

Modular Languages for Systems and Synthetic Biology

Michael Pedersen



Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2010

Abstract

Systems biology is a rapidly growing field which seeks a refined quantitative understanding of organisms, particularly studying how molecular species such as metabolites, proteins and genes interact in cells to form the complex emerging behaviour exhibited by living systems. *Synthetic biology* is a related and emerging field which seeks to engineer new organisms for practical purposes. Both fields can benefit from formal languages for modelling, simulation and analysis.

In systems biology there is however a trade-off in the landscape of existing formal languages: some are modular but may be difficult for some biologists to understand (e.g. process calculi) while others are more intuitive but monolithic (e.g. rule-based languages). The first major contribution of this thesis is to bridge this gap with a *Language for Biochemical Systems* (LBS). LBS is based on the modular *Calculus of Biochemical Systems* and adds e.g. parameterised modules with subtyping and a notion of nondeterminism for handling combinatorial explosion. LBS can also incorporate other rule-based languages such as Kappa, hence adding modularity to these. Modularity is important for a rational structuring of models but can also be exploited in analysis as is shown for the specific case of Petri net flows.

On the synthetic biology side, none of the few existing dedicated languages allow for a high-level description of designs that can be automatically translated into DNA sequences for implementation in living cells. The second major contribution of this thesis is exactly such a language for *Genetic Engineering of Cells* (GEC). GEC exploits the recent advent of standard genetic parts (“biobricks”) and allows for the composition of such parts into genes in a modular and abstract manner using logical constraints. GEC programs can then be translated to DNA sequences using a constraint satisfaction engine based on a given database of genetic parts.

Acknowledgements

I thank Microsoft Research for its funding through the European PhD Scholarship Programme. I thank Gordon Plotkin for his patient supervision, and support in all things academic, over the past three years, and for his role in co-authoring two papers on LBS, one of which is incorporated into this thesis. I also thank Andrew Phillips for supervising me during a stimulating internship experience at Microsoft Research and for his role in co-authoring a paper on GEC which is incorporated into this thesis; many of the diagrams in Chapter 8 are entirely due to him. I thank Vincent Danos for his enthusiasm and many inspiring discussions during and after Gordon's sabbatical. I thank Nicolas Oury for useful discussions on LBS; Monika Heiner for useful discussions on Petri net flows; Stuart Moodie and Anatoly Sorokin for useful discussions on SBGN; William Chen for answering questions about the ErbB pathway model; and Jane Hillston and Luca Cardelli for useful feedback in their role as my thesis examiners.

I am grateful to all my friends and colleagues in the School of Informatics who have helped make everyday life as a PhD student more fun. I am grateful to my family for always supporting me in what I want to do, even when this puts distance between us. Finally, I am grateful to Ros Marvin for her constant support and for continuing to remind me of what is truly important in life.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Michael Pedersen)

Table of Contents

1	Introduction	1
1.1	Background	1
1.2	Contributions	3
1.3	Thesis outline	6
2	Existing Formalisms for Biology	9
2.1	Petri nets	12
2.1.1	Basic Petri nets	12
2.1.2	Coloured Petri Nets	13
2.2	Rule-based languages	15
2.2.1	BIOCHAM	15
2.2.2	BioNetGen	16
2.2.3	κ	17
2.3	Rule-based Languages with Modularity	18
2.3.1	Little b	18
2.3.2	Antimony	19
2.4	Languages for Synthetic Biology	20
2.4.1	GenoCad	20
2.4.2	Antimony	21
2.5	CBS by Example	21
2.5.1	Gene Expression	23
2.5.2	A MAPK Cascade	24
2.5.3	A Scaffolded MAPK Cascade.	26
3	LBS by Example	29
3.1	Gene Expression	29
3.1.1	New Species and Compartment Definitions	29

3.1.2	Parameterised Modules	30
3.2	A MAPK Cascade	31
3.2.1	Parametric Type and Subtyping	32
3.2.2	Nondeterminism	34
3.2.3	A Modification Site Type with Binding	40
3.2.4	Model Variation	43
3.3	A Scaffolded MAPK Cascade	44
3.3.1	Species Expressions	44
3.3.2	Output Species Parameters	46
3.4	Case Study: The Yeast Pheromone Pathway	47
3.4.1	Overview of the Pathway	47
3.4.2	The LBS Model	48
3.4.3	Model Validation	48
3.5	Case Study: The ErbB Pathway	49
3.5.1	Overview of the Pathway	49
3.5.2	The LBS Model	49
3.5.3	The Modelling Process and Model Validation	50
4	The Abstract Syntax of LBS	53
4.1	Notation	53
4.2	Compartments	54
4.2.1	Compartment Expressions	54
4.2.2	Derived Compartment Expressions	55
4.3	Species	56
4.3.1	Modification Site Expressions	56
4.3.2	Species Expressions	56
4.3.3	Derived Species Expressions	59
4.4	Programs	59
4.4.1	Basic Programs	59
4.4.2	Derived Programs	61
4.5	Definitions	62
4.5.1	Basic Definitions	62
4.5.2	Derived Definitions	62

5 The General Semantics of LBS	65
5.1 Compartments	69
5.1.1 Compartment Values	69
5.1.2 The Denotation Function	69
5.1.3 Well-Typedness of Compartment Value Lists	70
5.1.4 Normal Forms of Compartment Value Lists	71
5.2 Species	71
5.2.1 Species Values	71
5.2.2 Well-Typedness of Species Values	74
5.2.3 The Denotation Function	76
5.2.4 Normal Forms of Species Values and Further Functions	80
5.2.5 Species Value Design Choices	83
5.3 Programs	84
5.3.1 Normal Form Reactions	84
5.3.2 The Denotation Function for Basic Programs	86
5.3.3 The Definition of Derived Programs	90
5.4 Definitions	95
6 Some Concrete Semantics of LBS	99
6.1 Preliminaries	99
6.1.1 Ground Normal Form Reactions	100
6.1.2 The General Semantics in Terms of Ground Normal Form Reactions	101
6.2 A Basic Petri Net Semantics	102
6.2.1 Basic Petri Nets	102
6.2.2 The Qualitative Semantics of Basic Petri Nets	103
6.2.3 The Concrete Basic Petri Net Semantics of LBS	104
6.3 A Coloured Petri Net Semantics	105
6.3.1 Coloured Petri Nets	105
6.3.2 The Qualitative Semantics of Coloured Petri Nets	106
6.3.3 The Concrete Coloured Petri Net Semantics of LBS	107
6.4 An ODE Semantics	108
6.4.1 ODEs	109
6.4.2 The Concrete ODE Semantics of LBS	109
6.5 A CTMC Semantics	110

