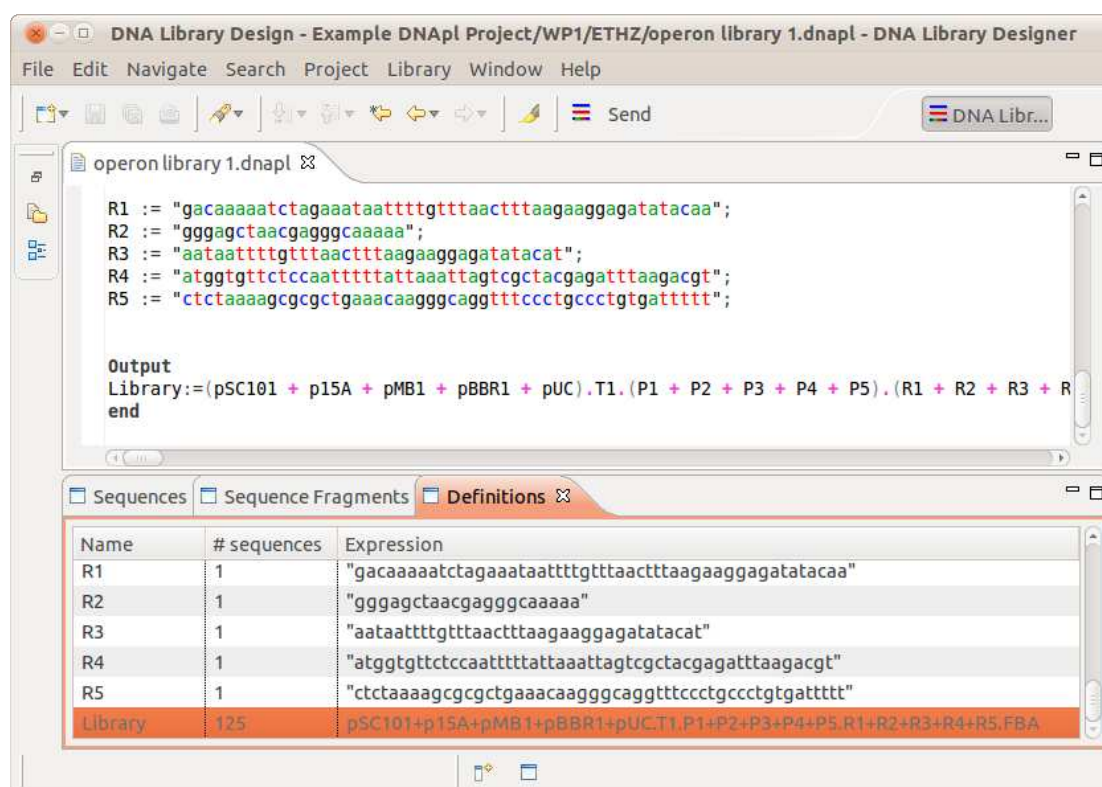


Figure 7.9: Definitions view.

with the input expression and gain confidence that the expression was understood correctly by the evaluator.

The Sequence Fragments view (figure 7.10) lists all original fragment of sequence, which are ultimately concatenated (in the manner of the Y operation) to create the designed sequences. Fragment tracking identifies the definition in which a fragment originates, the definition expressions and sequences in which it is reused, enabling visualization of the flow of sequence information through the library. Under the hood each original fragment of sequence is stored (interned) and sequences are composed of references to fragment slices rather than copied sequence data which saves memory, facilitates fast comparison and enables tracking of reuse within the library.

The Sequences view (figure 7.11) shows information about every sequence that the library defines, particularly useful when a definition such as Library (pictured) uses combinatorial operators which result in multiple sequences to be produced from a single definition. The name and number if applicable are shown along with the sequence type (inherited from the library section in which the sequences parent definition resides), the length of the sequence, its computed GC content and a concatenation expression that defines the sequence as a series of sequence fragment slices.

The Library view (figure 7.12) visualizes the computed sequences using a different colour for each sequence fragment to show their flow through the library. When zoomed out the nucleotide

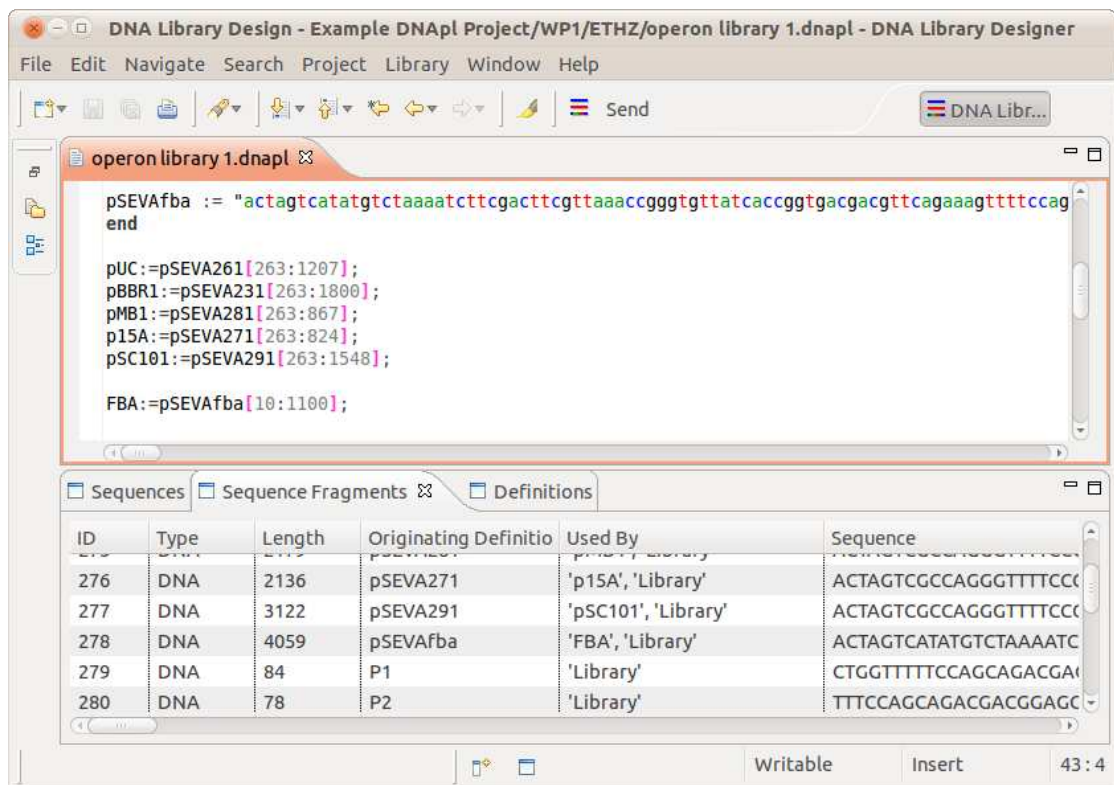


Figure 7.10: Sequence Fragments view.

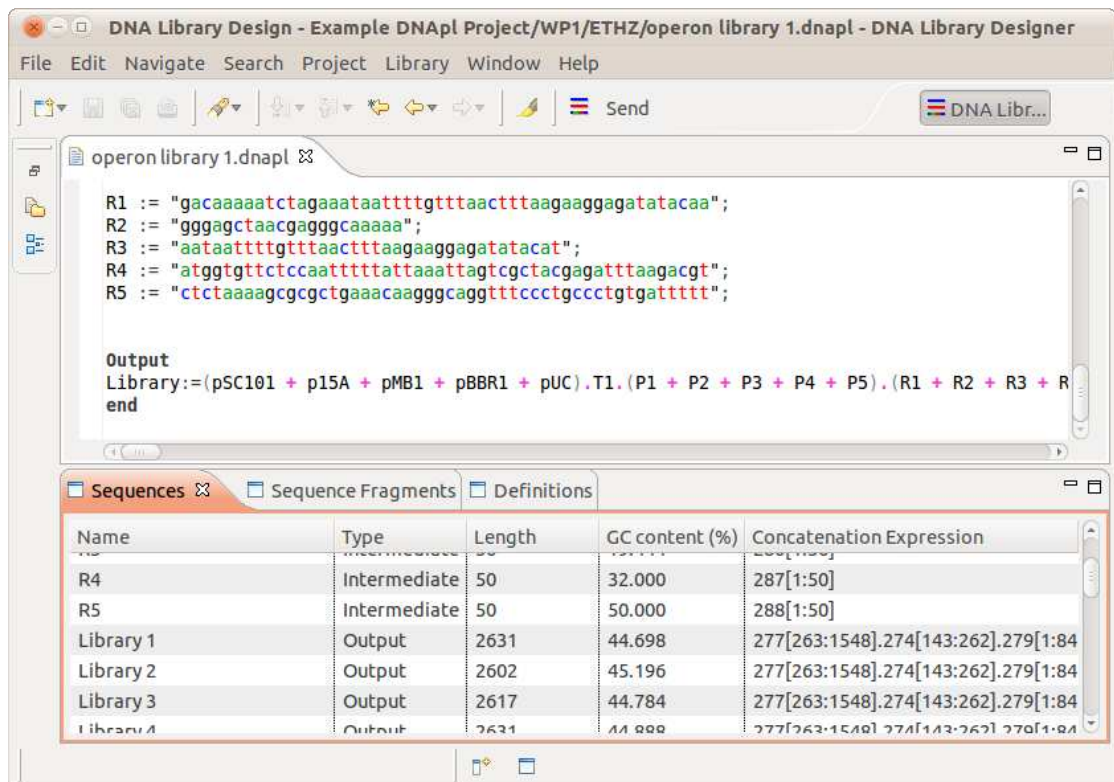


Figure 7.11: Sequences view.

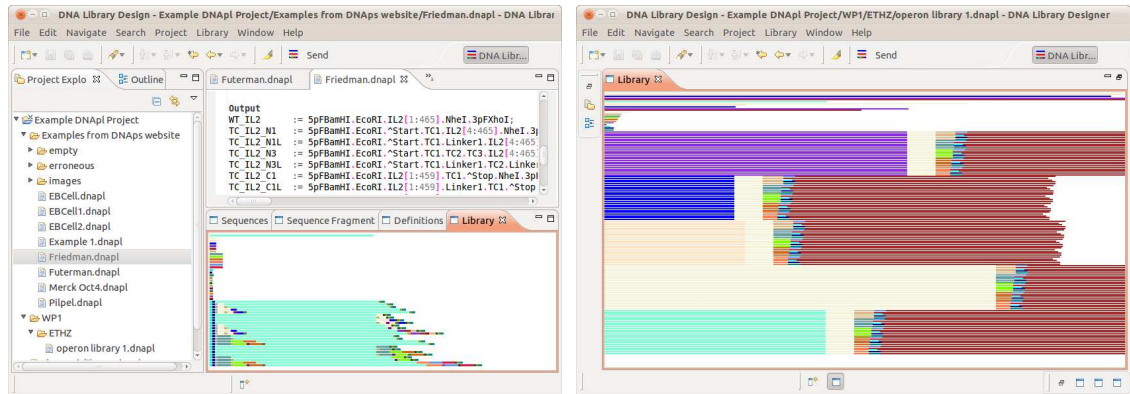
sequences are hidden, providing a purely compositional view of the outputs useful for quickly checking the order of fragments. When zoomed in close enough the nucleotides become visible, as in subfigure 7.12c, allowing the designer to confirm that the fragments are coloured correctly. This multifaceted visualisation is another important means of debugging libraries early on in their design phase. The combinatorial nature of the libraries, coupled to the generative nature of evaluating the library means that mistakes in definitions that are incorporated by many other definitions will be propagated through the library. Tracing the origin of the unexpected fragment will often lead directly to the problem. Fixing the problem at its source then propagates the fix automatically to every dependent sequence.

7.3 Conclusions

DNA Library Designer is a work in progress but its user friendly features - a real programming editor for a new language, and a robust validation/evaluation scheme - means that it can already be used for real-world projects.

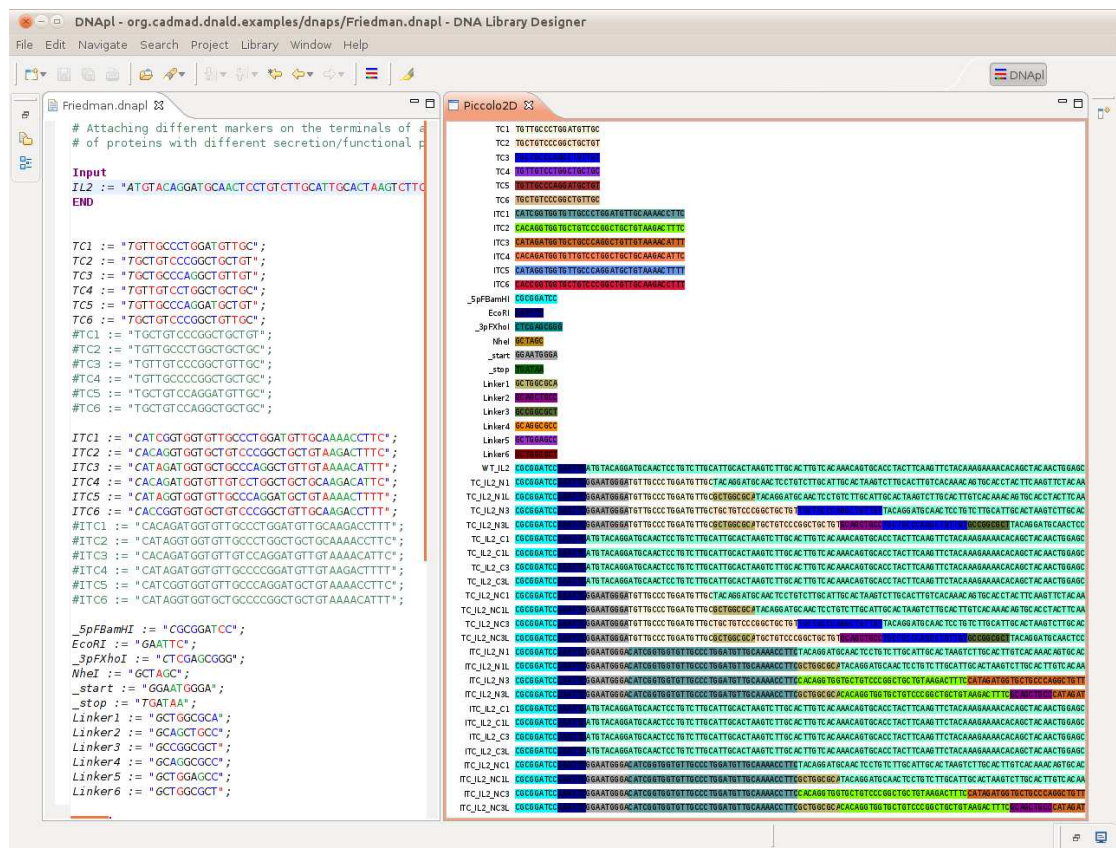
Regarding alternative workflows, as stated previously both human and non-human library consumers alike should be able to use our DNALD evaluator to compute the output sequences of a DNALD library, and export these in other formats (e.g. FASTA) as part of their investigation workflow. We are working on a command line interface to the evaluator which can be called manually, by shell scripts, or by programs.

Our datamodel for DNALD sequence sets, discussed in the next chapter, exposes a fluent Java API which allows programmers in Java and other JVM-based languages construct and manipulate sets of DNA sequences using DNALD operations. This fundamental design choice will expose the DNALD to a new set of power users who will hopefully want to improve the language and this software further.



(a) Embedded Library view

(b) Maximised Library view.



(c) Side-by-side textual specification and visualization of evaluated library.

Figure 7.12: Library visualisation view.

Chapter 8

Software engineering

Chapter abstract

This chapter is concerned with the design and implementation of the software presented in the previous chapters. An overview of the software stacks on which DNA Library Designer and the Infobiotics Workbench were developed is given with a attributions of their use. A selection of features and algorithms are used to illustrate the original code contribution of this thesis.

8.1 DNA Library Designer

8.1.1 Software stack

Only a few Eclipse plugins and Java libraries were used in the development of DNA Library Designer:

- | | |
|----------------|---|
| Eclipse | plugin framework, Workbench UI for RCP product and countless plugins providing project management, file comparison, etc. |
| Xtext | open-source framework for developing domain-specific languages. First version published in 2006 under the openArchitectureWare project and now on version 2.3.1 as an Eclipse project. Xtext handles the parser generation, editor UI and parsing of DNALD, as well as providing and running threaded validation, and generator skeletons using the Generation Gap Pattern ¹ . |
| Guice | dependency Injection framework from Google, used to configure Xtext but also to inject the <code>DNALDResourceScopeCache</code> singleton which connects the DNALD evaluator to the validator, various library views and the FASTA generator. |
| Guava | advanced Java Collections from Google with immutable types, static methods and fluent APIs. Highly recommended.
http://code.google.com/p/guava-libraries/ |

¹<http://www.drdobbs.com/architecture-and-design/231902091?pgno=3>

JGraphT generic graph data structures and topological sorting used by the evaluator.

Zest graph visualisation and layout algorithms.

Draw2D fast, component-based graphics framework, used to draw library visualisations.

The DNA Library Designer software stack is summarised in figure 8.1.

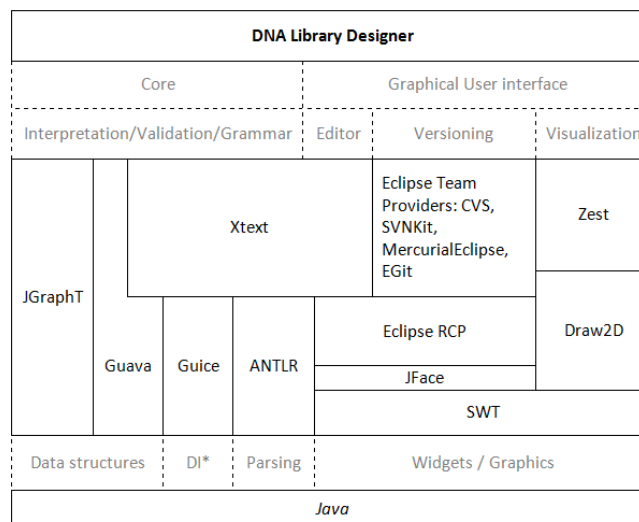


Figure 8.1: Dependencies of DNA Library Designer and its components. The diagram should be viewed as a stack where each layer above is dependent on functionality provided by the layer below.

8.1.2 DNALD grammar implementation

The structure and syntax of Xtext DSLs are specified as a grammar using an Xtext DSL implementing an extended Backus Naur Form (BNF) grammar. Xtext converts our grammar from its grammar into an ANTLR (ANother Tool for Language Recognition) grammar, and ANTLR generates a recursive-descent parser for DNALD. Figure 8.2 and those succeeding it visualise the rules of the DNALD grammar using the conventional railroad iconography. The name of the grammar rules are given to the left. Any path may be taken from left to right along the lines of railroad to achieve a successful parse. Attributes which delegate to other rules are highlighted in grey, keywords or symbols are shown in white boxes.

DNALD expression subgrammar

ANTLR generates LL(*) parsers which cannot handle left-recursion, e.g.: `Expression: Expression Operator Expression | Term`. DNALD expressions contain operators whose operands are expressions (section 6.2.2) which presented a problem: how to define the grammar without left-recursion? The solution was to left-factor the grammar according

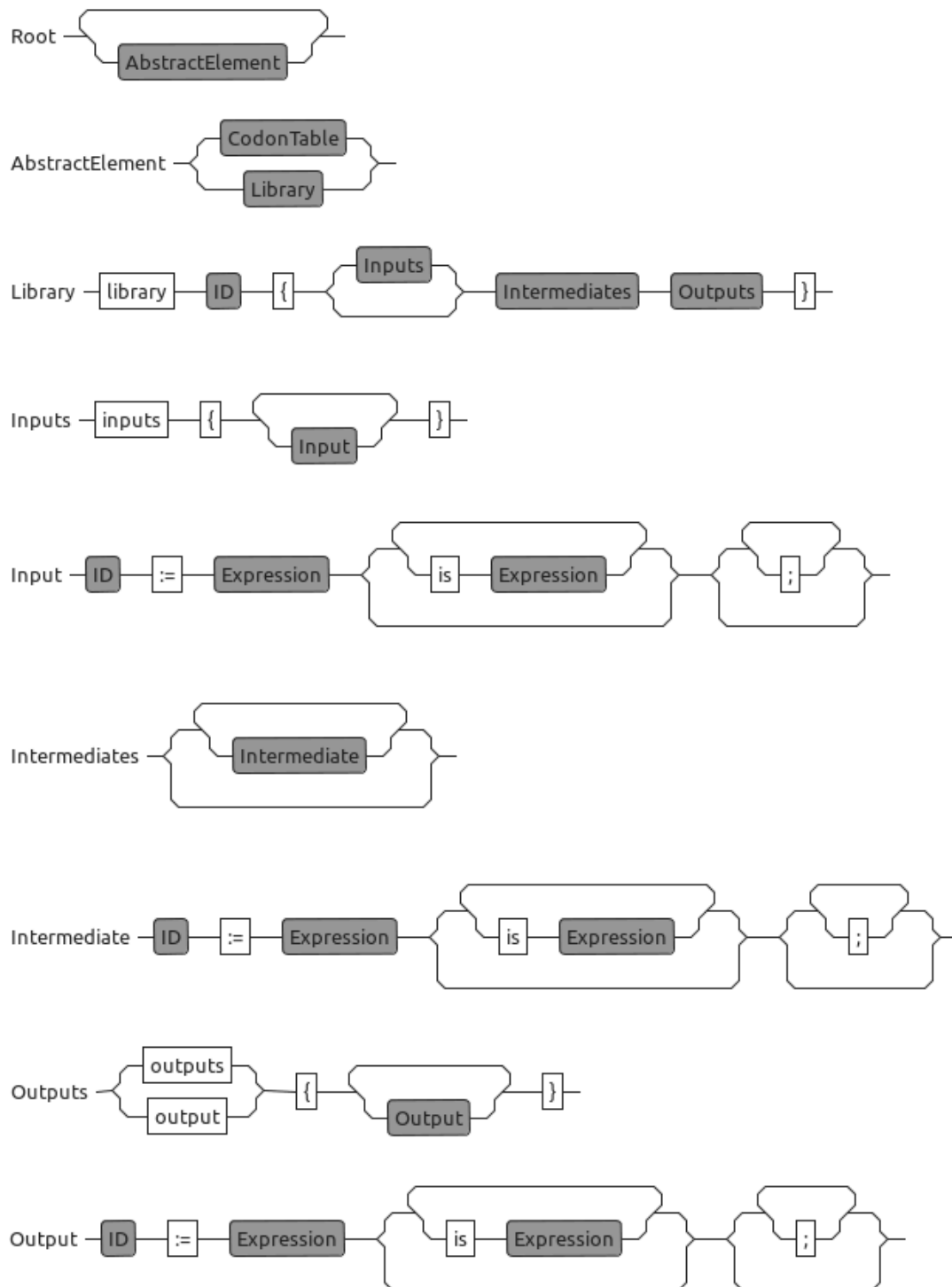


Figure 8.2: Railroad diagram of the file/library structure from the DNALD grammar.

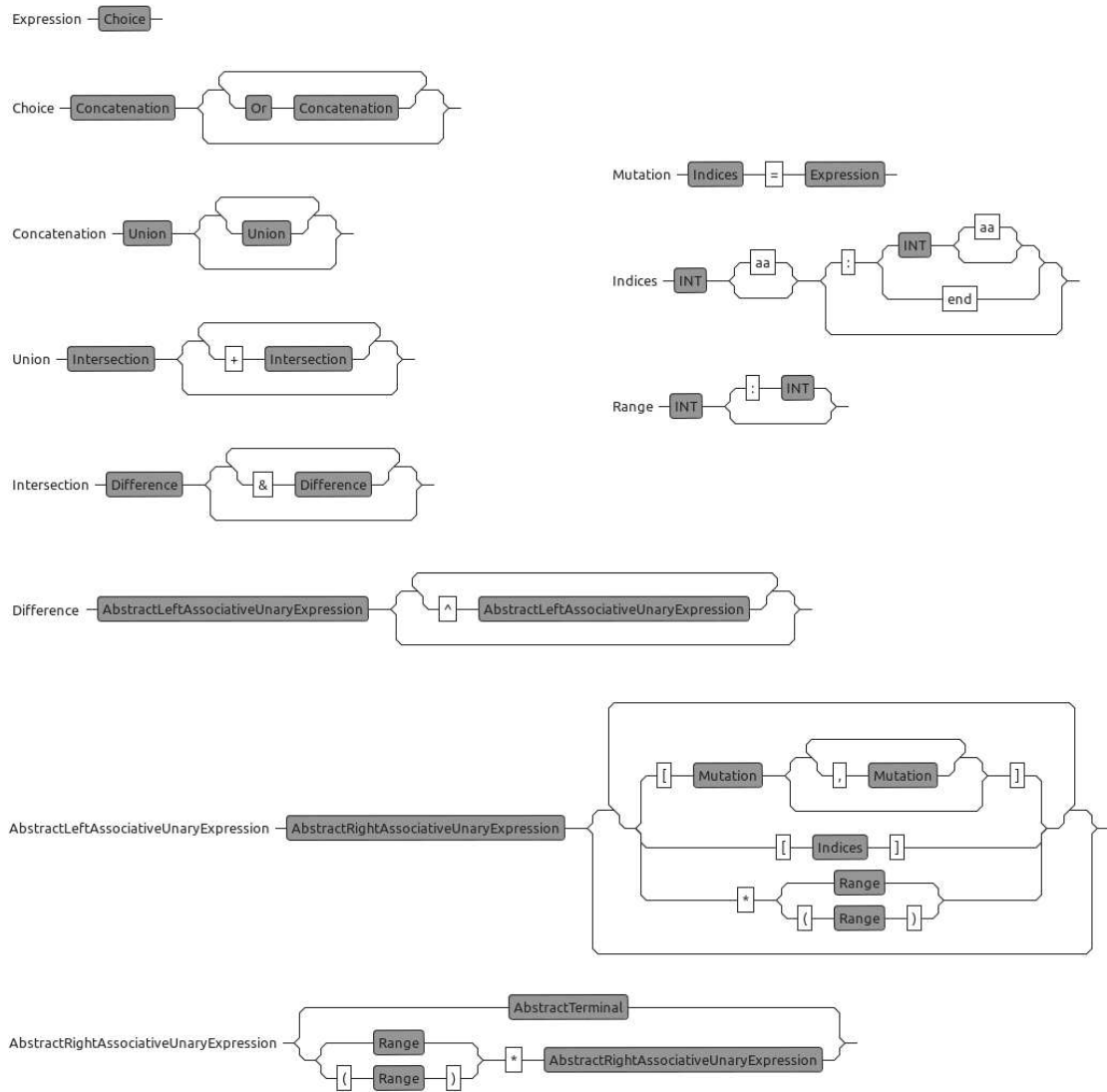


Figure 8.3: Railroad diagram of the DNALD expression grammar.

to: <http://blog.efftinge.de/2010/08/parsing-expressions-with-xttext.html>, so that each expression delegates to a higher priority expression or a terminal but never a lower priority expression which would create recursion. The top level expression `Expression` can however be reused as an attribute of a subexpression. Figure 8.4 shows a railroad diagram of the current left-factored DNALD expression grammar.

Terminals Figure 8.4 shows the terminals of the DNALD grammar to which allow expressions must eventually delegate. `AbstractTerminal` groups the terminals as a bridge from the expression grammar. Note that the terminal `ParenthesizedExpression` contains an `Expression` attribute which enables reuse of the expression grammar with increased priority, effectively implementing the standard functionality of parentheses directly as a consequence of the grammar.

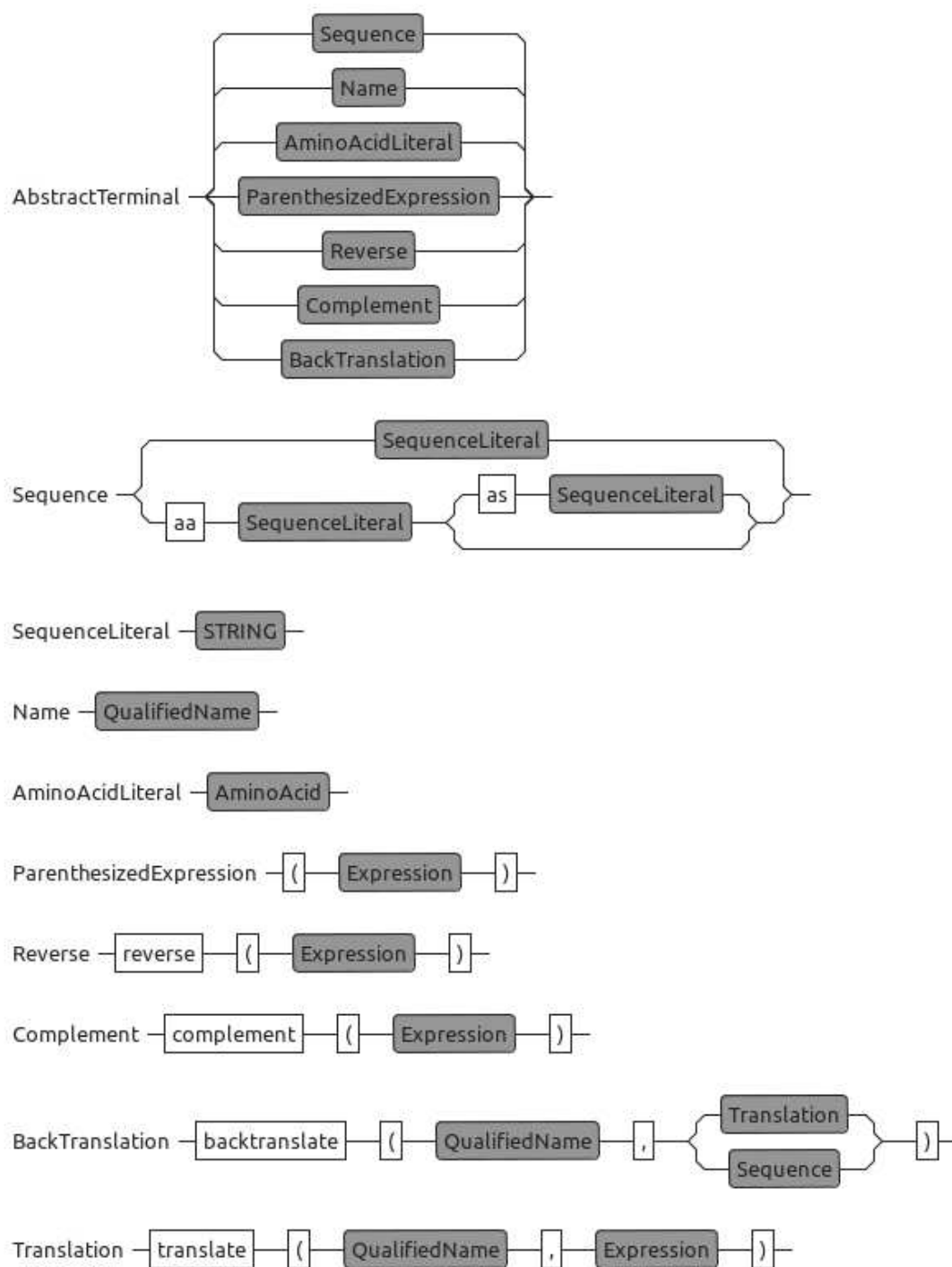


Figure 8.4: Railroad diagram of terminals in the DNALD grammar.

8.1.3 Parsing DNALD into Ecore models

Xtext uses the rules of the grammar to generate a metamodel of the language comprised of a Java interface and an implementing class for each rule. Parsing a DNALD file creates an abstract syntax tree (AST) that Xtext walks to populate a Ecore model, conforming to the metamodel, with instances of the rule objects containing the parsed data. Cross-linking between objects, such as when a DNALD expression references another definition, are also established at this point. We interact with the parsed model instance through its metamodel API when we are performing validation checks and evaluating the DNA library design.

To integrate with Eclipse, Xtext registers the `.dnald` file extension with the EMF registry used by Eclipse to determine which editor to open different file types with, in this case an Xtext-generated editor. Parsing happens transparently as the files are edited and syntax errors are handed to the validator. For testing and access to the parser outside Eclipse, e.g. for a command line evaluator, it is necessary to inject an `IResourceSetProvider` which refers to the EMF registry and provides an `IResourceSet` that loads and parses DNALD files, returning the parsed Ecore model.

The Ecore model is used by the validator for first pass semantic validation and as the input to the evaluator, whose output is the evaluated DNALD or *datamodel* (discussed in section 8.1.5).

8.1.4 Validation

Syntactic validation of DNALD files is managed by Xtext because it can be determined from the grammar (described above). Customisations to provide less generic and more informative error messages are achieved by implementing the `ISyntaxError MessageProvider` interface and registering the implementation with the Guice injector through `DNALDRuntimeModule` class. Our implementation translates, for instance, the message “no viable alternative at input ‘}’” into “library must contain an outputs section” when it can be sure that was the cause.

Semantic validation occurs twice: once when the DNALD is parsed into an Ecore model and again when it has been evaluated because more information is available. Semantic validation of parsed DNALD models occurs in our Guice injected (Xtend) `DNALDValidator` subclass of the Xtext generated `DNALDJavaValidator` class. `@Check` annotated methods are called automatically with appropriate model objects by Xtext when we evaluate the DNALD resource or it is edited in the GUI. At this point we check for out-of-range indices, invalid nucleotide codes, duplicated codons and uncomputable codon usages. Errors and warnings are logged and reported in the GUI as described in section 7.2.

`DNALDValidator` is itself injected with a `DNALDResourceScopeCache` singleton that it shares with the evaluator. This cache is updated with the evaluated object after a successful

evaluation. A subset of `@Check` annotated methods in the validator fail-fast if there is no evaluated object for the model object in the cache. When there is an evaluated object these check methods implement the reporting of errors which were created in the evaluator as a result of semantic errors in the parsed model. For example, if the DNALD library is not a directed acyclic graph (see section 6.2.2) this reraises errors such as “Expression creates cyclic dependency: $A \rightarrow B \rightarrow C \rightarrow A$ ” for each definition in the cycle.