6.5.1	CTMCs	111
6.5.2	The Concrete CTMC Semantics of LBS	111
6.6	A κ Semantics	112
6.6.1	κ	113
6.6.2	The Concrete κ Semantics of LBS	115
7	Concrete Petri Net Flow Semantics of LBS	121
7.1	Preliminaries	122
7.1.1	Flow Matrices	122
7.1.2	Petri Net Flows	123
7.1.3	A Running Example: Photosynthesis and Respiration	124
7.1.4	Existing Results	126
7.2	Flow Matrix Composition and Modular Duality	127
7.2.1	Matrix-Based Composition With Place Sharing	127
7.2.2	Modular Duality: Composition With Transition Sharing	128
7.3	Modular Minimal T-Flows	129
7.3.1	The Intuition	129
7.3.2	The Definition	130
7.3.3	Results	131
7.4	Modular Minimal P-Flows	132
7.4.1	The Intuition	132
7.4.2	The Definition	133
7.4.3	Results	134
7.5	Concrete Semantics of LBS	135
7.5.1	The Concrete Minimal T-Flow Semantics of LBS	136
7.5.2	The Concrete Minimal P-Flow Semantics of LBS	138
7.5.3	Results	140
7.6	Related Work	140
8	GEC by Example	143
8.1	The Databases	145
8.1.1	The Reaction Database	145
8.1.2	The Parts Database	145
8.1.3	Reactions Associated with Parts	147
8.2	The Basics of GEC	149
8.2.1	Sequences of Typed Parts	149

8.2.2	Part Variables and Properties	149
8.2.3	Parameterised Modules	150
8.2.4	Compartments and Reactions	151
8.3	Case Study: The Repressilator	152
8.3.1	The GEC Model	152
8.3.2	Translation and Simulation	152
8.3.3	The Revised GEC Model	153
8.4	Case Study: The Predator-Prey System	155
8.4.1	The GEC Model	155
8.4.2	Translation and Simulation	158
9	The Abstract Syntax and Semantics of GEC	163
9.1	The Abstract Syntax of GEC	164
9.2	The Substitution Semantics of GEC	166
9.2.1	The Intuition	166
9.2.2	The Definition	169
9.2.3	Results	173
9.3	The Device Semantics of GEC	173
9.3.1	The Intuition	173
9.3.2	The Definition	174
9.4	The Reaction Semantics of GEC	175
9.4.1	The Intuition	175
9.4.2	The Definition	178
10	Conclusions	183
10.1	Evaluation	183
10.2	Future Work	184
10.2.1	LBS	184
10.2.2	GEC	185
A	The Yeast Pheromone Pathway in LBS	189
B	The ErbB Pathway in LBS	195
C	Proofs	223
C.1	Proofs for Compartment Value Lists	223
C.2	Proofs for Petri Net Flows	224

C.2.1	Duality	224
C.2.2	Modular T-Flows	224
C.2.3	Modular P-Flows	227
C.2.4	Proofs for Concrete Flow Semantics	231
C.3	Proofs for GEC	232
C.3.1	Injectivity	232
C.3.2	Non-interference	234
	Table of Notation	237
	Bibliography	245

Chapter 1

Introduction

1.1 Background

Systems and synthetic biology *Systems biology* [50] is a rapidly growing field which seeks a refined quantitative understanding of organisms, particularly studying how molecular species such as metabolites, proteins and genes interact in cells to form the complex emerging behaviour exhibited by living systems. Such an understanding is, for example, important for the discovery and development of new drugs and to predict the impact of these on an organism [12].

Synthetic biology [34] is a related emerging field which seeks to engineer new organisms for practical purposes. Promising prospects include for example bacteria that produce hydrogen from sunlight and water, thus addressing the problem of global warming, and bacteria that detect environmental pollutants in an economically viable manner, thus leading to improved quality of life in impoverished regions [36].

Despite having seemingly different aims, systems and synthetic biology are in fact highly complementary. The process of engineering new organisms sheds light onto how evolution may have shaped existing organisms, and the knowledge of how existing organisms function in turn yields building blocks which can be used in the engineering of new organisms.

Mathematical modelling Mathematical modelling plays a key role in systems biology where it facilitates the generation of new knowledge of existing biological systems through the cycle of simulation, experimental validation, and model refinement [50, 57]. In synthetic biology the design of a new system is likewise validated through mathematical modelling and simulation before its in-vivo implementation [33]. Math-

ematical modelling has traditionally been based on ordinary differential equations (ODEs) or variants thereof which can be simulated through numerical integration. When stochastic effects are important, other mathematical structures such as continuous time Markov chains (CTMCs) are employed.

Modelling languages As our biological knowledge-base increases through rapid improvements of e.g. high-throughput genome sequencing methods, the models under study in systems biology also increase in size and complexity. Synthetic systems are likewise increasing in size following improvements of e.g. gene synthesis techniques. New methods are therefore needed to support the structured development of large models, and also to complement simulations with appropriate analysis methods.

Hence an abundance of formal modelling languages and frameworks inspired by computer science have found their way to biological modelling over the past decade. These include Petri nets [65, 77] and coloured Petri nets [48, 55]; process calculi such as the π -calculus [78], the stochastic π -calculus [75, 9, 17], the continuous π -calculus [53], Beta binders [76, 41], BlenX [32], PEPA [45, 15] and BioPEPA [24, 25, 2]; rule-based languages such as κ [30, 29], BioNetGen [35] and BIOCHAM [20]; state-based formalisms such as Statecharts [42]; and languages such as Bioambients [79], the Brane calculi [16], P-systems [67] and Bigraphs [27, 26, 63] with specialised features for describing biological compartments and membranes.

Limitations of existing modelling languages Some of the above mentioned languages, in particular those from the process calculus family, support modularity by allowing large systems to be described in terms of their components. Modularity allows for more structured models which are easier to maintain and understand, and can potentially be exploited to obtain more efficient simulation and analysis methods. These languages may however be difficult for non-specialists, including some biologists, to use and to understand. Other languages, for example from the rule-based family, are more intuitive to use but only allow flat, non-modular descriptions.

Any language used for modelling in systems biology can in principle also be used to model a novel system in synthetic biology. In fact one may argue that modelling in synthetic biology poses certain advantages, e.g. that the modeller can decide on the modules, whereas the extent to which natural systems exhibit modularity is a subject of much debate. However, the above mentioned languages are limited by their lack of dedicated support for the modelling of genes, requiring ad hoc approaches which

may be overly complicated and do not match the domain particularly well. It is indeed widely recognised that new methods supporting a structured approach to the engineering of genes and genetic networks are needed [4].

1.2 Contributions

Aim The general aim of this thesis is to close the gap in the above landscape by developing formal languages for biology which:

1. allow one to write modular models of large cellular systems, and
2. allow one to write intuitively and concisely.

The specific contributions towards this aim are two languages. The first, entitled *a Language for Biochemical Systems* (LBS), lies in the realm of systems biology and allows existing biological systems found in nature to be modelled and subsequently analysed. The second, a language for *Genetic Engineering of Cells* (GEC), lies in the realm of synthetic biology and allows for the modelling of desired phenotypical characteristics of new systems and the subsequent simulation and translation to a number of possible genetic devices.

A language for biochemical systems LBS is based on the *Calculus of Biochemical Systems* (CBS) [74] which allows the modular modelling of cellular systems as biochemical reactions of complexes of modified species, taking place in parallel and inside a hierarchy of compartments. CBS has a formal compositional semantics, translating programs into semantical objects such as ODEs, CTMCs, Petri nets and coloured Petri nets. The first two allow continuous and stochastic simulations to be carried out, and Petri nets are supported by a large range of established analysis methods that are useful in the biological setting.

CBS forms a good starting point because it combines the intuition associated with standard chemical reactions with a notion of modularity, but it does not go all the way towards our aim. The notion of modularity is limited because there is no means of parameterisation. Conciseness also suffers for large models because of a frequent need for duplication of molecular complexes and because of limited support for handling the combinatorial complexity inherent in many signalling pathways.

LBS is an extension of CBS, designed to address these practical problems. It allows for module reuse through parameterisation and a notion of subtyping and parametric

type; it facilitates the handling of large molecular complexes through dedicated species expressions; and it addresses the problem of combinatorial explosion through a notion of nondeterminism. In addition, LBS generalises the representation of species by allowing arbitrary modification site types. This allows for example for more detailed models at the level of species binding sites corresponding to that found in rule-based languages such as κ and BioNetGen. This in turn enables a translation of LBS models into κ and BioNetGen which excel in their support for handling combinatorial explosion and for which a growing number of novel analysis techniques are becoming available.

Modular flow analysis Modularity in LBS facilitates a structured engineering approach to modelling, but there is also the hope that modularity can be exploited in analysis. Another contribution of this thesis is to show that this is indeed the case in the particular context of Petri net *flows* (also known as *invariants*). Informally, *transition flows* (or T-flows) represent chemical reactions which together have no net effect on species populations in a model. They hence correspond to a notion of cyclic pathways, and they coincide with the notion of *flux modes* [84] in cases where reactions are irreversible [43]. *Place flows* (or P-flows) represent weighted sums of species populations which are always constant. They hence correspond to chemical conservation relations.

Flow analysis has proven an important tool in biological model validation: the modeller should be able to give biological justification for each flow, otherwise it is likely that the model is incorrect for the intended purpose [44, 43]. Flows are furthermore important in the general analysis of Petri nets; for example, P-flows can be used for determining boundedness, and T-flows for investigating liveness [80].

Our results on modular analysis show how the flows of a system can be computed based on the flows of its components, and how this can be exploited in a modular definition of flows of LBS programs. Flow analysis is computationally expensive, and a modular approach can potentially reduce the computational complexity of analysis dramatically and enable parallel computation. It also allows analysis results to be reused in different contexts. Related work in this area is discussed after presenting the technical results.

A language for genetic engineering of cells GEC is based on the recent advent of standard genetic parts [34], e.g. “biobricks” [64], which can be composed to form

genes and gene networks that encode living organisms. A GEC model specifies the phenotypical characteristics of such organisms at a high level of abstraction through logical properties and interrelationships between parts. Such logical properties could, for example, specify that a part should code for a specific protein which can form a complex with another, and that the resulting complex should be a transcription factor for a part regulating the expression of a third protein. Modules allow further abstraction away from individual parts, much as designs in electronic engineering abstract away from individual boolean gates.

The process of translating a GEC model to concrete DNA sequences relies on a given database of known parts which could be based on e.g. the MIT Registry [64]. However, for our purposes, we employ a minimal proof-of-concept database.

GEC includes compartments and reactions which are used both as a basis for simulation but also to impose constraints on parts. Therefore GEC could be designed as an extension of LBS. However, in order to focus on the central problem of translating from high-level descriptions to genetic devices, GEC is designed as a separate language with simpler representations of species and reactions. Ultimately the two languages may merge.

Implementation Compilers for LBS and GEC have been implemented in the functional programming language F# [87]. The compiler for LBS translates directly to the *Systems Biology Markup Language* (SBML) [47] rather than to e.g. Petri nets or ODEs because SBML is adequate for the examples and the case studies in this thesis and because it is supported by a large body of tools. The translation to SBML relies on the libSBML library [10]. The compiler for GEC translates both to textual strings representing DNA sequences, and also to LBS programs which can subsequently be translated to SBML for simulation.

A tool with a graphical user interface for both compilers has furthermore been implemented in C#. The tool includes an editor for the GEC database; textual editors for GEC and LBS programs; and a simulator allowing both deterministic and stochastic simulations to be carried out and plotted. The latter relies on the third-party simulators provided through the *Systems Biology Workbench* (SBW) [6].

Closely related languages A few other languages share some of our general design aims. Most notably, Little b [59] and Antimony [85] combine the intuitive rule- or reaction-based approach with modularity. However, they do not have a formally de-

fined semantics (barring the standard semantics of Lisp, on which Little b is based, which is not particularly well suited for the biological domain). Hence their meaning may not always be clear and they are not readily amenable to modular analysis. They also do not have a direct counterpart of the LBS species expressions, nondeterminism and subtyping.