During the second round we also test assertions of the form `name := expression (is assertion) *`, where each `assertion` is also an expression that evaluates to a sequence set. The assertion is true if the sets are equal (the set of sequences of the members regardless of fragment composition are equal). For example in the definition `x := 'ac' 'gt' is 'acgt' is 'acg'` the first assertion, that the concatenation of AC and GT is ACGT, is true, but the second is false. False assertions raise the warning “Evaluation does not match assertion” on the defining expression.

8.1.5 Evaluating DNALDs to DNA libraries

Before discussing how parsed models of DNALD files and the expressions within are evaluated it is important to understand how we model sequences in our implementation of DNALD. The key feature of our implementation is that it tracks the origin of sequence fragments throughout the DNA library.

Sequence data model

We model sequences as concatenations of sequence fragments. More precisely, instances of `DNALDSequence (sequence)` hold a list of `DNALDSequenceFragmentReference` instances (references). Each `DNALDSequenceFragmentReference` points to a `DNALDSequenceFragment (fragment)` and specifies a subsequence of that fragment with start and stop indices ranging from 1 to the length of the fragment. The DNA sequence of a `DNALDSequence` is the concatenation of the fragment subsequences specified by its references in the order they appear in the list.

Given the two DNALD definitions below:

```
x := 'ACGT'
y := x[2:3='GC']
```

the expression of `x` creates a new `DNALDSequenceFragment X` with the 4 nucleotide long sequence **ACGT**, but evaluates to a `DNALDSequence` with a single `DNALDSequenceFragmentReference` specifying the whole sequence of `X`, which we can write a `X[1 : 4]`. The expression of `y` creates a new

`DNALDSequenceFragment Y` with sequence `GC`. In our sequence model `y` evaluates to a `DNALDSequence` with three `DNALDSequenceFragmentReferences` `X[1]`, `Y[1 : 2]`, `X[4]` because the `y` mutates `x` replacing `CG` with `GC` but leaves `A` and `T` intact.

Our sequence model captures the interrelatedness of the sequences which will be manufactured, enabling the library designer to trace the propagation of sequences through the library and giving the library manufacture a head start in designing primers to extract and join subsequences of various fragments (assuming that the library design is relatively optimal in terms of sequence reuse).

This scheme is relatively impervious to poor library design. Copy-and-pasting the same, perhaps complex, expression several times, instead of creating and reusing an intermediate defined by that expression, will still result in sequences containing references to the original fragments. Sequence reuse can be artificially limited however, when the same or potentially derivable sequences are repeatedly stated and not referenced. The current implementation does not cache identical fragments (which would require linking the evaluated objects more tightly to the resource which defined them), or automatically refactoring/editing definitions to make optimal use of previously stated fragments, which is more challenging still. A suitable compromise would be a quick fix that suggests potential refactorings as this would alert the designer to the possibility of improved sequence reuse and potentially unforeseen symmetries in their library.

Sequence sets and fluent APIs

The language specification of DNALD (section 6.2.2) mandates that all operations return sequence sets. In our implementation of the DNALD datamodel `DNALDSequenceSet` not only collects sequences but is responsible for performing most of the languages operations on its members. The `DNALDSequence` class is designed so that it exposes methods that mirror the operations of the DNALD language pertaining to sequences (subsequence slicing, mutation, repetition, reverse and complement) and implements these in terms of the `DNALDSequenceFragmentReferences` that compose it. The `DNALDSequenceSet` class is designed so that it too exposes methods that mirror language operations, implementing those that pertaining to sets and delegating those that do not every member sequence. **Importantly, none of these methods change the state of their object or those within, returning only new altered copies. These classes are immutable with the concomitant benefits of thread safety and thereby amenability to parallelisation.**

The main methods of `DNALDSequenceSet` fluent API are:

`slice` delegates slicing to each `DNALDSequence` in turn. These return a new `DNALDSequence` of a deep-copied and truncated list of references to fragments, with the start and stop

indices of the 5' and 3' terminating fragments adjusted as necessary, are gathered into a new `DNALDSequenceSet` and returned.

`mutate` as for `slice` above, but the returning `DNALDSequences` have altered fragment references that are interspersed with those of belonging to each sequence in the mutation set.

`repeat` integer repetitions result in a new **union** where each sequence is composed of the concatenation of its input sequence's sequence fragment references, repeated *n* times (the number of repetitions). Variable length repetitions with lower and upper integer bounds on *n*, follow the same procedure for each possible value of *n*, but return a **choice** of those sequences in the set.

`concatenation` creates a new sequence for each sequence in the left hand set by concatenating the fragment references of that sequence to every member of the right hand set (the Cartesian product). It calls, with a list of just left and right, a more general method that passes any number of `DNALDSequenceSets` to Guava's `Sets.cartesianProduct` to produce every possible list that can be formed by choosing one element from each of the given sets in order, and concatenates the fragment references of all the sequences in each list to create a new `DNALDSequenceSet`.

`union`, `intersection`, `difference` and `symmetricDifference` delegate to their namesakes in the Google Guava's `Sets` helper class too. Guava's implementations of these, `cartesianProduct` above and other data structures like `ImmutableSet` are extremely well tested, performant and memory efficient, which improves the robustness of our DNALD implementation.

Shifting the computation of sequence and sequence set composition to methods on their implementing classes has a number of benefits. Primarily, it means that code resides where it is most relevant: evaluator code that is concerned with evaluating the meaning of parsed expressions belongs in the evaluator not in the sequence model. Secondly, the sequence composition code is more testable without the boilerplate necessary to create and invoking an evaluator on a string of DNALD, improving understandability. Lastly and importantly, it constitutes a *fluent interface* which both we as test writers and other programmers can utilise to perform traceable (via fragment references) manipulations on sets of DNA sequences.

Evaluation strategy

A standalone static method accepts an array of DNALDs and creates an evaluator which is then used repeatedly to evaluate each DNALD, populating an array of `EvaluatedDNALD` instances to return.

The evaluation of an individual DNALD proceeds as follows:

1. The contents (text) of the DNALD is parsed, yielding a `Ecore` model containing zero or more codon table definitions and one or more library designs (as a tree of `EObjects`, the types of which correspond to grammar rules). Syntactic errors are discovered but not immediately reported.
2. The semantic model is traversed to ensure that links between models objects which are lazily resolved in `Xtext` are fully resolved.
3. The resource containing the semantic model is then validated to obtain a list of issues: syntactic, linking and pre-evaluation semantic errors such as erroneous indices in subsequence operations and, crucially, cyclic dependencies between definitions (section 6.2.2 explains why), any of which will cause the evaluation to abort gracefully.²
4. An evaluation context is created which will hold the values computed from the expressions and assigned to a name by each definition. The context is initially populated with the default codon table from *E. coli* K12 shown in listing 6.4.
5. From this point onwards each model object, starting with the root, is evaluated in the current context by polymorphic dispatch to the relevant, type-checked `evaluate` method³. All `Expression` objects evaluate to a `DNALDSequenceSet` while other objects evaluate to corresponding evaluated version of their parsed form. If the object being evaluated has a name (i.e. it is a codon table, a library or a definition), the `evaluate` method updates the context by binding its result to name before returning that result to the caller.
 - The root object evaluates each codon table and library, adding the resulting objects to the `EvaluatedDNALD` it returns.
 - Each `CodonTable` evaluates to a `CodonUsageTable` which computes the probabilities of each codon encoding a particular amino acid and derives mappings for translation and back-translation.
 - Each `Library` evaluates to an `EvaluatedLibrary`, populating sets of evaluated inputs, intermediates and outputs with the results of the evaluating each of the definitions it

²It is possible however, to prune definitions with errors (in cycles or otherwise) and those that depend on them, before proceeding to evaluate the remainder. This is only relevant when the DNALD is being edited in DNA Library Designer, because it enables issues identified by post-evaluation semantic validation of these to be reported to the user, even though errors exist elsewhere.

³Xtend's `dispatch` methods (<http://www.eclipse.org/xtend/documentation/index.html#polymorphicDispatch>) obviate the need for the quite invasive visitor pattern.

contains. Prior to evaluating the definitions, all definitions with errors are pruned from a copy of the library's dependency graph, along with any dependent definitions recursively. The remaining valid definitions are evaluated in topological sorted order using JGraphT's `TopologicalOrderIterator` which uses Tarjan's algorithm [257].

- Each `Definition` evaluates to the `DNALDSequenceSet` which results from evaluating its expression.
- Sequence expressions evaluate to a `DNALDSequenceSet` containing one `DNALDSequence` that is comprised of one `DNALDSequenceFragmentReference` pointing to full sequence of a newly created `DNALDSequenceFragment`.
- Names evaluate to the value of their name in the current context.
- `AminoAcidLiterals` and `AminoAcidReferences` evaluate to the choice of codons defined by the three letter code of the amino acid in the default or a named codon table respectively.
- `Subsequence` and `Mutation` expressions both switch on the various combinations of parameters (whether only a single index was specified, or if not whether a stop index or the **end** keyword was given) and call the appropriate `slice` or `mutate` method on the `DNALDSequenceSet` with, in the case of a `Mutation`, the result of evaluating its mutation expression. As usual a new `DNALDSequenceSet` is returned.
- The other expression types, mainly concatenation and set operations, devolve the actual implementation of evaluation to methods of the `DNALDSequenceSet` resulting from evaluation of their input expression.

The previous two chapters detailed the design and implemented functionality of DNA Library Designer. The first half of this unevenly split chapter summarised the core contribution of the language implementation and software engineering done in the CADMAD project. We now present a short discussion of some software engineering notes that will hopefully be of use to future developers of the Infobiotics Workbench, last seen in Chapter 5.

8.2 The Infobiotics Workbench

8.2.1 Software stack

The Infobiotics Workbench software stack, including those of the experiment executables and the Dashboard, are summarised in figure 8.5. The functionality the direct dependencies provide for our software are:

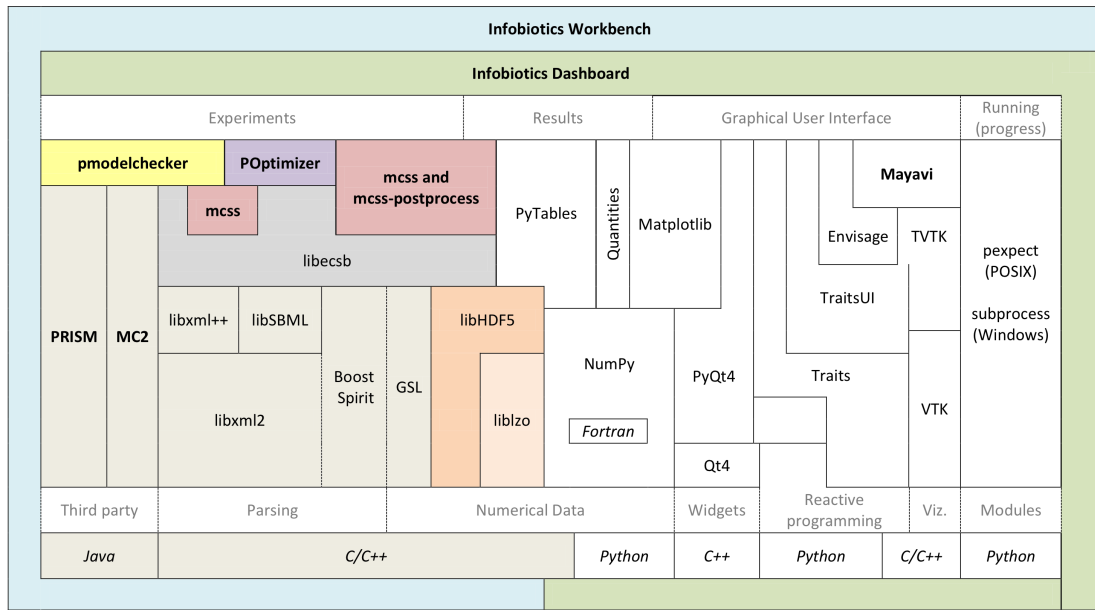


Figure 8.5: The Infobiotics Workbench software stack. The diagram should be viewed as a stack where each layer above is dependent on functionality provided by the layer below. At the bottom are the programming languages the tools above are written in. The Infobiotics Workbench and the Infobiotics Dashboard are mutually dependent as the Workbench contains the experimental executables needed by the Dashboard and the Dashboard is the GUI of the Workbench. This is shown by the entwined blue and green areas.

PRISM and MC2 probabilistic and simulative model checkers, simply executed with appropriate command line arguments, as specified in the (Dashboard generated) .params file, by PMODELCHER using the C++ stdlib system function.

libxml2, libxml++ reading and writing P system XML files (libxml2 is also a dependency of libSBML).

libSBML reading MCSS-SBML files into Infobiotics experiments

Boost Spirit implementing Domain Specific *Embedded* Languages (DSELs) grammars, from which parser objects can be generated for the human-writable LPP system, SP system, lattice and module library data formats (see section 4.3.3 on page 78 for example file listings). The LPP parser was written by Dr. Francisco Romero-Campero.

GSL the GNU Scientific Library [240] provides extensively tested functions which LIBECSEB uses for random number generation, MCSS for fast logarithms, ODE solvers and distributions, and MCSS-POSTPROCESS for the function `Tcdfinv` used to calculate the confidence interval of the standard deviation of the sample (number of simulation runs) at each simulation logging interval. GSL is licensed under the GNU Public License⁴.

⁴Due to the “copyleft” provision of the GNU Public License (GPL) all software that uses it, including the Infobiotics Workbench, must also be released under the GPL, recursively.

<http://www.gnu.org/software/gsl/>

libHDF5 1.8 high-performance, hierarchical data storage format used by MCSS to write simulation output that is read by mcss-postprocess and the Dashboard (see PyTables below).

PyTables to retrieve simulation data from HDF5 on demand. PyTables combines HDF5, Python, NumPy and Cython, providing extremely simple and fast programmatic access to data stored in HDF5 arrays.

liblzo used by libhdf5 and PyTables as a compression filter. Compression is turned on and set to the maximum level by default, as the time spent writing large datasets to disk is generally greater than the time spent compressing, actually boosting performance. 50x compression is often achieved, due to MCSS storing the quantities of all species at each logging interval, when it can be the case that many do not change between timepoints, particularly in large models, making the datasets particularly amenable to compression.

NumPy is the fundamental package for working with arrays in Python. It is used by the Dashboard primarily to compose high-dimensional arrays of simulation and model checking results, and to perform statistics along their axes. Together with its extension SciPy, NumPy fulfills most of the same functionality of GSL but for Python rather than C++.

Quantities unit conversion (time, lengths, volumes) and amounts scaling of species quantities in the Dashboard. On top of Quantities default units, we defined molecules mole and molar UnitQuantity objects.

<http://packages.python.org/quantities/user/tutorial.html>

Matplotlib [245] 2D plotting of timeseries and histograms (section 5.2).

pexpect running and communicating with Infobiotics experiment executables under POSIX. pexpect made it possible to report the progress of experiments to the user when all else failed. Unfortunately none of the existing Windows ports could be made to work. <http://www.noah.org/wiki/pexpect>

Qt4 and PyQt design and implementation of the simulation results and surface comparison interfaces.

Mayavi, TVTK and VTK 3D plotting of colony-level surface plotting and PRISM results interfaces. Mayavi [246] is also based on Traits.

Traits, Traits UI and Envisage validating experiment parameter classes; timeseries, surface and histogram datamodels; Workbench UI.

http://docs.enthought.com/traits/traits_user_manual/

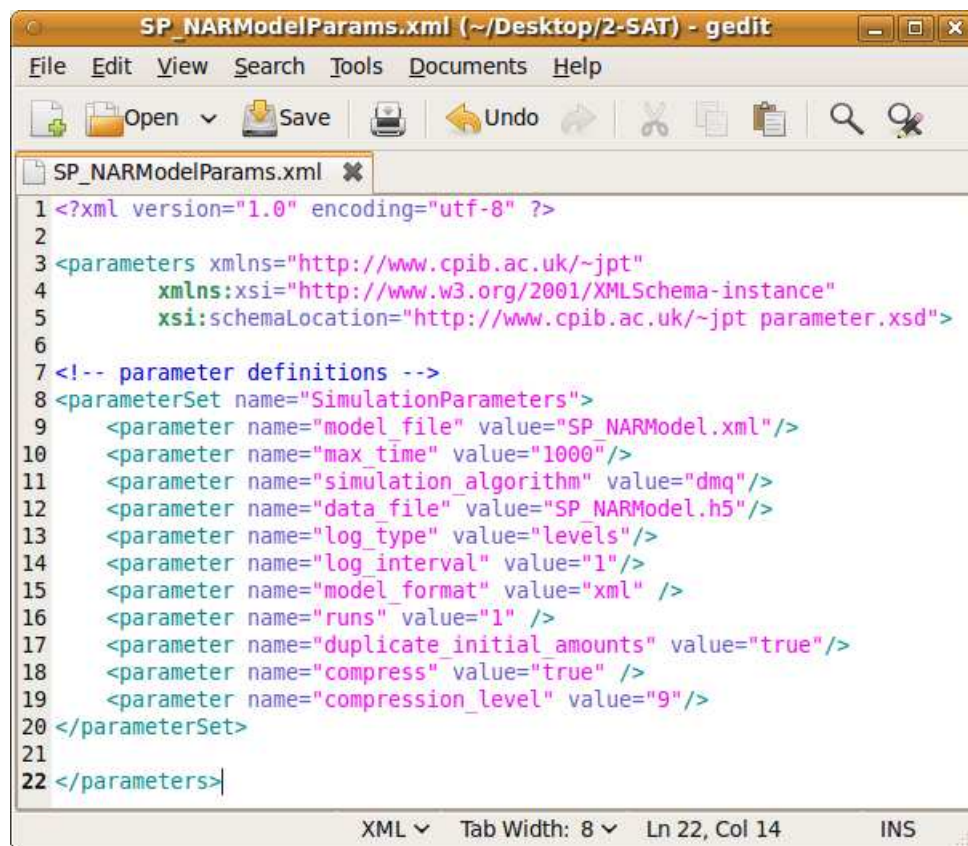
8.2.2 Experiment parameter classes of the Infobiotics Dashboard

The primary role of the Dashboard is as a container in which the various Infobiotics experiments can be presented as an integrated software suite. This is achieved by reusing as many of the GUI patterns as possible between components interfaces, while still using appropriate customisations when necessary. For example, the loading, saving and performing subinterfaces are common to all experiments. Experiment progress is interpreted and reported similarly. The results interfaces differ because they are tailored to the data but still complex subinterfaces such as 2D and 3D plotting widgets are also shared. **The overall intention is to create a sense of unification when in fact the underlying implementation is fairly heterogeneous.** A case in point was experiment parameter handling.