Antimony and another language called GenoCad [13, 14] furthermore have dedicated constructs for modelling genes following the genetic part approach that we also adopt for GEC. However, neither of these allow the abstraction away from individual parts through logical properties that GEC does. Hence large models in these languages may be difficult to comprehend.

1.3 Thesis outline

Outline We start in Chapter 2 with an overview of relevant existing modelling languages and frameworks, and focus in particular on CBS and its limitations. In Chapter 3 we introduce LBS informally through a number of small examples and we outline two larger case studies of the yeast pheromone [51] and ErbB [56] signalling pathways which are included in full in Appendix A and B. The formal presentation of LBS starts in Chapter 4 with an abstract syntax. Chapter 5 presents a general semantics of LBS which is independent of any particular choice of target semantical objects. Concrete semantics in terms of Petri nets, coloured Petri nets, ODEs, CTMCs, and κ are given in Chapter 6 by appropriate instantiations of the general framework. The results on modular analysis of Petri net flows are presented as yet another concrete semantics in Chapter 7. In Chapter 8 we turn to synthetic biology with an informal introduction to GEC through small examples and case studies; Chapter 9 defines GEC formally through an abstract syntax and semantics. Finally, Chapter 10 concludes and gives an overview of future work. Detailed proofs are given in Appendix C.

Publications Parts of this thesis were published elsewhere. An early version of LBS, without full details of the semantics, was published in [70] with Gordon Plotkin. This has been subsumed by [71], published with Gordon Plotkin, which essentially consists of the material in Chapters 3, 4, 5 and 6 but without the case studies and without the examples and formal definitions relating to κ . The material in Chapter 7 on compositional definitions of Petri net flows was published in [68]. The material on GEC in Chapters 8 and 9 was published in [69] with Andrew Phillips.

Prerequisites and Chapter Dependencies We have endeavoured to keep the thesis self-contained with respect to the necessary biological background, although an elementary understanding of cell biology is assumed. The technical chapters (4, 5, 6, 7 and 9) assume some familiarity with ideas from programming language theory and basic mathematics. The non-technical chapters (2, 3 and 8) can be read independently of the rest, and the chapters on LBS can be read independently of the chapters on GEC.

Chapter 2

Existing Formalisms for Biology

Overview In this chapter we give an informal overview of some existing formalisms for biology which have direct relevance to LBS and GEC. We start with Petri nets in Section 1 and rule-based languages in Section 2. Both are significant in the context of this thesis because LBS affords a translation to these. In Section 3 we consider two rule- or reaction-based languages which share our design objective of modularity and in Section 4 we consider two dedicated languages for synthetic biology.

In Section 5 we introduce CBS in some detail through a number of examples. The aim of doing so is two-fold. First, CBS is the basis of LBS, and therefore CBS is a natural starting point for introducing LBS. Second, the examples highlight the limitations of CBS; we show in the next chapter how these can be overcome with LBS.

This chapter is not intended as a comprehensive literature review. Particularly, we omit a treatment of the large body of work on process calculi which is not of direct relevance to the work presented in this thesis.

A categorisation of intra-cellular systems Intra-cellular systems are often divided into three principal categories. The first category is that of metabolic pathways which involve an enzyme-catalysed transformation of chemicals, for example of glucose into pyruvate as in the glycolysis pathway [3]. The second category is that of gene regulatory networks which map the effects that the activation of one gene may have on the activation of others, as for example in the repressilator circuit [33]. The third category is that of signal transduction pathways which involve the propagation of a signal from the extracellular space into the nucleus of a cell where a gene may be activated to initiate a response, as for example in the yeast pheromone pathway [51]; the signal is typically propagated through a series of protein phosphorylations.

Although the three categories are closely interrelated and one formalism is often capable of modelling systems spanning them all, we choose to focus on the latter two in this thesis. Gene regulatory networks play a central role in GEC because genes are the primary target of a translation of GEC programs. We discuss gene regulatory networks briefly in Section 4 in the context of languages for synthetic biology. Signal transduction pathways are of interest because they are difficult to model concisely, and conciseness is one of the overall aims for the languages presented in this thesis. The problem is one of combinatorial explosion as we illustrate next through a small example.

A small example exhibiting combinatorial explosion Consider the two reactions in Figure 2.0.1a in which proteins A and B bind to form a complex and where B is subsequently phosphorylated, a scenario typical of signal transduction pathways. There are three potential sources of combinatorial explosion of the number of reactions needed to model such a system:

1. **Internal combinatorial explosion.** Suppose that there is an additional modification site on protein A and that the reactions can take place regardless of the phosphorylation state of this site. One then obtains two copies of each reaction, one for each of the two phosphorylation states, as shown in Figure 2.0.1b. Generally, the number of reactions grows exponentially with the number of new sites that are added.
2. **Species variant combinatorial explosion.** Proteins sometimes exist in different variants with common functionality. Suppose for example that protein A has variants A₁ and A₂, both of which can participate in the reactions. One then obtains the four reactions shown in Figure 2.0.1c, one for each choice of the As. Generally, the number of reactions grows exponentially with the number of variants of each protein.
3. **Contextual combinatorial explosion.** Proteins can sometimes participate in a reaction regardless of which other proteins they are in complex with. Suppose for example that a third protein, C, can bind A regardless of whether or not A is bound to B, and that A can participate in the reactions regardless of whether or not it is bound to C. One then obtains two copies of each reaction, one where A is bound and one where it is not, as shown in Figure 2.0.1d. This complexity may resemble that of modification sites but is in fact worse because C can generally

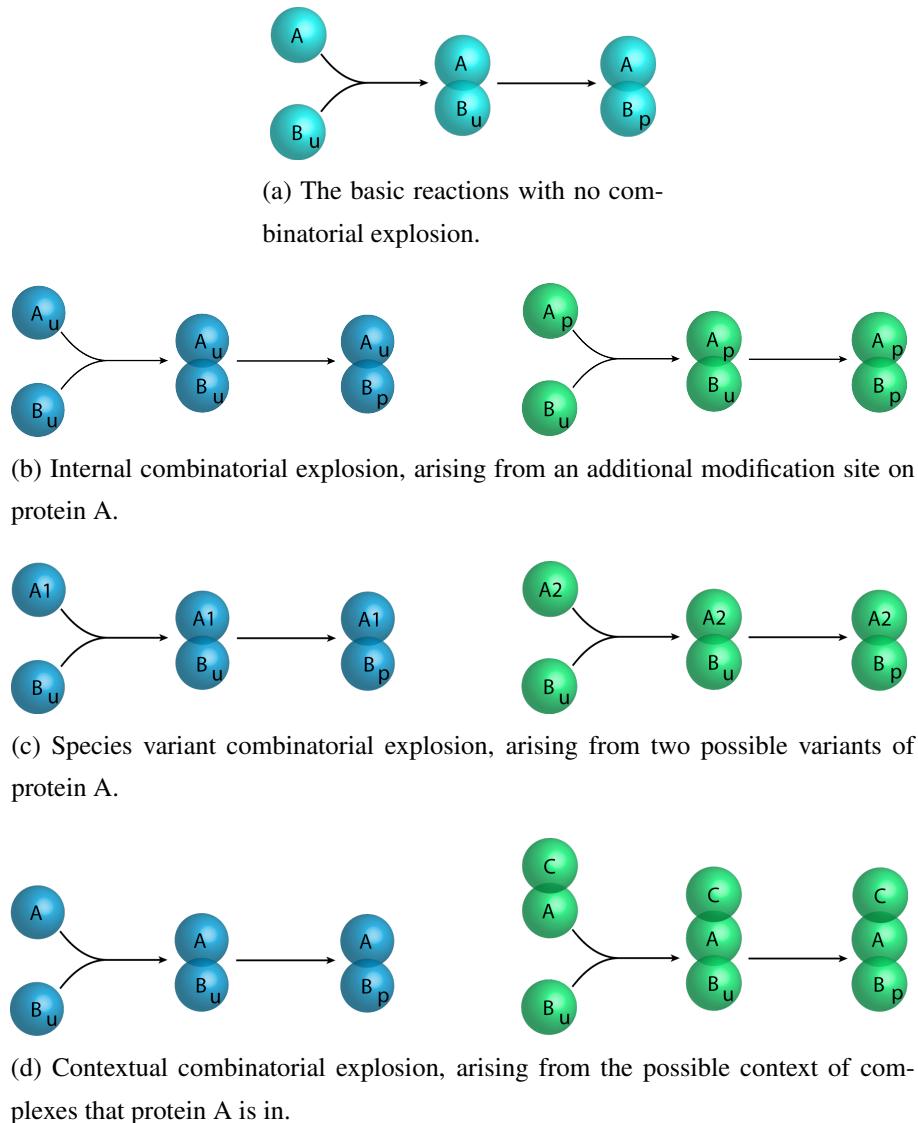


Figure 2.0.1: Graphical representations of reactions for the binding of proteins A and B with subsequent phosphorylation of B illustrated with three possible sources of combinatorial explosion. Modification sites in their unphosphorylated and phosphorylated states are represented with labels u and p, respectively.

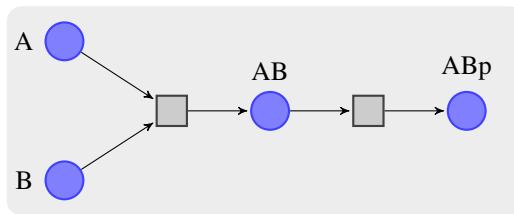


Figure 2.1.1: A basic Petri net representation of the two reactions in Figure 2.0.1a.

have many possible bindings itself. In fact, in cases where polymerisation takes place, this can even give rise to infinitely many concrete reactions.

When discussing existing formalisms in the following, we also outline any support that these provide for ameliorating the problem of combinatorial explosion.

2.1 Petri nets

Since their introduction in the sixties, Petri nets have been applied to the modelling of a wide range of distributed systems and have also been the subject of extensive theoretical studies. They are well suited for modelling biological systems because of their intuitive visual representation and the way in which they directly capture chemical reactions. We first introduce the basic notion of Petri nets and then one of their many extensions.

2.1.1 Basic Petri nets

Basic Petri nets [65] are weighted, directed bipartite graphs with nodes that are either *places* or *transitions*. A Petri net modelling the example in Figure 2.0.1a is shown in Figure 2.1.1 where places, depicted as circles, represent species, and transitions, depicted as rectangles, represent reactions. In the general case arc weights represent reaction stoichiometry, but in the example all weights are 1 and have thus been omitted. Applications of basic Petri nets to biological modelling were first reported in [77] and many others have since followed, e.g. in [38, 43, 39, 82, 86].

The state of a basic Petri net is given by a *marking*, which is an assignment of a non-negative integer number of *tokens* to each place, typically representing a population count or concentration level of the corresponding species. A transition can *fire* and when doing so, it removes a number of tokens from its input places and adds a number of tokens to its output places. The number of tokens removed (respectively added) by

a transition is given by the weights on the corresponding in-going (respectively out-going) arcs, and a transition can fire only if its input places contain at least the number tokens specified by the corresponding ingoing arc weights. The rules of this “token game” can be formalised to define the qualitative semantics of a basic Petri net and we do so in Chapter 6. Transitions can also be equipped with rates, giving rise to stochastic Petri nets [40] from which CTMCs can be derived for stochastic simulation.

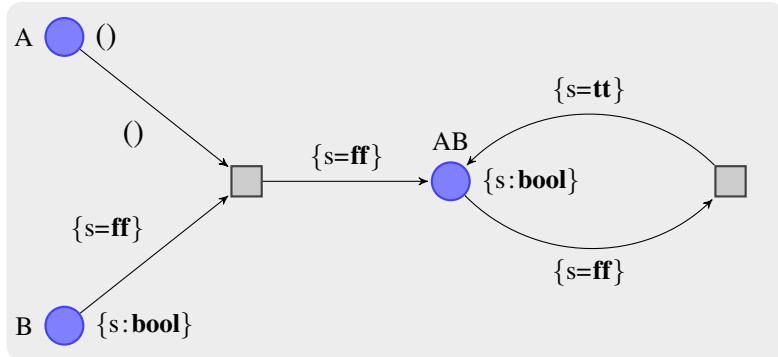
In addition to simulation, a number of well-understood analysis methods have found applications in the biological setting [43]. These include e.g. model-checking using Computation Tree Logic (CTL) and methods for *flow analysis* which give insights into chemical conservation relations and cyclic pathways in a model. Flow analysis will be treated in depth in Chapter 7 where a modular method is presented.