The executables implementing Infobiotics experiments are parameterised via XML format *parameters files* with the extension `.params` or `.xml` (figure 8.6 shows an MCSS parameters file). Importantly, this decoupling of experiment parameters from models and the analysis of experimental results enhances the reproducibility of *in silico* experimentation by allowing multiple sets of experimental parameters to be used with each model and shared between experimenters. Within a parameters file each parameter element has a `name` and `value` attribute. Each experiment executable parses the parameters file using an object of a C++ class generated by the `make-parameter-class` executable of LIBECSB from a *parameters template file* such as `mcss-parameters-template.xml` (figure 8.7). As well as making the generation of parameter parsing simple to update for the different experiments, it was expected that this would be enough information from which to generate the parameter setting interfaces in the GUI.

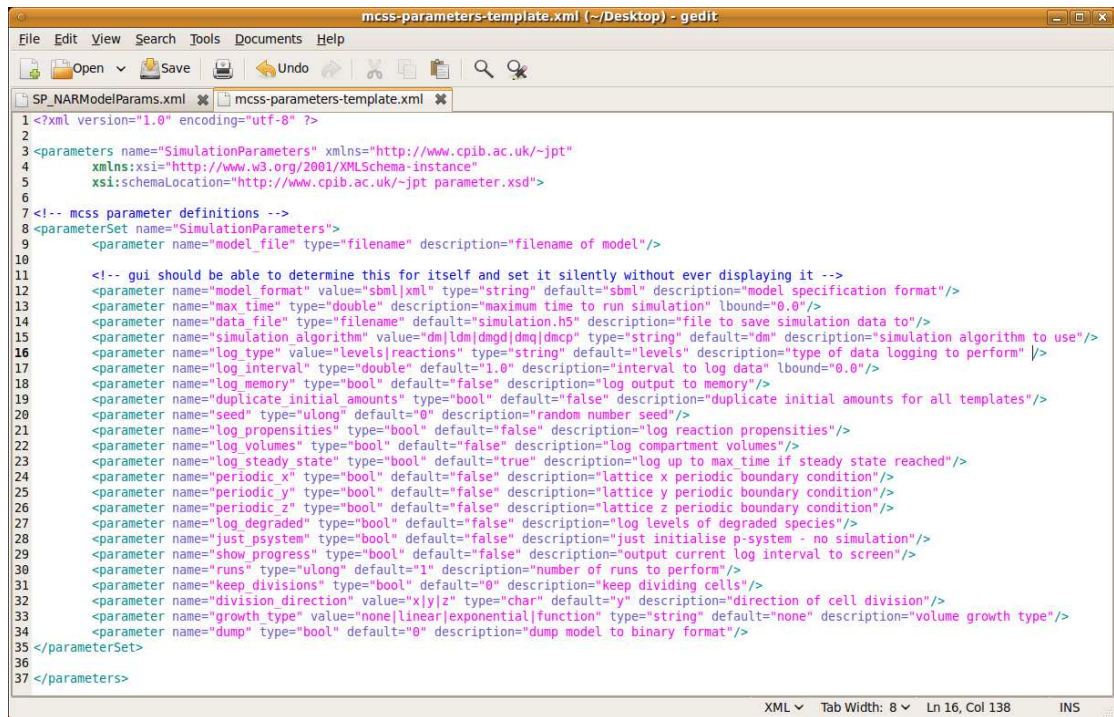
Generative Qt4/C++ prototype The initial Qt4/C++ experiment parameterisation interface of the Infobiotics Workbench was generated from parameter-template files such as the one shown in figure 8.7 that are used by all of the Infobiotics experiment components written in C++ (MCSS, PMODELCHECKER and POPTIMIZER) for their own code generation of parameter handling classes. It was assumed that by using the same data (in the parameter-template.xml files) to generate both the command line and graphical user interfaces for experiment parameterisation that the interfaces could be easily kept in sync as the capabilities of individual components changed.

Rather than perform an offline code generation step to obtain the GUI interface an online approach was taken where parameter-template files distributed with the program were read in at runtime and created the interface on-the-fly. The intention was to eventually create an interface system that would be capable of dynamically adapting to changes in the values of parameters, to hide or reveal certain parameters whether or not they depended on the values of others, as a means of reducing the complexity of the experiments. In this schema each type



```
1 <?xml version="1.0" encoding="utf-8" ?>
2
3 <parameters xmlns="http://www.cpib.ac.uk/~jpt"
4             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5             xsi:schemaLocation="http://www.cpib.ac.uk/~jpt parameter.xsd">
6
7 <!-- parameter definitions -->
8 <parameterSet name="SimulationParameters">
9   <parameter name="model_file" value="SP_NARModel.xml"/>
10  <parameter name="max_time" value="1000"/>
11  <parameter name="simulation_algorithm" value="dmq"/>
12  <parameter name="data_file" value="SP_NARModel.h5"/>
13  <parameter name="log_type" value="levels"/>
14  <parameter name="log_interval" value="1"/>
15  <parameter name="model_format" value="xml" />
16  <parameter name="runs" value="1" />
17  <parameter name="duplicate_initial_amounts" value="true"/>
18  <parameter name="compress" value="true" />
19  <parameter name="compression_level" value="9"/>
20 </parameterSet>
21
22 </parameters>
```

Figure 8.6: Example parameter file for a simulation experiment.



```

1 <?xml version="1.0" encoding="utf-8" ?>
2
3 <parameters name="SimulationParameters" xmlns="http://www.cpi.ac.uk/~jpt"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://www.cpi.ac.uk/~jpt parameter.xsd">
6
7 <!-- mcoss parameter definitions -->
8 <parameterSet name="SimulationParameters">
9   <parameter name="model_file" type="filename" description="filename of model"/>
10
11   <!-- gui should be able to determine this for itself and set it silently without ever displaying it -->
12   <parameter name="model format" value="sbml|xml" type="string" default="sbml" description="model specification format"/>
13   <parameter name="max time" type="double" description="maximum time to run simulation" lbound="0.0"/>
14   <parameter name="data file" type="filename" default="simulation.h5" description="file to save simulation data to"/>
15   <parameter name="simulation algorithm" value="dm|ldm|dmq|dmcp" type="string" default="dm" description="simulation algorithm to use"/>
16   <parameter name="log type" value="levels|reactions" type="string" default="levels" description="type of data logging to perform"/>
17   <parameter name="log interval" type="double" default="1.0" description="interval to log data" lbound="0.0"/>
18   <parameter name="log memory" type="bool" default="false" description="log output to memory"/>
19   <parameter name="duplicate initial amounts" type="bool" default="false" description="duplicate initial amounts for all templates"/>
20   <parameter name="seed" type="ulong" default="0" description="random number seed"/>
21   <parameter name="log propensities" type="bool" default="false" description="log reaction propensities"/>
22   <parameter name="log volumes" type="bool" default="false" description="log compartment volumes"/>
23   <parameter name="log steady state" type="bool" default="true" description="log up to max time if steady state reached"/>
24   <parameter name="periodic x" type="bool" default="false" description="lattice x periodic boundary condition"/>
25   <parameter name="periodic y" type="bool" default="false" description="lattice y periodic boundary condition"/>
26   <parameter name="periodic z" type="bool" default="false" description="lattice z periodic boundary condition"/>
27   <parameter name="log degraded" type="bool" default="false" description="log levels of degraded species"/>
28   <parameter name="just psystem" type="bool" default="false" description="just initialise p-system - no simulation"/>
29   <parameter name="show progress" type="bool" default="false" description="output current log interval to screen"/>
30   <parameter name="runs" type="ulong" default="1" description="number of runs to perform"/>
31   <parameter name="keep divisions" type="bool" default="0" description="keep dividing cells"/>
32   <parameter name="division direction" value="x|y|z" type="char" default="y" description="direction of cell division"/>
33   <parameter name="growth type" value="none|linear|exponential|function" type="string" default="none" description="volume growth type"/>
34   <parameter name="dump" type="bool" default="0" description="dump model to binary format"/>
35 </parameterSet>
36
37 </parameters>

```

Figure 8.7: An example parameters template file used to generate a parameters class for each Infobiotics experiment. Each parameter element in the template file contains an extra attribute `type` which defines the C++ data type a parameter of that name will set the value for. The `value` attribute contains either a default value or a set of possible values for an `enum` type delimited with an `|` character. Optional upper and lower bound attributes for numeric types and a `description` attribute may also be specified.

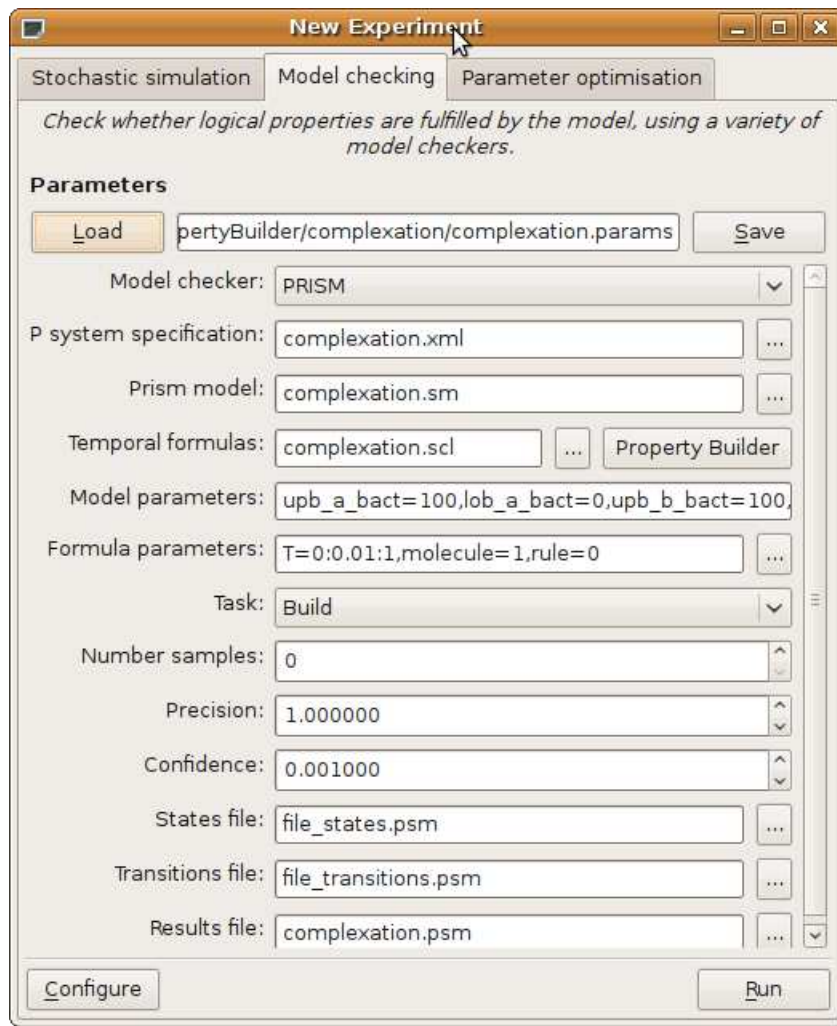


Figure 8.8: Generated parameterisation interface for model checking experiments using the Qt4/C++ implementation that was later shelved.

of parameter generates one of the several possible widgets that could be used to interact with that type. Classes for each parameter type were created that could be constructed with values and constraints given in the parameter element's attributes. Names were treated separately to create the aesthetically pleasing two-column arrangement shown in figure 8.8. Description attributes were converted into tooltips (not shown). Specialization was achieved by mapping a combination of parameter template filenames, `parameter` and `parameterSet` XML element name attribute values, parameter names or types to custom widget implementations. For example, the Property Builder button (shown in figure 8.8) was implemented by extending the `FileParameterWidget` class with a special case for parameters of type file named 'temporal_formulas'.

Despite successfully coding the working prototype in figure 8.8, it quickly became clear that this was not a suitable strategy for constructing the experiment parameterisation GUIs. Es-

entially, the information contained in the template files is insufficient to enable generation of a high-quality, dynamic interface that reflected the semantics behind the choice of parameter values, the main reason for producing such an interface. Several problems were identified. Mutual exclusivity of certain parameters depending on the mode of operation of the experimental component cannot be captured in the template file format as it was . Additionally, the lack of structure in the XML format (other than the order of parameters in the template) means that related parameters cannot be grouped to better highlight the existing synergy. The ordering of parameter elements in the XML was used to order the parameter widgets in the generated interface. These were wrapped in a scrollable widget to ensure that the dialog did not become too tall for the screen. Beyond constraining numeric parameters with maximum and minimum values or presenting choices in a combo box, the semantic validation of parameter values is impossible because the generation process itself is not governed by a model of each experiments operation. Users of a GUI dialog expect it to provide only a set of valid options, so that when they eventually click 'OK' (or 'Run' in this case) they know that the choices they have made are really OK, if this is not true then the user experience is deeply unsatisfactory and people will quick tire of using the software [258].

It is interesting to note that none of these issues are problematic for a generated command line interface. On the command line irrelevant parameters can simply be ignored by the program, although it would be better to provide feedback explaining this. Parameters can be given in the order the user prefers, meaning they can impose their own understanding of the semantics of the parameter set. Most saliently, the user expects the program to exit early if they pass it an invalid parameter value, and hopefully they will receive an explanation for the failure. This very much contrasts with the expectations users have of a graphical user interface.

Python/Traits-based solution Adding the validation layer in the generative approach was considered to be outside the remit of the project. We decided to switch to a hard-coded approach and identified a framework, EnthoUGHT Traits for Python, that had the necessary features to enable a dynamic, validating GUI. The general aesthetic can be seen in figure 5.6, which shows the input parameters of a simulation experiment and the progress of a running simulation, conducted within the main interface of the Dashboard.

Traits are typed-attributes of Python objects that provide the following advantages:

1. Initialisation: default values, static when known in advance, dynamic when dependent on other traits or object attributes - used to manage dependent and mutually exclusive parameter combinations.

2. Validation: type-checking (for a dynamically typed language) **and** value-checking at the point of attribute setting. Invalid values are highlighted red, a tooltip displays the reason and experiments with invalid parameters are prevented from performing.
3. Notification: setting a traits value triggers events which can be statically or dynamically linked to other traitled objects including visualisations so that they are automatically updated. This feature was used extensively to synchronise parameters, sometimes across GUIs.
4. GUI generation via TraitsUI: typed-attributes have default editors, e.g. combo boxes for enumerations, sliders for ranges, open dialogs for files, which are used to create a graphical interface for manipulating traitled objects. Multiple views and handlers can be specified for the same objects and so a variety of interfaces appropriate to the situation can be used. We made full use of the various of handlers to create the most effective interfaces possible for each experiments parameter set. Custom trait handlers were even written for files with relative paths (not handled by the framework) that could validate whether a directory was writable or not.

These features together enable a style of programming known as Reactive Programming which results in clear, readable classes that integrate well into larger applications like the Dashboard. Furthermore Traits provided the impetus to switch development language from C++ to Python, which meant a lot of useful language features and libraries suddenly became available (PyTables, NumPy, Matplotlib, MayaVi) without which the analysis portion of the Infobiotics Workbench would not have been possible to produce within the duration of this project.

8.2.3 Handling simulation results with the `McssResults` class

The `McssResults` class provides a programmatic interface to the HDF5 output files produced by MCSS. In the Infobiotics Dashboard the main user of `McssResults` is the `McssResultsWidget` class which essentially provides a graphical wrapper around most of the simulation data extraction functions, and then uses that data to populate the various plotting interfaces it provides. Objects of `McssResults` are stateful, meaning selection of units, species, compartments, runs and timepoints are held and reused by various calls to methods returning either raw data species amounts or compartment volumes data as NumPy arrays or Traits-based objects for timeseries, surfaces or histograms.

A feature of `McssResults` that is not fully exploited by the Dashboard GUI is the ability to apply any functions over all or some of the simulation runs through the `functions_of_amounts_over_runs` method. The simulation results interface only uses this method to compute the mean and standard deviations of amounts over runs. When it does the runs dimension of the amounts array

(as extracted from the simulation data file in the previous section) is replaced with a functions dimension, so that 1000s of runs are replaced by the 2 statistics of the amounts of each selected species in each selected compartment at each (evenly spaced) selected timepoint. Developers interested in adding functionality to the Infobiotics Workbench and working with very large models should consider starting by exploiting the latent power of `functions_of_amounts_over_runs` through Python code or a live iPython session. The remainder of this section describes the implementation of the method for those that may follow after. Listings 8.1 and 8.2 contain the code of split over two pages.