The examples in Figures 2.0.1b, 2.0.1c and 2.0.1d can be modelled in a similar, straightforward fashion. However, separate places and transitions are needed for each possible species and reaction. Basic Petri nets are hence susceptible to the problem of combinatorial explosion. In the context of large-scale modelling they are also limited by their lack of modularity, although there have been efforts towards modular basic Petri nets [83, 91, 23].

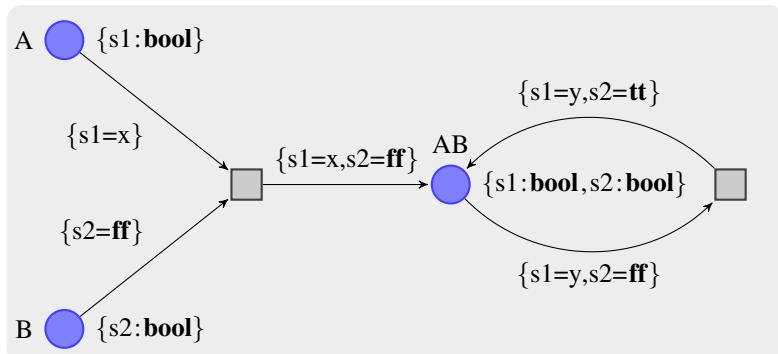
2.1.2 Coloured Petri Nets

Coloured Petri Nets (CPNs) [48] provide higher levels of abstraction by allowing tokens to have “colour”, i.e. to be marked with elements of a given datatype. Arc weights are replaced by more general arc expressions operating on the types of tokens in the associated places, and transitions may be equipped with boolean guards. An example CPN model of the reactions in Figure 2.0.1a is shown in Figure 2.1.2a; note how a single place now represents a species with modification sites by a record with boolean fields for each site. Other types than the booleans can be used for record fields, allowing for example DNA sequences (strings) or location (real-valued pairs) to be represented explicitly. Applications of CPNs to biological modelling have been reported in e.g. [88, 81, 55].

This added structure can be used to give a compact representation of systems which suffer from combinatorial explosion at the level of modification sites as demonstrated by the CPN representation in Figure 2.1.2b of the reactions in Figure 2.0.1b. Only the types and arc expressions are changed, the latter now including *variables*, and no new places or transitions are added. Combinatorial explosion at the level of atomic



(a) CPN representation of the two reactions in Figure 2.0.1a.



(b) CPN representation of the four reactions in Figure 2.0.1b illustrating how combinatorial explosion at the level of modification sites can be handled. Arc expressions here contain the variables x and y.

Figure 2.1.2: Coloured Petri net representations of the reactions in Figures 2.0.1a and 2.0.1b. Types are specified to the right of the places and arcs are labelled with expressions of the type associated with the appropriate input and output places.

species and complexes still requires addition of new places and transitions if one uses the approach outlined here, although an approach along the lines of κ or BioNetGen, discussed below, may be conceivable.

CPNs additionally have a notion of modularity. Modules, termed *pages*, can be defined and used, possibly multiple times, as building blocks of a more complex *super-net*. This is done by defining *substitution transitions* in the super-net, which can then be replaced by a particular page instance. This in turn requires a *port assignment* specifying how the places of the page are connected to the places of the substitution transition in the super-net. These features support the modelling process by allowing large models to be composed from their components.

2.2 Rule-based languages

In contrast to Petri nets which are general-purpose formalisms, rule-based languages have been designed specifically for the modelling of biological systems. Common to them all is that they describe systems in terms of rules which are abstract representations of one or more reactions.

2.2.1 BIOCHAM

BIOCHAM [20] is based on a notion of rules, originating from [22, 19], which may contain variables for modification site states, for atomic species names and for complexes, hence providing support for handling all three sources of combinatorial explosion. Variables must range over specified finite sets of values. Here is the BIOCHAM representation of the reactions in Figure 2.0.1a:

- ¹ $A + B \Rightarrow A-B$
- ² $A-B \Rightarrow A-B^{\sim} s$

Complexes are formed using the $-$ operator and $B^{\sim} s$ means that B is phosphorylated on a site called s . Reactions may also include compartments and rates.

The next example shows how variables, identified by the \$ symbol, can be used to model the four reactions in Figure 2.0.1b which arise due to combinatorial explosion at the level of modification sites:

- ¹ $A^{\sim} \$s + B \Rightarrow A^{\sim} \$s-B$ where $\$s$ in $\{ \{ \}, \{ s \} \}$
- ² $A^{\sim} \$s-B \Rightarrow A^{\sim} \$s-B^{\sim} s$ where $\$s$ in $\{ \{ \}, \{ s \} \}$

Here $\$s$ can take two values, namely the empty set of modification sites (meaning unphosphorylated) and the singleton set of one modification site (meaning phosphorylated on that site). The “where” part of the rules can be omitted if the species A is predefined with a specification of its possible modifications.

The following example shows how variables can be used to model the four reactions in Figure 2.0.1c which arise due to combinatorial explosion at the level of atomic species:

```

1 $A + B => $A-B where $A in { A1, A2 }
2 $A-B => $A-B^~s where $A in { A1, A2 }
```

A similar mechanism is used for modelling the reactions in Figure 2.0.1d which arise due to combinatorial explosion at the level of complexes:

```

1 $AC + B => $AC-B where $AC in { A, A-C }
2 $AC-B => $AC-B^~s where $AC in { A, A-C }
```

2.2.2 BioNetGen

As the name suggests, BioNetGen [35] is a language designed for generating a biochemical network, essentially a set of reactions or a basic Petri net, from an abstract, rule-based description. More recently, BioNetGen also allows simulation to be carried out directly at the level of rules [8], without generating the underlying network which may be very large or even infinite.

In contrast to BIOCHAM, there is no direct support for compartments, and rules do not use variables to range over atomic species or complexes. Instead they describe complexes at the lower level of binding sites, and bindings may be left unspecified which gives rise to a notion of pattern matching. Here is a BioNetGen representation of the reactions in Figure 2.0.1a:

```

1 A(s) + B(s^~u) -> A(s!1).B(s^~u!1)
2 A(s!1).B(s^~u!1) -> A(s!1).B(s^~p!1)
```

$A(s)$ represents A with the site s unbound but in any state of modification, and $B(s^~u)$ represents B with s unbound and unphosphorylated. Complexes are formed using the $.$ operator and bindings within complexes are specified using natural number labels which have scope of the complexes in which they occur. The product side of the first rule then represents a complex where the site s in A is bound to the site s in B.

The first rule can be applied to *any* instances of A and B that are unbound on site s and where B is unphosphorylated on this site, with the condition that they are not otherwise bound in the same complex. In particular it can be applied to instances of A which have other modification sites as in Figure 2.0.1b, or to instances which are bound to other proteins as in Figure 2.0.1d. In this way BioNetGen transparently supports combinatorial explosion at the level of species modifications and at the level of complexes. There is however no support for combinatorial explosion at the level of atomic species, so the reactions in Figure 2.0.1c are modelled using two additional rules.

The explicit representation of binding sites in support of contextual combinatorial explosion leads to increased expressiveness compared to e.g. Petri nets: BioNetGen, and also κ to be discussed next, are Turing-complete [18].

2.2.3 κ

The κ calculus [30] is syntactically very similar to BioNetGen although there are some subtle differences. Here is the κ representation of the reactions in Figure 2.0.1a:

```

1 A(s) , B(s~u) -> A(s!1) , B(s~u!1)
2 A(s!1) , B(s~u!1) -> A(s!1) , B(s~p!1)
```

Compared to the corresponding BioNetGen representation, there is only the single “comma” operator in place of the sum and complex formation operators. Semantically, the difference is that the rule may be applied to instances of A and B which are already bound together in some complex, either directly on some other site than s or through some intermediary, which allows for more efficient simulation of rules [31]. There are also some semantical subtleties regarding (the lack of) commutativity of the comma operator, and we return to this in Chapter 6 where we give a concrete semantics of LBS in terms of κ .

As for BioNetGen, κ excels in its support for handling combinatorial explosion at the level of modification sites and complexes but not at the level of atomic species. A recent meta-language [29] addresses this problem through a notion of *generic* agents. The reactions in Figure 2.0.1c can then be represented as follows:

```

1 generic A(s);
2 concrete A1 <: A;
3 concrete A2 <: A;
4 concrete B(s);
5
```

6 A(s), B(s~u) → A(s!1), B(s~u!1)
 7 A(s!1), B(s~u!1) → A(s!1), B(s~p!1)

Note how the rules remain the same, but the definition of A as a generic agent with two concrete variants yields the expected result. There are some semantics design choices when for example A occurs multiple times on the reactant side, and we return to this question in Chapter 5 when treating the nondeterminism feature of LBS.

2.3 Rule-based Languages with Modularity

Although other formalisms such as process calculi provide support for modularity, this is not generally the case for rule-based languages such as the ones discussed above. In this section we briefly discuss two languages which combine the reaction or rule-based approach with a notion of modularity.

2.3.1 Little b

Little b [59] is designed to support the sharing and reuse of models through modularity and is built around the functional language Lisp. The Lisp foundation provides a high degree of flexibility since any custom functionality can in principle be programmed into a model. The module system supports parameterisation and is in some respects more powerful than the module system for LBS, introduced in the next chapter; for example, Little b allows for modules which phosphorylate a variable number of sites on a species parameter, which is not currently possible in LBS.

Little b employs a notion of rules [60] at a similar level of abstraction to those of BioNetGen and κ. The reactions in Figure 2.0.1a can be represented in BioNetGen as follows:

```
1 (defmonomer A s)
2 (defmonomer B s1 (s2 :states (member :u :p)))
3
4 (define r1 {[A _] + [B _ :u] ->> [[A 1][B 1 :u]]})
5 (define r2 {[ [A 1][B 1 :u]] ->> [[A 1][B 1 :p]]})
```

Lines 1 and 2 define the two species A and B, and in the case of B, two separate sites are required for binding (s1) and for phosphorylation (s2). The latter is specified to take two modification values representing the unphosphorylated and phosphorylated states.

Lines 4 and 5 then define the two rules, r1 and r2. Both species and rules evaluate to Lisp objects which can subsequently be manipulated or simulated.

Little b has a notion of wild cards. These can be used to handle combinatorial explosion at the level of modification sites and bindings following the approach in κ and BioNetGen, the difference being that wild cards are implicit in these latter languages. Wild cards can also be used in place of atomic species names, providing a means of handling combinatorial explosion arising from species variants. A tagging mechanism furthermore provides a means of restricting wild card matches.

The flexibility of Little b does however come at a price. For example it requires users of the language to have some familiarity with Lisp. Although e.g. infix notation helps to achieve a more natural representation of reactions, the syntax does not appear to be particularly close to the biological domain. And although Little b by virtue of its Lisp foundation does have a formal semantics, the semantics is not very transparent and it is not well suited for e.g. the compositional Petri net analysis that we study later in this thesis.

2.3.2 Antimony

Antimony [85] is designed as a human-readable and modular language for both systems and synthetic biology; we discuss the latter aspect in the next subsection. Models are specified in terms of reactions which are simpler than those of Little b, as can be seen from the following Antimony representation of the reactions in Figure 2.0.1a:

```

1 A + B -> A.B
2 A.B -> A.Bs

```

There is no language-level support for modification sites and complexes: all species are treated as atomic identifiers, i.e. the underscore used in the complex A.B is not an operator and the s, indicating phosphorylation, is arbitrarily appended to the identifier A.B. Reactions do not allow for rule-based abstraction, so there is no direct support for handling combinatorial explosion.

Antimony does however provide a range of other features with a particular focus on those found in SBML. These include for example dynamic compartment volumes and events for e.g. instantaneous changes to species populations. The module system allows parameterisation on species, compartments and rates.