`functions_of_amounts_over_runs` proceeds by attempting to create an empty $functions \times species \times compartments \times timepoints$ array in which to put the results, exiting with a warning (via a popup or print to `stdout` depending on whether the `McSSResults` instance has a parent widget or not) if the array is too large to allocate in memory. Ideally all of the data needed to compute the results could then be loaded into memory and the given reduce functions applied along the runs axis to populate the results. However, in the case where many datapoints are selected, or many functions specified, the size of the allocated array can be very large, limiting the amount of memory available into which to load the simulation data from disk. To handle this situation we implemented a chunking algorithm that takes advantage of the independence of the non-run axes by splitting the input data into manageable chunks along the timepoints axis and processing chunks one at a time to populate the results array.

A reasonable `chunk_size` is determined by repeatedly attempting to allocate a buffer array of size $functions \times species \times compartments \times chunk_size$ (initially the total number of timepoints), catching the `MemoryError` exception raised if unsuccessful and repeating with the floor of `chunk_size` divided by 2, either until the allocation of `buffer` is successful or `chunk_size = 0` (meaning an array of size $functions \times species \times compartments \times 1$ was too large to fit into memory, in which case the function suggests to the caller that they reduce the size of the datapoints selection in the same manner as before).

Next, the number of selected timepoints is floor divided by `chunk_size` to obtain the *quotient*; the number of times the buffer array can be fully filled with amounts data. The number of selected timepoints modulo the `chunk_size` is used to obtain the *remainder*: the size of the timepoints axis for a smaller buffer array. A parameterised inner function `iteration` is defined that extracts variable sized portion of the selected data into `buffer` using the private method `_extract_amounts` and fills the appropriate portion of the results array by applying each statistical function to `buffer` along the runs axis. `iteration` is then called the *quotient* number of times with `chunk_size`, updating the starting point for the next extraction after each. If `remainder > 0` a smaller buffer array is allocated and used in the call to `iteration` with `remainder` instead of `chunk_size`. Finally the array of calculated statistics over runs is returned.

```

1 def functions_of_amounts_over_runs(self, functions,
  quantities_display_type=None, quantities_display_units=None,
  volume=None, **ignored_kwargs):
    '''Returns a 4D array of floats with the shape (functions, species,
    compartments, timepoint).

    '''
    import types
    if type(functions) in (types.FunctionType, types.LambdaType):
        functions = (functions,)
    results = self._allocate_array(# outputs error message if anything
    goes wrong
    (
        len(functions),
        len(self.species_indices),
        len(self.compartment_indices),
        len(self.timepoints)
    ),
    'Could not allocate memory for functions.\n' \
    'Try selecting fewer functions, a shorter time window or a
    bigger time interval multiplier.'
    )
    if results is None:
        return

    # create biggest possible buffer
    chunk_size = len(self.timepoints)
    buffer = None
    while buffer == None:
        try:
            buffer = np.zeros(
                (
                    len(self.run_indices),
                    len(self.species_indices),
                    len(self.compartment_indices),
                    chunk_size,
                ),
                self.type
            )
        except MemoryError:
            if chunk_size == 0:
                if self.parent is not None:
                    message = 'Could not allocate memory for chunk.\nTry
                    selecting fewer runs, a shorter time window or a
                    bigger time interval multiplier.'
                    QMessageBox.warning(self.parent, QString('Out of
                    memory'), QString(message))
                else:
                    print message
                    return
            # progressively halve chunk_size until buffer fits into
            memory
            chunk_size = chunk_size // 2
            buffer = None
            continue

    h5 = tables.openFile(self.filename)

```

Listing 8.1: McssResults.functions_of_amounts_over_runs

```

1      def iteration(chunk_size=chunk_size, buffer=buffer,
    quantities_display_units=quantities_display_units):
        '''Fills buffer with amounts for all runs and updates results
        with outcome of functions applied to runs.'''
        self.amounts_chunk_stop = amounts_chunk_start + (chunk_size *
    5         self.step)
        self._extract_amounts(h5, buffer, amounts_chunk_start,
            self.amounts_chunk_stop)
        buffer, quantities_display_units =
            self.convert_amounts_quantities(Quantity(buffer,
                substance_units[self.quantities_data_units]),
                quantities_display_type, quantities_display_units, volume)
        self.stat_chunk_stop = stat_chunk_start + chunk_size
        for fi, f in enumerate(functions):
            stat = results[fi] # stat is a 'view' on results so change
    10         stat changes results
            stat[:, :, stat_chunk_start:self.stat_chunk_stop] =
                f(buffer, axis=0) # axis 0 is runs
        return quantities_display_units

    amounts_chunk_start = self.start
    stat_chunk_start = 0
    15    # for each whole chunk
    quotient = len(self.timepoints) // chunk_size
    for _ in range(quotient):
        quantities_display_units = iteration(chunk_size, buffer)
        amounts_chunk_start = self.amounts_chunk_stop
    20         stat_chunk_start = self.stat_chunk_stop

    # and the remaining timepoints
    remainder = len(self.timepoints) % chunk_size
    25    if remainder > 0:
        buffer = np.zeros(
            (
                len(self.run_indices),
                len(self.species_indices),
                len(self.compartment_indices),
    30                 chunk_size,
            ),
            self.type
        )
        quantities_display_units = iteration(remainder, buffer)
    35    h5.close()

    results = Quantity(results, quantities_display_units)
    return results

```



```

1 | sum = lambda array, axis: np.sum(array, axis, dtype=dtypedefault)
   | var = lambda array, axis: Quantity(np.var(array.magnitude, axis, ddof=1,
   | dtype=dtypedefault), array.units) if isinstance(array, Quantity) else
   | np.var(array, axis, ddof=1, dtype=dtypedefault)

```

Listing 8.2: NumPy reduce functions used by `functions_of_amounts_over_runs`.

Listing 8.2 shows Python `lambdas` wrapping NumPy reduce functions⁵, that are members of the `mcss_results` Python module, which demonstrate the form the input functions should take in order to cope with the quantities conversions. The functions have been wrapped in `lambda` functions in order to a) work around the Quantity object methods `std` and `var` not having a `ddof` (degree of freedom) parameter, and b) set `ddof` parameter to 1 (standard deviation or variance of the sample, as opposed to 0, the standard deviation or variance of the population, which is the default).

To reemphasise, `functions_of_amounts_over_runs` is a vital algorithm for our software. It allows us to obtain statistics over all simulation runs for a large number of model entities and timepoints, with unit conversion and volume fluctuations. Each of the timeseries, histogram and surface plotting features of the simulation results interface depend on it for their efficient operation.

In this chapter we have touched on some of the software engineering highlights of this work: our DNALD implementation and two central pillars of the Infobiotics Dashboard. The unifying themes were building on the best available libraries to create good, reliable programs that do something original, putting functionality where it can have the most benefit, and exposing that functionality through programmer friendly APIs. In the final chapter we restate the motivation for this research, summarise and evaluates the contributions, report ongoing refinements and suggests future directions.

⁵Reduce functions are universal functions (`ufuncs`) supporting broadcasting and type casting which operate on NumPy N-dimensional array objects (`ndarrays`) in an element-by-element fashion, applying a function across one axis of the array and returning an array with one fewer dimensions in which each item is the result of the function <http://docs.scipy.org/doc/numpy/reference/ufuncs.html>

Chapter 9

Conclusions and future directions

Chapter abstract

This chapter restates the motivations for the research, summarises and evaluates the contributions, reports ongoing refinements and suggests future directions.

9.1 Restatement of motivations

Biology, medicine and the pharmaceutical industry are adopting mathematical and new computational tools to analyse and integrate this data into dynamic models that can recreate normal and pathological states, and thereby suggest mechanisms and circumstances in which these are reached. Synthetic biology uses the same tools to design engineered bacteria that can accomplish novel functions, such as biosynthesis of valuable substances, intelligent drug release and larger scale assemblage of matter. Compiling these *in silico* representations into real biological entities requires editing the genomes of cells to insert DNA encoding novel or adjusted functions. Improving the tools on which the future progress of these fields rests is therefore of paramount importance today.

Working at the right level of abstraction is crucial to any modelling endeavour and this is especially true for systems as integrated, fault-tolerant and non-deterministic as those found in biology. Because cells and cellular populations are subject to and exploit stochasticity, we are, to some extent, forced to work at both the level of individual reactions involving genetic regulation and at the population level where communication between cells via small molecules mediates and is mediated by population genetics. Additional uncertainty exists at the level of the biologists comprehension of the system being modelled. Each unknown presents another degree of freedom, defining a space of models and their encodings. Only with a suitably generic approach and application-agnostic facilities for representing and reusing recurrent organisational patterns can we overcome the combinatorially explosive descriptions of increasingly complex systems.

9.2 Overview and contributions

The research presented in this dissertation has attempted to affect the expansion of two computer-aided methodologies in synthetic systems biology:

- modelling of cellular systems at the level of molecular interactions, from single cells to large multicellular systems;
- specification of DNA constructs needed to validate model-driven hypotheses and implement synthetic biological designs, from single sequences to large combinatorial DNA libraries.

We took the constructive approach of developing conceptual and software frameworks that reified the problems in order to uncover and overcome the theoretical and technical issues the scaling up of these processes raises. The key issue faced for both expansions was how to achieve scalability of specification from a single instance, cell or sequence, to very many instances with a high-degree of overlap between instances. This issue demanded a satisfactory and generic solutions to ensure the longevity of our software by not precluding unexpected usages. The solutions we developed involved the creation of modular languages for multicellular models and DNA libraries, where larger application-specific subcomponents are constructed from smaller reusable subcomponents. Effective tools for processing these languages into computational representations of the systems they were being used to model, and computing with those representations to produce useful results was another major effort.

We developed and delivered the Infobiotics Workbench, a software package for performing *in silico* experiments on Lattice Population P system models (described in chapter 4). Those experiments are: deterministic and stochastic simulation; exact, approximate and simulative model checking; parameter and model structure optimisation (discussed in detail in chapter 5). The experiments are conducted via the Infobiotics Dashboard which provides sophisticated, easy to use and consistent parameterisation, progress reporting and analysis interfaces (the design and development of the which is covered in chapter 8).

Our modelling framework provides a mesoscopic, continuous-time and discrete-quantitative rule-based formalism for spatially-discrete models of multicellular systems. Definitions of individual P systems with a hierarchical compartment structure representing either single cell, intracellular domains or subpopulations of cells, with rulesets that might include library modules, can be distributed on a regular lattice to model systems where spatial interactions determine system dynamics, grounded in stochastic chemical kinetics. This approach was underpinned by the ongoing biological investigations described in chapter 1 and the literature on mathematical and executable biology reviewed in chapters 2 and 3.

We extended stochastic P systems to include the concept of a module of rules, allowing decomposition of large rulesets into meaningful and composable units. These modules of rules may be parameterised, with the values of species, rate constants and compartments only partially-specified, and can therefore be made independent of a specific system, stored in libraries and reused in multiple models, improving consistency and speeding up model development.

Fully abstract modules capture just the structure of a reaction network, communicating the roles of species involved through the choice of parameter names. These are simple sets of related biochemical interactions such as gene regulatory motifs, modelled at a certain level of detail. Fully-specified modules capture well-characterised systems where all interactions and rates are known, which can be reused as off-the-shelf components in synthetic biological circuits.

Our initial SBML implementation of the new formalism enabled models with many compartments to be specified, visually using CellDesigner, without the need to replicate reaction networks for each instance of the same cell type, by defining modules as SBML compartments with reaction networks with all species and rate constants set (i.e. fully parameterised, a limitation of SBML). Different, related cell types can be defined by mixing in modules via compartment names. This format bootstrapped the design of our simulator and its HDF5 output data model, on which the simulation results plotting interfaces were built.

Our own XML format provided a more direct mapping of the LPP formalism to a machine-readable data structure without the conceptual constraints of SBML. The ability to specify parameterised modules meant we could develop model structure optimisation algorithms that recombined modules, rather than single reactions, for a more biologically plausible exploration of model space. Additional attributes describing the potential range and scale of rate constants support concomitant parameter optimisation.

Our latest DSL for LPP systems refined our XML format and made it human-writable, adding the ability to associate DNA sequences with modules representing genetic elements or proteins with known coding sequences, outputting concatenations of these sequences (when grouped in a higher order module) in FASTA format.

All of our formats are accepted by the simulator and model checking components of the Informatics Workbench, enabling multicellular systems to be specified succinctly using a variety of tools, processed by the Workbench and analysed verbosely with the Dashboard's visualisation capabilities. Any combination of model variables can be plotted, with the software imposing important constraints and leveraging these to setup plots correctly, in order to maximise the information gain from expensive computational experiments.

DNALD and its implementation DNA Library Designer, which currently represent one years work on a three year project, repeat the formula of designing and refining a language and its supporting software concurrently. DNALD enables large sets of variant DNA sequences to be

defined in a programmatic manner where operations corresponding to biological interventions like splicing and mutation are applied to sequence sets to generate large combinatorial DNA libraries.

9.3 Evaluation

The Infobiotics Workbench integrates the several researchers work, comprising a major software engineering effort with independent and integrative code bases. Furthermore, each component, the Dashboard in particular, builds on a large variety of third party libraries to deliver a more capable product than would have been possible starting from scratch. Ensuring the distribution of the software on multiple platforms, each with their own take on dependency management was particularly challenging. Coordinating the communication between components on different platforms was unexpectedly difficult also.

The processing of DNALD files and the resulting outputs is more tightly integrated than for Infobiotics models and their experiments, due to this author being the only developer and opting to develop solely within the Eclipse framework. Eclipse's OSGi implementation effectively solves the problem of adding discrete units of functionality as plugins/bundles, and reduced the distribution effort by building itself with the appropriate plugins for each target platform.

Regarding the practical use of the new languages introduced in this research, one criticism that can be made of the Infobiotics DSLs, which we sought to amend with DNA Library Designer for DNALD, is that tooling for working with the language is poor. For example, a new user's initial interaction with the Infobiotics Workbench will begin by following the tutorial provided in the Infobiotics documentation¹ and adapting the examples using the simple code editor provided by the Infobiotics Workbench. The editor does not offer ongoing visual clues such as syntax highlighting of keywords or structural aids like code folding of subsections. In the current implementation syntax checking occurs only when the model is consumed by an Infobiotics experiment (simulation or model checking). If the parser encounters an invalid token it prints to STDOUT the line number of the offending token which is caught by the Dashboard and used to jump the editor to the line containing the error. If the model parses successfully it is still possible that it is semantically incorrect: a species name might be misspelled in a rule so that it does not appear in the alphabet; the upper bound of a range may be less than the lower bound. All of these problems should be detected and reported together, ideally overlaid on the model files being edited. However, the parser exits immediately when it encounters an error, so that only a single error can be reported for each attempt at performing the experiment. This results in a negative user experience of having to repeatedly perform the experiment only to discover that

¹Tutorial Using the Infobiotics Modelling Language: http://www.infobiotic.org/tutorial/tutorial_1.html

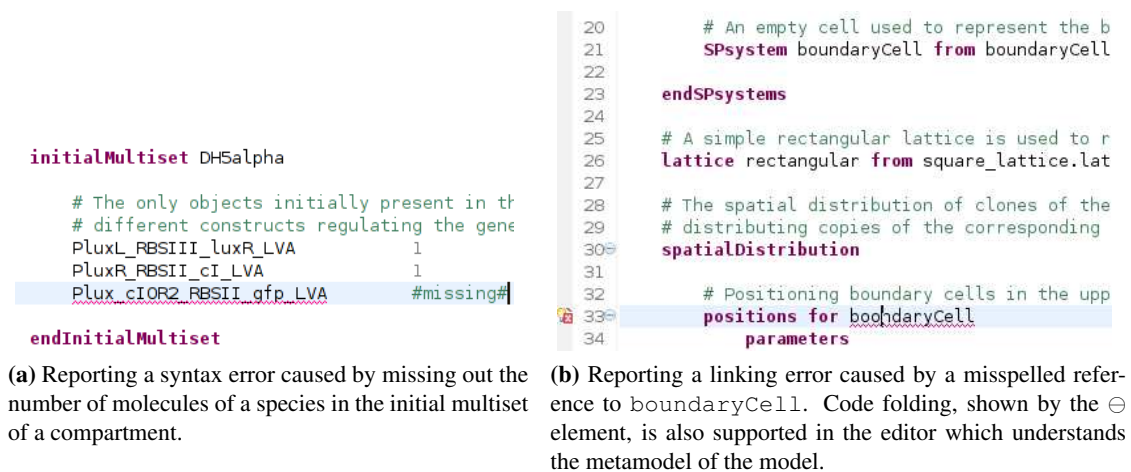


Figure 9.1: Screenshots of a prototype Xtext-based editor for Infobiotics DSLs, showing how the validation mechanism reports (a) syntactic errors when the input file does not conform to the grammar, and (b) linking errors when references are made to undefined elements.

it has failed again but for a different reason. The accretion of these bad experiences ultimately undermines confidence in the language and the software.

To create a first-class Infobiotics DSL editor we now are developing an Eclipse/Xtext-based editor, in the same vein as DNA Library Designer, which highlights syntax, validates semantics and reports errors at design time. Figure 9.1 demonstrates the reporting of syntactic and referencing errors in a prototype of this application.

With hindsight the Infobiotics Dashboard could have been developed from the ground up as an Eclipse RCP application as this offers similar input validation benefits as the present Traits framework (used for the parameterisation and results interfaces). In fact the Envisage and its Workbench plugin is a lighter weight clone of the Eclipse and its Workbench plugin. Eclipse’s project management capabilities, augmented with a distributed version control plugin such as MercurialEclipse can provide an excellent platform for iterative model development and collaboration that was unfeasible to replicate in Dashboard using Envisage components. On the other hand, Traits being Python and its validation mechanism being hooked directly into object attributes radically simplifies the development of executable workflows based on Infobiotics components as Python scripts, or even in the interactive interpreter like Matplotlib’s pylab and Mayavi’s mlab.

Fortunately, because the Infobiotics software was designed according to the Polyvalent-Program Pattern, a next-generation model editor that takes advantage of Xtext will not obsolete the existing implementations of either the Workbench’s experiments or their Dashboard interfaces. It is possible to edit the models within an Eclipse application, then by invoking a Simulate command, for example, have the main `ibw` executable open the appropriate experiment parameterisation GUI with the model parameter set automatically. Performing the experiment will

show its progress and when finished load the results interface in turn. Nevertheless, the overall look-and-feel of the application would be slightly compromised by the use of two GUI widget frameworks with differing layout and interaction rules: Traits command buttons being at the bottom of the dialog, the Dashboard interfaces floating outside the Eclipse window.

Our modelling approach is largely incompatible with other existing approaches and software tools precisely because it incorporates features that are not and cannot be supported within the current paradigm. Our software overcomes this deficit by integrating simulation, model checking and optimisation methods for working with these models, with a helpful user interface and informative visualisations. Nevertheless, forging connections between the Infobiotics Workbench and upcoming model repositories [259] will be important for our work to remain relevant; this would occur at the level of modules rather than models.

It must also be recognised that a large number of potentially important phenomena are not captured by our approach. Cells and multicellular systems have constantly changing morphologies, they grow and divide subject to biophysical forces, diversifying under evolutionary pressure. Fluctuating environmental temperature and pH can dramatically affect the molecular interactions. Mechanisms for modelling the physical interventions of researchers are also important. Neither do we address the problem of handling combinatorial complexes.

Over the course of this PhD research in systems and synthetic biology has boomed, launching a new wave of modular biomodel specification languages and tools with facilities for handling and validating genetic constructs in addition to reaction networks. These include Antimony [118], TinkerCell [260, 261], littleb [262, 117] Genetic Engineering of Cells (GEC) [263], proto [264] and GenoCAD [234, 265, 266]. They have developed in parallel to our own effort and each other with the result that the field is becoming increasingly fragmented. While by introducing yet another biological modelling approach we are to some extent contributing to this fragmentation, it can also be viewed as a very active area of research to which an all encompassing formulation has not been arrived at yet, and perhaps never will. As such this work represents a step forward that highlights some solutions but also the outstanding problems of multi-scale biological modelling, which we hope will challenge and guide other practitioners in the field that are developing increasingly comprehensive, modular and composable modelling formalisms.

9.4 Future directions

Expected uses of the fluent API provided by DNALD's datamodel are: import functions and export plugins that read and write established sequence and annotation file formats; constraint checkers incorporating secondary structure prediction algorithms such as those provided by the mFold and Vienna packages; additional visualizations of library designs; visual programming interfaces; an interactive interpreter.

The Synthetic Biology Open Language (SBOL), the recently published [267] standard for representing bioparts formalises the object model of a biopart as a simple recursive structure of annotated sequences within sequences that captures genes, parts, devices and even genomes. SBOL is gaining significant traction, as evidenced by its recent incorporation into Gene Designer, and is it therefore important to establish its relationship with DNALD. SBOL's datamodel inverts DNALD's datamodel, with sequence fragments derived from sequences rather than sequences derived from sequence fragments. It should be possible to interconvert between these in order to read and write SBOL with DNALD.

Linking Infobiotics models to DNALD (as a means to biomatter compilation) via the exported FASTA files for operon LPP modules is also straightforward task but would not exploit DNALD's unique selling point of combinatorial library generation. Adding set operations to our model definitions, in a manner analogous to DNALD, would enable families of related models to be defined in a compact form, explicitly capturing uncertainties or alternative mechanisms at exactly the junctions where those models diverge. Where those alternative submodels contain DNA sequences, a proper DNALD library can be extracted. This combinatorial modelling approach could also provide another mechanism for model structure optimisation: either the space of models could be well-defined by the modeller or the optimisation would produce this specification from parts libraries.

Summing up, the methods and tools we have introduced increase the scale of models, designs and libraries of biological systems that can be created by researchers. We have bridged the gap between abstract and concrete molecular networks, in single cells and in heterogeneous cellular populations. The benefits of this framework were demonstrated by its contribution to the understanding of several natural and synthetic biological systems. We believe that the utility of the DNALD language coupled with the high quality tooling that we are developing for it will drive uptake of users, and that with the involvement of the community in its ongoing design and improvement DNALD will become the *de facto* standard for design and communication of rationally designed combinatorial DNA libraries, synthesised by biotechnological platforms based on DNA reuse.

Bibliography

- [1] Ideker T, Galitski T, and Hood L. A New Approach to Decoding Life: Systems Biology. *Annu Rev Genomics Hum Genet*, 2:343, 2001
- [2] Kitano H. Computational systems biology. *Nature*, 420(6912):206, 2002
- [3] Alon U. *An Introduction to Systems Biology: Design Principles of Biological Circuits*. 1st edn. Chapman & Hall/CRC, 2006
- [4] Alon U. Network motifs: theory and experimental approaches. *Nature reviews Genetics*, 8(6):450, 2007
- [5] Moya A, Krasnogor N, Peretó J, and Latorre A. Goethe's dream. Challenges and opportunities for synthetic biology. *EMBO reports*, 10 Suppl 1:S28, 2009
- [6] Gibson DG, Glass JJ, Lartigue C, Noskov VN, *et al*. Creation of a Bacterial Cell Controlled by a Chemically Synthesized Genome. *Science*, 329(5987):52, 2010
- [7] Deamer D. A giant step towards artificial life? *Trends in Biotechnology*, 23(7):336, 2005
- [8] Hartwell LH, Hopfield JJ, Leibler S, and Murray AW. From molecular to modular cell biology. *Nature*, 402:C47, 1999
- [9] Porcar M, Danchin A, Lorenzo V, dos Santos Va, *et al*. The ten grand challenges of synthetic life. *Systems and Synthetic Biology*, 5(1):1, 2011
- [10] Smaldon J. Modelling Tools and Methodologies for Rapid Protocell Prototyping. Ph.D. thesis, University of Nottingham, 2010
- [11] Twycross J, Band LR, Bennett MJ, King JR, *et al*. Stochastic and deterministic multiscale models for systems biology: an auxin-transport case study. *BMC Systems Biology*, 4:34, 2010
- [12] Dupeux F, Santiago J, Betz K, Twycross J, *et al*. A thermodynamic switch modulates abscisic acid receptor sensitivity. *The EMBO journal*, (May):1, 2011
- [13] Diggle SP, Crusz SA, and Cámara M. Quorum sensing. *Current Biology*, 17(21):R907, 2007
- [14] Rampioni G, Pustelny C, Fletcher MP, Wright VJ, *et al*. Transcriptomic analysis reveals a global alkyl-quinolone-independent regulatory role for PqsE in facilitating the environmental adaptation of *Pseudomonas aeruginosa* to plant and animal hosts. *Environmental microbiology*, 12(6):1659, 2010
- [15] Garcia-Ojalvo J, Elowitz MB, and Strogatz SH. Modeling a synthetic multicellular clock: Repressilators coupled by quorum sensing. *Proc Natl Acad Sci*, 101(30):10955, 2004
- [16] Elowitz MB and Leibler S. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335, 2000
- [17] Basu S, Gerchman Y, Collins CH, Arnold FH, *et al*. A synthetic multicellular system for programmed pattern formation. *Nature*, 434(7037):1130, 2005
- [18] Davidson EH. *Gene Regulatory Systems: In Development and Evolution*. Academic Press, 2001
- [19] Misirli G, Hallinan JS, Yu T, Lawson JR, *et al*. Model Annotation for Synthetic Biology: Automating Model to Nucleotide Sequence Conversion. *Bioinformatics*, 27(7):973, 2011
- [20] Ben Yehezkel T, Linshiz G, Buaron H, Kaplan S, *et al*. De novo DNA synthesis using single molecule PCR. *Nucleic acids research*, 36(17):e107, 2008
- [21] Shabi U, Kaplan S, Linshiz G, Ben Yehezkel T, *et al*. Processing DNA molecules as text. *Systems and Synthetic Biology*, 4:227, 2010
- [22] Linshiz G, Yehezkel TB, Kaplan S, Gronau I, *et al*. Recursive construction of perfect DNA molecules from imperfect oligonucleotides. *Molecular Systems Biology*, 4(191):191, 2008
- [23] Rodrigo G, Carrera J, and Jaramillo A. Computational design of synthetic regulatory networks from a genetic library to characterize the designability of dynamical behaviors. *Nucleic Acids Research*, 39(20), 2011
- [24] Feist AM and Palsson BO. The growing scope of applications of genome-scale metabolic reconstructions using *Escherichia coli*. *Nature Biotechnology*, 26:659, 2008
- [25] Romero-Campero FJ. P Systems, a Computational Modelling Framework for Systems Biology. Ph.D. thesis, University of Seville, 2007
- [26] Krasnogor N, Gheorghe M, Terrazas G, Diggle S, *et al*. An appealing computational mechanism drawn from bacterial quorum sensing. *Bulletin of the EATCS*, 85:135, 2005

- [27] Bernardini F, Gheorghe M, Krasnogor N, Muniyandi R, *et al.* On P Systems as a Modelling Tool for Biological Systems. *Membrane Computing*, 3850:114, 2006
- [28] Bianco L, Pescini D, Siepmann P, Krasnogor N, *et al.* Towards a P Systems Pseudomonas Quorum Sensing Model. *WMC 7 LNCS 4361*, Volume 436:197, 2006
- [29] Bernardini F, Gheorghe M, and Krasnogor N. Quorum sensing P systems. *Theoretical Computer Science*, 371(1-2):20, 2007
- [30] Romero-Campero FJ and Pérez-Jiménez MJ. Modelling gene expression control using P systems: The Lac Operon, a case study. *Biosystems*, 91(3):438, 2008
- [31] Pérez-Jiménez M and Romero-Campero F. P systems, a new computational modelling tool for systems biology. In *Transactions on Computational Systems Biology VI*, 176–197. Springer, 2006
- [32] Romero-Campero FJ and Pérez-Jiménez MJ. A Model of the Quorum Sensing System in *Vibrio fischeri* Using P Systems. *Artificial Life*, 14(1):95, 2008
- [33] Gillespie DT. Exact Stochastic Simulation of Coupled Chemical Reactions. *The Journal of Physical Chemistry*, 81(25):2340, 1977
- [34] Romero-Campero F, Gheorghe M, Bianco L, Pescini D, *et al.* Towards Probabilistic Model Checking on P Systems Using PRISM. In *Membrane Computing*, vol. 4361 of *LNCS*, 477–495. Springer Berlin Heidelberg, 2006
- [35] Kwiatkowska M, Norman G, and Parker D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, vol. 6806 of *LNCS*, 585–591. 2011
- [36] Knight T, Rettberg R, Chan LY, Endy D, *et al.* Idempotent Vector Design for the Standard Assembly of Biobricks. 2003
- [37] Shetty RP, Endy D, and Knight TF. Engineering BioBrick vectors from BioBrick parts. *Journal of Biological Engineering*, 2:5, 2008
- [38] Weiner N. *Cybernetics: or the Control and Communication in the Animal and the Machine: Or Control and Communication in the Animal and the Machine*. 2nd edn. MIT Press, 1948
- [39] Von Bertalanffy L. *General System Theory: Foundations, Development, Applications*. George Braziller Inc, 1968
- [40] Klipp E, Liebermeister W, Wierling C, Kowald A, *et al.* *Systems Biology: A Textbook*. Wiley, 2009
- [41] Herrgard MJ, Swainston N, Dobson P, Dunn WB, *et al.* A consensus yeast metabolic network reconstruction obtained from a community approach to systems biology. *Nature Biotechnology*, 26(10):1155, 2008
- [42] Harel D. A Grand Challenge for Computing: Full Reactive Modeling of a Multi-Cellular Animal. *Bulletin of the EATCS*, 81:226, 2003
- [43] Jones D. All systems go. *Nature Reviews Drug Discovery*, 7:278, 2008
- [44] Gibson DG, Benders GA, Andrews-Pfannkoch C, Denisova EA, *et al.* Complete Chemical Synthesis, Assembly, and Cloning of a *Mycoplasma genitalium* Genome. *Science*, 1151721—, 2008
- [45] Sole RV, Munteanu A, Rodriguez-Caso C, and Maca J. Synthetic protocell biology: from reproduction to computation. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 362(1486):1727, 2007
- [46] Fellermann H, Rasmussen S, Ziöck HJ, and Sole RV. Life Cycle of a Minimal Protocell - A Dissipative Particle Dynamics Study. *Artificial Life*, 13(4):319, 2007
- [47] Rodrigo G, Montagud A, Aparici A, Aroca MC, *et al.* Vanillin cell sensor. *IET Synth Biol*, 1(1-2):74, 2007
- [48] Rodrigo G, Carreral J, and Jaramillo A. ECOLITASTER: cellular biosensor. *BMC Systems Biology*, 1(Suppl 1):P38, 2007
- [49] Weinberger LS, Schaffer DV, and Arkin AP. Theoretical design of a gene therapy to prevent AIDS but not human immunodeficiency virus type 1 infection. *Journal of Virology*, 77(18):10028, 2003
- [50] Hodgkin A and Huxley A. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500, 1952
- [51] Noble D. *The Music of Life*. OUP, 2006
- [52] Di Ventura B, Lemerle C, Michalodimitrakis K, and Serranò L. From in vivo to in silico biology and back. *Nature*, 443:527, 2006
- [53] Hänggi P. Stochastic Resonance in Biology: How Noise Can Enhance Detection of Weak Signals and Help Improve Biological Information Processing. *CHEMPHYSICHEM*, 3:285, 2002
- [54] Field RJ and Noyes RM. Oscillations in Chemical Systems IV. Limit cycle behavior in a model of a real chemical reaction. *The Journal of Chemical Physics*, 60:1877, 1974
- [55] Gillespie DT. Stochastic simulation of chemical kinetics. *Annual review of physical chemistry*, 58:35, 2007
- [56] Gillespie D. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403, 1976
- [57] Gillespie DT. A rigorous derivation of the chemical master equation. *Physica A*, 188:404, 1992
- [58] Pahle J. Biochemical simulations: stochastic, approximate stochastic and hybrid approaches. *Briefings in Bioinformatics*, 10(1):53, 2009

- [59] Gibson M and Bruck J. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J Phys Chem A*, 104(9):1876, 2000
- [60] Manninen T, Makiraatikka E, Ylipaa A, Pettinen A, *et al.* Discrete stochastic simulation of cell signaling: comparison of computational tools. In *Engineering in Medicine and Biology Society, 2006. EMBS '06. 28th Annual International Conference of the IEEE*, 2013–2016. 2006
- [61] Cao Y, Petzold LR, Rathinam M, and Gillespie DT. The numerical stability of leaping methods for stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 121(24):12169, 2004
- [62] McCollum JM, Peterson GD, Cox CD, Simpson ML, *et al.* The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behaviour. *Comput Bio Chem*, 30:39, 2006
- [63] Cox CD, Peterson GD, Allen MS, Lancaster JM, *et al.* Analysis of Noise in Quorum Sensing. *OMICS*, 7(3):317, 2003
- [64] Li H and Petzold L. Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems. Tech. rep., Department of Computer Science, University of California, 2006
- [65] Slepoy A, Thompson AP, and Plimpton SJ. A constant-time kinetic Monte Carlo algorithm for simulation of large biochemical reaction networks. *The Journal of chemical physics*, 128(20):205101, 2008
- [66] Ramaswamy R, González-Segredo N, and Sbalzarini IF. A new class of highly efficient exact stochastic simulation algorithms for chemical reaction networks. *The Journal of Chemical Physics*, 130(24):244104, 2009
- [67] Ramaswamy R and Sbalzarini IF. Fast Exact Stochastic Simulation Algorithms Using Partial Propensities. *Theoretical Computer Science*, 1338–1341, 2010
- [68] Ramaswamy R and Sbalzarini IF. A partial-propensity variant of the composition-rejection stochastic simulation algorithm for chemical reaction networks. *The Journal of Chemical Physics*, 132(4):044102, 2010
- [69] Ramaswamy R and Sbalzarini IF. A partial-propensity formulation of the stochastic simulation algorithm for chemical reaction networks with delays. *The Journal of chemical physics*, 134(1):14106, 2011
- [70] Li H, Cao Y, Petzold LR, and Gillespie DT. Algorithms and software for stochastic simulation of biochemical reacting systems. *Biotechnology Progress*, 24(1):56, 2008
- [71] Gillespie DT. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 115(4):1716, 2001
- [72] Cao Y and Petzold L. Trapezoidal tau-leaping formula for the stochastic simulation of biochemical systems. In *Proc. Found. Syst. Biol. Eng. (FOSBE 2005)*. 2005
- [73] Cao Y, Gillespie DT, and Petzold LR. Avoiding negative populations in explicit Poisson tau-leaping. *The Journal of chemical physics*, 123(5):54104, 2005
- [74] Cao Y, Gillespie DT, and Petzold LR. Efficient step size selection for the tau-leaping simulation method. *The Journal of chemical physics*, 124(4):44109, 2006
- [75] Cao Y, Gillespie DT, and Petzold LR. Adaptive explicit-implicit tau-leaping method with automatic tau selection. *The Journal of Chemical Physics*, 126(22):224101, 2007
- [76] Cao Y, Gillespie DT, and Petzold LR. The slow-scale stochastic simulation algorithm. *The Journal of Chemical Physics*, 122(1):14116, 2005
- [77] Cao Y, Gillespie DT, and Petzold LR. Accelerated stochastic simulation of the stiff enzyme-substrate reaction. *The Journal of Chemical Physics*, 123(14):144917, 2005
- [78] Cao Y and Petzold L. Slow Scale Tau-leaping Method. *Comput Methods Appl Mech Eng*, 197:43, 2008
- [79] Puchalka J and Kierzek AM. Bridging the gap between stochastic and deterministic regimes in the kinetic simulations of the biochemical reaction networks. *Biophysical Journal*, 86:1357, 2004
- [80] Munsky B and Khammash M. The finite state projection algorithm for the solution of the chemical master equation. *The Journal of chemical physics*, 124(4):44104, 2006
- [81] Drawert B, Lawson MJ, Petzold L, and Khammash M. The diffusive finite state projection algorithm for efficient simulation of the stochastic reaction-diffusion master equation. *The Journal of chemical physics*, 132(7):74101, 2010
- [82] Dematté L and Prandi D. GPU computing for systems biology. *Briefings in bioinformatics*, 11(3):323, 2010
- [83] Petzold L. Efficient Parallelization of the Stochastic Simulation Algorithm for Chemically Reacting Systems On the Graphics Processing Unit. *International Journal of High Performance Computing Applications*, 24(2):107, 2009
- [84] Klingbeil G, Erban R, Giles M, and Maini PK. STOCHSIMGPU : Parallel stochastic simulation for the Systems Biology Toolbox 2 for MATLAB. *Bioinformatics*, 2010
- [85] Dittamo C and Cangelosi D. Optimized Parallel Implementation of Gillespie's First Reaction Method on Graphics Processing Units. In *2009 International Conference on Computer Modeling and Simulation*, vol. 1, 156–161. Ieee, 2009
- [86] Salwinski L and Eisenberg D. In silico simulation of biological network dynamics. *Nature Biotechnology*, 22:1017, 2004
- [87] Lok L. The need for speed in stochastic simulation. *Nature Biotechnology*, 22:964, 2004

- [88] Hazapis OG and Manolakos ES. Scalable FRM-SSA SoC Design for the Simulation of Networks with Thousands of Biochemical Reactions in Real Time. In *2011 21st International Conference on Field Programmable Logic and Applications*, 459–463. Ieee, 2011
- [89] Cazzaniga P, Pescini D, Romero-campero FJ, Besozzi D, *et al.* Stochastic Approaches in P Systems for Simulating Biological Systems. In *Proceedings of the Fourth Brainstorming Week on Membrane Computing*, vol. 1 (edited by Gutierrez-Naranjo MA, Paun G, Riscos-Nuez A, and Romero-Campero FJ), 145–165. Sevilla, 2006
- [90] Cazzaniga P, Pescini D, Besozzi D, and Mauri G. Tau Leaping Stochastic Simulation Method in P Systems. In *Proc of the 7th International Workshop on Membrane Computing*, vol. 4361 (edited by Hoogeboom HJ), 298–313. LNCS, 2006
- [91] Lu T, Volfson D, Tsimring L, and Hasty J. Cellular growth and division in the Gillespie algorithm. *Systems Biology, IEE Proceedings*, 121–128, 2004
- [92] Lecca P. A time-dependent extension of gillespie algorithm for biochemical stochastic pi-calculus. *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, 137–144, 2006
- [93] Smaldon J, Blakes J, Krasnogor N, and Doron Lancet. A multi-scaled approach to artificial life simulation with P systems and dissipative particle dynamics. In *GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA*, 249–256. 2008
- [94] Versari C and Busi N. Stochastic Simulation of Biological Systems with Dynamical Compartment Structure. *Computational Methods in Systems Biology*, 4695:80, 2007
- [95] Versari C and Busi N. Efficient Stochastic Simulation of Biological Systems with Multiple Variable Volumes. *Electronic Notes in Theoretical Computer Science*, 194(3):165, 2008
- [96] Versari C. Stochastic modelling of cellular growth and division by means of the $\pi@$ calculus. In *Formal Methods in Molecular Biology, Dagstuhl Seminar Proceedings*, 1–15. 2009
- [97] Elf J and Ehrenberg M. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Systems Biology*, 1(2):230, 2004
- [98] Hattne J, Fange D, and Elf J. Stochastic reaction-diffusion simulation with MesoRD. *Bioinformatics*, 21:2923, 2005
- [99] Ander M, Beltrao P, Di Ventura B, Ferkinghoff-Borg J, *et al.* SmartCell, a framework to simulate cellular processes that combines stochastic approximation with diffusion and localisation: analysis of simple networks. *Systems Biology*, 1(1), 2004
- [100] McAdams HH and Arkin AP. Stochastic mechanisms in gene expression. *Proc Natl Acad Sci USA*, 94:814, 1997
- [101] McAdams HH and Arkin A. Simulation of prokaryotic genetic circuits. *Annual Review of Biophysics and Biomolecular Structure*, 27(1):199, 1998
- [102] Romero-Campero FJ, Twycross J, Cao H, Blakes J, *et al.* A Multiscale Modeling Framework Based on P Systems. In *WMC9 2008*, 63–77. Springer-Verlag Berlin, 2009
- [103] Novák B and Tyson JJ. Design principles of biochemical oscillators. *Nature reviews Molecular cell biology*, 9(12):981, 2008
- [104] Kwiatkowska M, Norman G, Parker D, and O. Simulation and verification for computational modelling of signalling pathways. In *Proceedings of the 2006 Winter Simulation Conference* (edited by Perrone LF, Wieland FP, Liu J, Lawson BG, *et al.*), 1666–1674. 2006
- [105] Kwiatkowska M, Norman G, and Parker D. Using probabilistic model checking in systems biology. *ACM SIGMETRICS Performance Evaluation Review*, 35(4):14, 2008
- [106] Kwiatkowska M, Norman G, and Parker D. Probabilistic Model Checking for Systems Biology. In *Symbolic Systems Biology* (edited by Iyengar MS), 31–59. 2010
- [107] Ananiadou S, Kell DBB, and Tsujii JII. Text mining and its potential applications in systems biology. *Trends in Biotechnology*, 24(12):571, 2006
- [108] Goldberg DE. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Welsey, 1989
- [109] Krasnogor N and Smith J. MAFRA: A Java Memetic Algorithms Framework. In *Proceedings of the 2000 International Genetic and Evolutionary Computation Conference (GECCO 2000)*. The Rivera Hotel and Casino, Las Vegas, Nevada, USA, 2000
- [110] Storn R and Price K. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *J Global Optim*, 11:341, 1997
- [111] Hansen N. The CMA Evolution Strategy: A Comparing Review. In *Towards a new evolutionary computation. Advances in estimation of distribution algorithms* (edited by Lozano JA, Larranaga P, Inza I, and Bengoetxea E), 75–102. Springer, 2006
- [112] Tomshine J and Kaznessis YN. Optimization of a Stochastically Simulated Gene Network Model via Simulated Annealing. *Biophysical Journal*, 91:3196, 2006
- [113] Smaldon J and Freitas Aa. A new version of the ant-miner algorithm discovering unordered rule sets. *Proceedings of the 8th annual conference on Genetic and evolutionary computation - GECCO '06*, 43, 2006

- [114] Iqbal M, Freitas AA, and Johnson CG. Protein Interaction Inference Using Particle Swarm Optimization Algorithm. In *EvoBIO'08 Proceedings of the 6th European conference on Evolutionary computation, machine learning and data mining in bioinformatics*, 61–70. 2008
- [115] Banga J. Optimization in computational systems biology. *BMC Systems Biology*, 2:47+, 2008
- [116] Koza JR, Mydlowec W, Lanza G, Yu J, *et al.* Reverse engineering of metabolic pathways from observed data using genetic programming. In *Pacific Symposium on Biocomputing*, 434–445. 2001
- [117] Mallavarapu A, Thomson M, Ullian B, and Gunawardena J. Programming with models: modularity and abstraction provide powerful capabilities for systems biology. *J R Soc Interface*, 6:257, 2009
- [118] Smith LP, Bergmann FT, Chandran D, and Sauro HM. Antimony: a modular model definition language. *Bioinformatics*, 25(18):2452, 2009
- [119] Fisher J and Henzinger TA. Executable cell biology. *Nature Biotechnology*, 25(11):1239, 2007
- [120] Fisher J and Henzinger T. Executable Biology. In *Proceedings of the 2006 Winter Simulation Conference* (edited by Perrone LF, Wieland FP, Liu J, Lawson BG, *et al.*), 1675–1682. 2006
- [121] Priami C. Algorithmic systems biology. *Communications of the ACM*, 52(5):80, 2009
- [122] Priami C, Regev A, Shapiro E, and Silverman W. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25, 2001
- [123] Heiner M, Lehrack S, Gilbert D, and Marwan W. Extended Stochastic Petri Nets for Model-Based Design of Wetlab Experiments. In *Trans. on Comput. Syst. Biol. XI, LNBI 5750*, 138–163. Springer-Verlag, 2009
- [124] Dematte L, Priami C, and Romanel A. Modelling and simulation of biological processes in BlenX. *ACM Sigmetrics Performance Evaluation Review*, 35(4):32, 2008
- [125] Kuttler C, Lhoussaine C, and Nebut M. Rule-based Modeling of Transcriptional Attenuation at the Tryptophan Operon. In *Formal Methods in Molecular Biology, Dagstuhl Seminar Proceedings*, 1–22. 2009
- [126] Hucka M, Finney a, Sauro HM, Bolouri H, *et al.* The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524, 2003
- [127] Hucka M, Bergmann F, Hoops S, Keating S, *et al.* The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core (Release 1 Candidate). *Nature Precedings*, 2010
- [128] Le Novère N, Bornstein B, Broicher A, Courtot M, *et al.* BioModels Database: a free, centralized database of curated, published, quantitative kinetic models of biochemical and cellular systems. *Nucleic Acids Research*, 34(Database issue):D689, 2006
- [129] Kauffman SA. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22:437, 1969
- [130] Kauffman SA. *The Origins of Order: Self-Organization and Selection in Evolution*. OUP USA, 1993
- [131] Feiglin A, Hacohen A, Sarusi A, Fisher J, *et al.* Static network structure can be used to model the phenotypic effects of perturbations in regulatory networks. *Bioinformatics (Oxford, England)*, 28(21):2811, 2012
- [132] Heiner M, Koch I, and Will J. Model validation of biological pathways using Petri nets—demonstrated for apoptosis. *BioSystems*, 75:15, 2004
- [133] Reddy VN, Mavrovouniotis ML, and Liebman MN. Petri net representations in metabolic pathways. In *Proc Int Conf Intell Syst Mol Biol*, 328–336. 1993
- [134] Reddy VN, Liebman MN, and Mavrovouniotis ML. Qualitative analysis of biochemical reaction systems. *Comput Biol Med*, 26:9, 1996
- [135] Chaouiya C. Petri net modelling of biological networks. *Briefings in Bioinformatics*, 8:210, 2007
- [136] Will J and Heiner M. Petri nets in Biology, Chemistry, and Medicine. Bibliography. Tech. rep., Brandenburg University of Technology at Cottbus, 2002
- [137] Gilbert D, Heiner M, and Lehrack S. A unifying framework for modelling and analysing biochemical pathways using Petri nets. In *Proceedings of the 2007 international conference on Computational methods in systems biology*, CMSB'07, 200–216. Springer-Verlag, Berlin, Heidelberg, 2007
- [138] Gilbert D, Fuss H, Gu X, Orton R, *et al.* Computational methodologies for modelling, analysis and simulation of signalling networks. *Briefings in Bioinformatics*, 7(4):339, 2006
- [139] Heiner M, Gilbert D, and Donaldson R. Petri Nets for Systems and Synthetic Biology. *Formal Methods for Computational Systems Biology*, 5016:215, 2008
- [140] Koch I, Junker BH, and Heiner M. Application of Petri net theory for modelling and validation of the sucrose breakdown pathway in the potato tuber. *Bioinformatics*, 21:1219, 2005
- [141] Goss PJE and Peccoud J. Quantitative modelling of stochastic systems in molecular biology by using stochastic Petri nets. *Proc Natl Acad Sci USA*, 95:6750, 1998
- [142] Milner R. *Communicating and Mobile Systems: the π -Calculus*. Cambridge University Press, Cambridge, 1999
- [143] Milner R. *A Calculus of Communicating Systems*. Springer Verlag, 1980
- [144] Hoare CAR. Communicating sequential processes. *Communications of the ACM*, 21(8):666, 1978
- [145] Regev A and Shapiro E. Cells as computation. *Nature*, 419:343, 2002
- [146] Regev A and Shapiro E. The π -calculus as an abstraction for biomolecular systems. In *Modelling in Molecular*

- Biology* (edited by Ciobanu G and Rozenberg G), 219–266. Springer, 2004
- [147] Regev A, Silverman W, and Shapiro E. Representing biomolecular processes with computer process algebra: pi-calculus programs of signal transduction pathways. *American Association for Artificial Intelligence Publication*, 2000
 - [148] Priami C and Quaglia P. Modelling the dynamics of biosystems. *Briefings in Bioinformatics*, 5(3):259, 2004
 - [149] Prandi D, Priami C, and Quaglia P. Process calculi in a biological context. *Bulletin of the EATCS*, 85:52, 2005
 - [150] Priami C. Process Calculi and Life Science. *Electronic Notes in Theoretical Computer Science*, 162:301, 2006
 - [151] Priami C. Computational Thinking in Biology. In *Transactions on Computational Systems Biology VIII*, 63–76. 2007
 - [152] Priami C. Stochastic π -calculus. *Computer Journal*, 38(7):578, 1995
 - [153] Regev A, Silverman W, and Shapiro E. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pac Symp Biocomput 2001*, vol. 26, 459–470. 2001
 - [154] Phillips A and Cardelli L. A Correct Abstract Machine for the Stochastic Pi-calculus. In *Concurrent Models in Molecular Biology, BioConcur '04*, ENTCS. 2004
 - [155] Phillips A and Cardelli L. Efficient, Correct Simulation of Biological Processes in the Stochastic Pi-calculus. In *Computational Methods in Systems Biology, LNCS 4695*, LNCS, 184–199. Springer, 2007
 - [156] Phillips A and Cardelli L. A Graphical Representation for the Stochastic Pi-calculus. In *Concurrent Models in Molecular Biology*. 2005
 - [157] Phillips A, Cardelli L, and Castagna G. A Graphical Representation for Biological Processes in the Stochastic Pi-calculus. *Transactions in Computational Systems Biology*, 4230:123, 2006
 - [158] Versari C. A Core Calculus for a Comparative Analysis of Bio-inspired Calculi. *Programming Languages and Systems*, 4421:411, 2007
 - [159] Regev A, Panina EM, Silverman W, Cardelli L, *et al.* BioAmbients: an abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141, 2004
 - [160] Cardelli L and Gordon AD. Mobile ambients. *Theoretical Computer Science*, 240(1):177, 2000
 - [161] Cardelli L. Brane Calculi. Interactions of Biological Membranes. Tech. rep., Microsoft Research, 2004
 - [162] Cardelli L. Bitonal membrane systems: Interactions of biological membranes. *Theoretical Computer Science*, 404(1-2):5, 2008
 - [163] Priami C and Quaglia P. *Computational Methods in Systems Biology*, vol. 3082 of *Lecture Notes in Computer Science*, chap. Beta Binde, 20–33. Springer Berlin / Heidelberg, 2005
 - [164] Dematté L, Priami C, and Romanel A. The BlenX Language: A Tutorial. In *Formal Methods for Computational Systems Biology, 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2008*, no. 5054 in *Lecture Notes in Computer Science*, 123–138. 2008
 - [165] Monod J, Wyman J, and Changeux JP. On the nature of allosteric transitions: a plausible model. *Journal of Molecular Biology*, 12:88, 1965
 - [166] Calder M, Gilmore S, and Hillston J. Automatically deriving ODEs from process algebra models of signalling pathways. In *Proceedings of Computational Methods in Systems Biology (CMSB 2005)* (edited by Plotkin G), 204–215. Edinburgh, Scotland, 2005
 - [167] Calder M, Gilmore S, Hillston J, and Vyshemirsky V. Formal methods for biochemical signalling pathways. In *Formal Methods: State of the Art and New Directions*, 185–215. Springer, 2006
 - [168] Calder M, Duguid A, Gilmore S, and Hillston J. Stronger computational modelling of signalling pathways using both continuous and discrete-state methods. In *Proceedings of the Fourth International Conference on Computational Methods in Systems Biology (CMSB 2006)*, vol. 4210 of *Lecture Notes in Computer Science* (edited by Priami C), 63–77. 2006
 - [169] Gerosa L. Stochastic process algebras as design and analysis framework for synthetic biology modelling. Master's thesis, University of Trento, 2007
 - [170] Calder M and Hillston J. *Process algebra modelling styles for biomolecular processes*, chap. Modelling, 1–25. No. 4230 in LNCS. Springer, 2009
 - [171] Ciocchetta F and Hillston J. Bio-PEPA: A framework for the modelling and analysis of biological systems. *Theoretical Computer Science*, 410(33-34):3065, 2009
 - [172] Ciocchetta F, Gilmore S, Guerriero ML, and Hillston J. Integrated Simulation and Model-Checking for the Analysis of Biochemical Systems
 - [173] Stenico M. Modelling molecular systems with discrete concentration levels in the context of the process algebra PEPA: Stochastic and deterministic interpretations. Corso di laurea in informatica specialistica, Università degli Studi di Trento, 2006
 - [174] Danos V and Laneve C. Formal Molecular Biology. *Theoretical Computer Science*, 325:69, 2004
 - [175] Hlavacek WS, Faeder JR, Blinov ML, Posner RG, *et al.* Rules for Modeling Signal-Transduction Systems. *Sci STKE*, 2006(344):1, 2006

- [176] Fontana W and Buss LW. *Boundaries and Barriers: On the Limits of Scientific Knowledge*, chap. The barrier. Perseus Books, U.S., 1999
- [177] Danos V, Feret J, Fontana W, and Krivine J. *Programming Languages and Systems*, 139–157. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2007
- [178] Danos V, Feret J, Fontana W, Harmer R, *et al.* Rule-based modelling of cellular signalling. In *CONCUR 2007, LNCS 4703* (edited by Caires L and Vasconcelos V), 17–41. Springer, 2007
- [179] Lok L and Brent R. Automatic generation of cellular reaction networks with Molecuizer 1.0. *Nature Biotechnology*, 23:131, 2005
- [180] Lis M, Artyomov MN, Devadas S, and Chakraborty AK. Efficient stochastic simulation of reaction-diffusion processes via direct compilation. *Bioinformatics (Oxford, England)*, 25(17):2289, 2009
- [181] Păun G. Computing with Membranes. *Journal of Computer and System Sciences*, 61 (2000)(1):108, 2000
- [182] Gutierrez-Naranjo MA, Perez-Jimenez MJ, and Romero-Campero FJ. A uniform solution to SAT using membrane creation. *Theoretical Computer Science*, 371:54, 2007
- [183] Gheorghe M, Krasnogor N, and Camara M. P systems applications to systems biology. *BioSystems*, 91(3):435, 2008
- [184] Păun G and Romero-Campero FJ. Membrane Computing as a Modeling Framework. Cellular Systems Case Studies. In *Formal Methods for Computational Systems Biology, LNCS 5016*, Lecture Notes in Computer Science, 168–214. 2008
- [185] Păun G. *Membrane Computing: An Introduction*. Springer-Verlag Berlin Heidelberg, 2002
- [186] Păun G. Introduction to Membrane Computing. In *Applications of Membrane Computing*, 1–42. 2006
- [187] Gheorghe M, Manca V, and Romero-Campero FJ. Deterministic and stochastic P systems for modelling cellular processes. *Natural Computing*, 9(2):457, 2009
- [188] Păun A and Păun G. The Power of Communication: P Systems with Symport/Antiport. *New Generation Computing*, 20:295, 2002
- [189] Păun G. P Systems with Active Membranes: Attacking NP-Complete Problems. *Journal of Automata, Languages and Combinatorics*, 6(1):75, 2001
- [190] Besozzi D, Zandron C, Mauri G, and Sabadini N. P Systems with Gemmation of Mobile Membranes. In *ICTCS '01 Proceedings of the 7th Italian Conference on Theoretical Computer Science*, 136–153. Springer-Verlag London, 2001
- [191] Bernardini F and Manca V. P Systems with Boundary Rules. In *WMC-CdeA 2002, LNCS 2597*, Lecture Notes in Computer Science, 107–118. Springer-Verlag Berlin Heidelberg, 2003
- [192] Martín-Vide C and Paun G. Tissue P systems. *Theoretical Computer Science*, 296(2):295, 2003
- [193] Bernardini F and Gheorghe M. Population P systems. *Journal of Universal Computer Science*, 10(5):509, 2004
- [194] Auld J, Bianco L, Ciobanu G, Gheorghe M, *et al.* Population P systems a Model for the Behaviour of Systems of Bio-Entities. In *Workshop on Membrane Computing 7 - At the Crossroads of Cell Biology and Computation*, 15–19. 2006
- [195] Bernardini F, Gheorghe M, Romero-Campero FJ, and Walkinshaw N. A Hybrid Approach to Modeling Biological Systems. In *Workshop on Membrane Computing*, 138–159. 2007
- [196] Kelemen J, Kelemenová A, and Păun G. On the Power of a Biochemically Inspired Simple Computing Model: P Colonies. *Workshop on Artificial Chemistry ALIFE09*, 82–86, 2004
- [197] Ionescu M, Păun G, and Yokomori T. Spiking Neural P Systems. *Fundamenta Informaticae*, 71(2-3):279, 2006
- [198] Terrazas G, Krasnogor N, Gheorghe M, Bernardini F, *et al.* An environment aware P-system model of quorum sensing. In *Computability in Europe, LNCS 3526* (edited by Cooper B, Lowe B, and Torenvliet L), 479–485. Springer-Verlag Berlin Heidelberg, 2005
- [199] Bernardini F, Romero-Campero FJ, Gheorghe M, Pérez-Jiménez MJ, *et al.* On P Systems with Bounded Parallelism. In *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, 399–406. 2005
- [200] Bernardini F, Romero-Campero FJ, Gheorghe M, and Pérez-Jiménez MJ. A Modeling Approach Based on P Systems with Bounded Parallelism. In *Workshop on Membrane Computing* (edited by Hoogeboom HJ), 49–65. Springer-Verlag Berlin Heidelberg, 2006
- [201] Ciobanu G, Pan L, Păun G, and Pérez-Jiménez MJ. P systems with minimal parallelism. *Theoretical Computer Science*, 378:117, 2007
- [202] Bianco L and Castellini A. Psim: A Computational Platform for Metabolic P systems. In *Workshop on Membrane Computing*, 1–20. 2007
- [203] P System Modelling Framework. http://www.dcs.shef.ac.uk/~marian/PSimulatorWeb/P_Systems_applications.htm
- [204] Romero-Campero FJ, Twycross J, Bennett M, Camara M, *et al.* Modular Assembly of Cell Systems Biology Models Using P systems. In *International Journal of Foundations of Computer Science*

- [205] Wilkinson DJ. The Discrete Stochastic Models Test Suite. <http://code.google.com/p/dsmts/>
- [206] Cavaliere M and Sedwards S. Modelling Cellular Processes Using Membrane Systems with Peripheral and Integral Proteins. In *Computational Methods in Systems Biology (CMSB 2006)*, LNCS 4210 (edited by Priami C), 108–126. Springer, 2006
- [207] Cavaliere M and Sedwards S. Membrane Systems with Peripheral Proteins: Transport and Evolution. *Electr Notes Theor Comput Sci*, 171(2):37, 2007
- [208] Sedwards S and Mazza T. Cyto-Sim: a formal language model and stochastic simulator of membrane-enclosed biochemical processes. *Bioinformatics*, 23:2800, 2007
- [209] Cyto-Sim biochemistry simulator. http://www.cosbi.eu/Rpty_Soft_CytoSim.php
- [210] Mazza T. Towards a Complete Covering of SBML Functionalities. In *Workshop on Membrane Computing 8 (WMC8)*, 425–444, 2007
- [211] Pescini D, Besozzi D, Mauri G, and Zandron C. Dynamic probabilistic P systems. *International Journal of Foundations of Computer Science*, 1(17):183, 2006
- [212] Besozzi D, Cazzaniga P, Pescini D, and Mauri G. Modelling metapopulations with stochastic membrane systems. *BioSystems*, 91(3):499, 2008
- [213] Mauri G. *Membrane Systems and Their Application to Systems Biology*, 551–553. Springer Berlin / Heidelberg, 2007
- [214] Bianco L. Membrane Models of Biological Systems. Ph.D. thesis, Università degli Studi di Verona Dipartimento di Informatica, Università di Verona Dipartimento di Informatica Strada le Grazie 15, 37134 Verona Italy, 2007
- [215] Palsson BO. *Systems Biology: Properties of Reconstructed Networks*. 1st edn. Cambridge University Press, 2006
- [216] Bianco L and Manca V. Encoding-Decoding Transitional Systems for Classes of P Systems. In *WMC 2005*, LNCS 3850 (edited by Freund R), LNCS, 134–143, 2005
- [217] Castellini A, Manca V, and Suzuki Y. Metabolic P System Flux Regulation by Artificial Neural Networks. 2009
- [218] MetaPlab
- [219] Giavitto JL and Michel O. MGS: A Rule-Based Programming Language for Complex Objects and Collections. *Electronic Notes in Theoretical Computer Science*, 59(4):286, 2001
- [220] Maus C, Rybacki S, and Uhrmacher AM. Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology*, 5(1):166, 2011
- [221] Spicher A, Michel O, Cieslak M, Giavitto JL, *et al.* Stochastic P systems and the simulation of biochemical processes with dynamic compartments. *Bio Systems*, 91(3):458, 2008
- [222] Gheorghe M and Krasnogor N. P-systems and X-machines. *Natural Computing*, 8(4):777, 2009
- [223] Alur R and Henzinger T. Reactive modules. *Formal Methods in System Design*, 15:7, 1999
- [224] Donaldson R and Gilbert D. A Monte Carlo Model Checker for Probabilistic LTL with Numerical Constraints. Tech. rep., Bioinformatics Research Centre, University of Glasgow, Glasgow, 2008
- [225] Romero-campero FJ, Righetti K, Blakes J, Heeb S, *et al.* Pattern Formation in Synthetic Bacterial Colonies. 2010
- [226] Gillespie DT. Concerning the validity of the stochastic approach to chemical kinetics. *Journal of Statistical Physics*, 16(3):311, 1977
- [227] Lagarias JC. Point lattices. *Handbook of Combinatorics*, 1, 1996
- [228] Canton B, Labno A, and Endy D. Refinement and standardization of synthetic biological parts and devices. *Nature Biotechnology*, 26(7):787, 2008
- [229] Hucka M, Finney A, Bornstein J, Keating M, *et al.* Evolving a lingua franca and associated software infrastructure for computational systems biology: the Systems Biology Markup Language (SBML) project. *Systems Biology*, 2004
- [230] Funahashi A, Matsuoka Y, Jouraku A, Morohashi M, *et al.* CellDesigner 3.5: A Versatile Modeling Tool for Biochemical Networks. *Proceedings of the IEEE*, 96(8):1254, 2008
- [231] Le Novère N, Hucka M, Mi H, Moodie S, *et al.* The Systems Biology Graphical Notation. *Nature Biotechnology*, 27(8):735, 2009
- [232] van Iersel MP, Villéger AC, Czauderna T, Boyd SE, *et al.* Software support for SBGN maps: SBGN-ML and LibSBGN. *Bioinformatics (Oxford, England)*, 1–6, 2012
- [233] Cuellar AA, Lloyd CM, Nielsen PF, Bullivant DP, *et al.* An Overview of CellML 1.1, a Biological Model Description Language. *Simulation*, 79(12):740, 2003
- [234] Cai Y, Hartnett B, Gustafsson C, and Peccoud J. A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. *Bioinformatics (Oxford, England)*, 23(20):2760, 2007
- [235] Blakes J, Twycross J, Romero-Campero FJ, and Krasnogor N. The Infobiotics Workbench: an integrated in silico modelling platform for Systems and Synthetic Biology. *Bioinformatics*, 27(23):3323, 2011
- [236] Hoops S, Sahle S, Gauges R, Lee C, *et al.* COPASI—a COMplex PATHway Simulator. *Bioinformatics*,

- 22(24):3067, 2006
- [237] Mendes P, Hoops S, Sahle S, Gauges R, *et al.* Computational modeling of biochemical networks using COPASI. *Methods In Molecular Biology*, 500:17, 2009
 - [238] Raymond ES. *The Art of Unix Programming*. Addison-Wesley, 2003
 - [239] Romero-Campero FJ, Twycross J, Cámara M, Bennett M, *et al.* Modular Assembly of Cell Systems Biology Models Using P Systems. *International Journal of Foundations of Computer Science*, 20(03):427, 2009
 - [240] Galassi M. *GNU Scientific Library Reference Manual*. 3rd edn. GNU, 2009
 - [241] Dougherty MT, Folk MJ, Zadok E, Bernstein HJ, *et al.* Unifying Biological Image Formats with HDF5. *ACM Queue Bioscience*, 7(9):1
 - [242] Oberhumer MFXJ. LZO real-time data compression library. <http://www.oberhumer.com/opensource/lzo/>
 - [243] Goryachev aB, Toh DJ, and Lee T. Systems analysis of a quorum sensing network: design constraints imposed by the functional requirements, network topology and kinetic constants. *BioSystems*, 83(2-3):178, 2006
 - [244] Petersson SV, Johansson AI, Kowalczyk M, Makoveychuk A, *et al.* An auxin gradient and maximum in the Arabidopsis root apex shown by high-resolution cell-specific analysis of IAA distribution and synthesis. *The Plant cell*, 21(6):1659, 2009
 - [245] Hunter JD. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90, 2007
 - [246] Ramachandran P and Varoquaux G. Mayavi: Making 3D data visualization reusable. In *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, SciPy, 51–56. 2008
 - [247] PRISM - Probabilistic Symbolic Model Checker. <http://www.prismmodelchecker.org/>
 - [248] Donaldson R. MC2(PLTLc) Monte Carlo Model Checker for PLTLc properties. <http://www.brc.dcs.gla.ac.uk/software/mc2/>
 - [249] Donaldson R and Gilbert D. A Model Checking Approach to the Parameter Estimation of Biochemical Pathways. In *CMSB 2008, LNBI 5307*, 269–287. Springer-Verlag, 2008
 - [250] Hansen N and Ostermeier A. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computing*, 9:159, 2001
 - [251] Romero-Campero FJ, Cao H, Camara M, and Krasnogor N. Structure and parameter estimation for cell systems biology models. *Proceedings of the 10th annual conference on Genetic and Evolutionary Computation (GECCO '08)*, 331–339, 2008
 - [252] Cao H, Romero-Campero FJ, Heeb S, Cámara M, *et al.* Evolving cell models for systems and synthetic biology. *Systems and synthetic biology*, 4(1):55, 2010
 - [253] Hansen LH, Bentzon-Tilia M, Bentzon-Tilia S, Norman A, *et al.* Design and Synthesis of a Quintessential Self-Transmissible IncX1 Plasmid, pX1.0. *PloS one*, 6(5):e19912, 2011
 - [254] Rothmund PWK. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297, 2006
 - [255] Farnham PJ and Platt T. Rho-independent termination: dyad symmetry in DNA causes RNA polymerase to pause during transcription in vitro. *Nucleic Acids Research*, 9(31981):563, 1980
 - [256] Zuker M. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Research*, 31(13):3406, 2003
 - [257] Tarjan RE. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171, 1976
 - [258] Spolsky J. *User Interface Design for Programmers*. Apress, 2001
 - [259] Cooling MT, Rouilly V, Misirli G, Lawson J, *et al.* Standard virtual biological parts: a repository of modular modeling components for synthetic biology. *Bioinformatics*, 26(7):925, 2010
 - [260] Chandran D, Bergmann FT, and Sauro HM. TinkerCell: modular CAD tool for synthetic biology. *Journal of Biological Engineering*, 3:19, 2009
 - [261] Chandran D, Bergmann FT, and Sauro HM. Computer-aided design of biological circuits using TinkerCell. *Bioengineered Bugs*, 1(4):274, 2010
 - [262] Mallavarapu A, Thomson M, Ullian B, and Gunawardena J. Modular model building. *arXiv:07103421v1*, 1–24, 2007. 0710.3421
 - [263] Pedersen M and Phillips A. Towards programming languages for genetic engineering of living cells, *Journal of the. Journal of The Royal Society Interface*, 375–378, 2009
 - [264] Beal J, Lu T, and Weiss R. Automatic Compilation from High-Level Biologically-Oriented Programming Language to Genetic Regulatory Networks. *PLoS ONE*, 6(8):e22490, 2011
 - [265] Cai Y, Lux MW, Adam L, and Peccoud J. Modeling Structure-Function Relationships in Synthetic DNA Sequences using Attribute Grammars. *PLoS Computational Biology*, 5(10):10, 2009
 - [266] Cai Y, Wilson ML, and Peccoud J. GenoCAD for iGEM: a grammatical approach to the design of standard-compliant constructs. *Nucleic Acids Research*, 38(8):2637, 2010
 - [267] Galdzicki M, Wilson ML, Rodriguez CA, Adam L, *et al.* Synthetic Biology Open Language (SBOL) Version 1.0.0. Tech. Rep. C, 2011