A significant limitation of Antimony is its lack of a formal semantics. There are tools for translating Antimony to SBML, but the mechanism of this translation is not

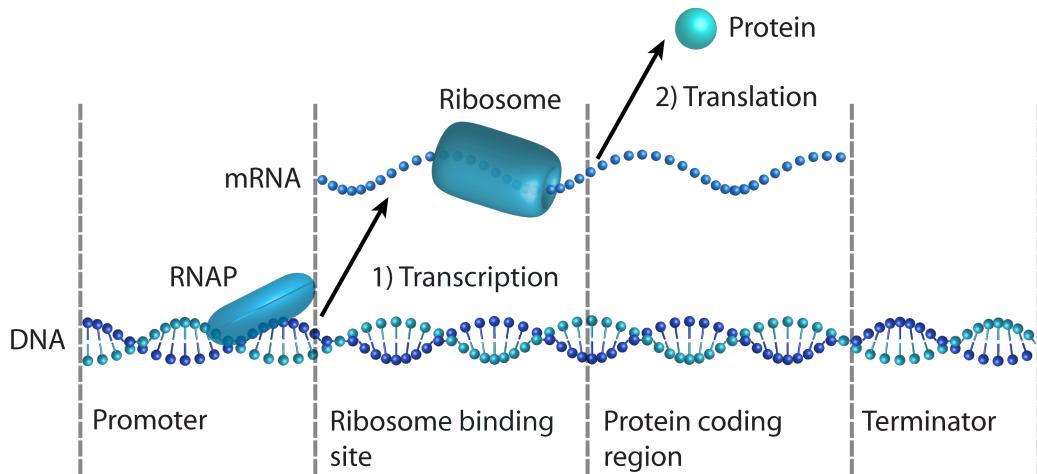


Figure 2.4.1: Illustration of gene expression in bacteria (and other prokaryotes) and the role of four principal DNA parts constituting the gene.

formally defined, and neither is the semantics of SBML itself. The lack of any formal semantics, let alone a compositional one, also means that Antimony is not readily suited for the kind of compositional Petri net analysis that we study later in this thesis.

2.4 Languages for Synthetic Biology

We now turn briefly to languages which explicitly support synthetic biology through abstractions for standard genetic parts. Parts are sequences of DNA that can be composed to form genes. A typical gene is composed of at least four parts as shown in Figure 2.4.1. The *promoter* is responsible for binding a “transcriber”, RNAP, which transcribes the down-stream DNA into mRNA. The *ribosome binding site* is transcribed into mRNA which binds to a “translator”, the ribosome. The ribosome translates the down-stream mRNA resulting from the *protein coding region* into a target protein. Finally, the *terminator* signals end-of-transcription to the RNAP.

Graphical tools for designing genes based on standard parts and for simulating the dynamics of gene expression have recently started to emerge [61]. As of yet there are however only two languages, other than our GEC, which support this process.

2.4.1 GenoCad

The first is GenoCad [13], a simple context-free language consisting of biologically meaningful sequences of part names. Any sequence of part names adhering to the scheme of *promoter*, *ribosome binding site*, *protein coding region* and *terminator*

shown in Figure 2.4.1 is for example included, as for example the following GenoCad program with part names referring to those found in the MIT Registry:

```
1 r0040 b0034 c0040 b0015
```

Sequences with e.g. the promoter (r0040) and ribosome binding site (b0034) swapped are not included.

A recent extension [14] gives a semantics to GenoCad programs in terms of reactions which represent the emerging dynamics of gene expression. This semantics is defined using attribute grammars and relies on having appropriate mass-action rates associated with the various parts; a rate of transcription is for example associated with a promoter part. We give a similar translation to reactions from GEC models in Chapter 9 using standard ideas from denotational semantics rather than attribute grammars.

2.4.2 Antimony

We have already described the central features of Antimony for systems biology modelling. Antimony models can furthermore contain sequences of standard genetic parts for representing gene networks. In contrast to GenoCad, these sequences can be composed in a modular fashion and models can be parameterised on part names. Models can be simulated as in GenoCad, but rather than using mass-action rates, Antimony employs the notion of *Polymerases Per Second* (PoPS) and *Ribosomes Per Second* (RiPS) as in [61]. These are measures of how many RNAPs, respectively ribosomes, pass over a specified area of DNA, respectively mRNA, per second, and can be used to derive ODEs.

2.5 CBS by Example

CBS is syntactically very similar to BIOCHAM. The reactions in Figure 2.0.1a can be written as follows:

```
1 A + B{ s=ff } -> A-B |
2 A-B -> A-B{ s=tt }
```

Complexes are formed by composing modified primitive species using the *complex formation operator*, `-`, as in BIOCHAM. Modifications sites have boolean values with `ff` (false) representing “unphosphorylated” and `tt` (true) representing “phosphorylated”. As a shorthand we may write e.g. `Fus3` instead of `Fus3{p=ff}` and `Fus3{p}` instead of

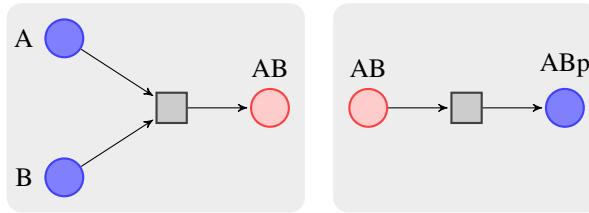


Figure 2.5.1: Petri nets arising from the individual rules of a CBS program. These can be composed by merging the common places (highlighted) to obtain the Petri net in Figure 2.1.1 arising from the full CBS program.

$\text{Fus3}\{\text{p}=\text{tt}\}$, again following the approach of BIOCHAM. In contrast to BIOCHAM however, reactions are separated by the *parallel composition operator*, $|$, which forms the basis of a modular semantics of CBS.

Semantically, the above CBS program gives rise to the Petri net previously encountered in Figure 2.1.1, and this can be computed in a modular manner as illustrated in Figure 2.5.1. The first reaction gives rise to a Petri net with the three places A, B and AB together with a transition connecting these. The second reaction gives rise to a Petri net with two places AB and ABp and an associated transition. The full Petri net associated with the parallel composition is obtained by merging the common places as determined by syntactic equality on names, in this case just AB. We observe that the sum and parallel composition operators are commutative, and so is the complex formation operator because place names are formally *multisets* of modified atomic species. Modular semantics in terms of ODEs and CTMCs can also be defined assuming that reactions are labelled with rate constants.

CBS allows the assignment of general boolean expressions to modification sites, and boolean expressions may include variables, hence ameliorating the problem of combinatorial explosion at this level. The reactions in Figure 2.0.1b can for example be represented as follows, where x is a variable:

1 $A\{ s=x \} + B \rightarrow A\{ s=x \}-B \mid$
 2 $A\{ s=x \}-B \rightarrow A\{ s=x \}-B\{ s \}$

Semantically the above CBS program gives rise to the CPN previously encountered in Figure 2.1.2b, and this can be computed in a modular manner as for the previous example. CBS does not allow variables in place of atomic species or complexes as in BIOCHAM, so there is no support for handling the other sources of combinatorial explosion.

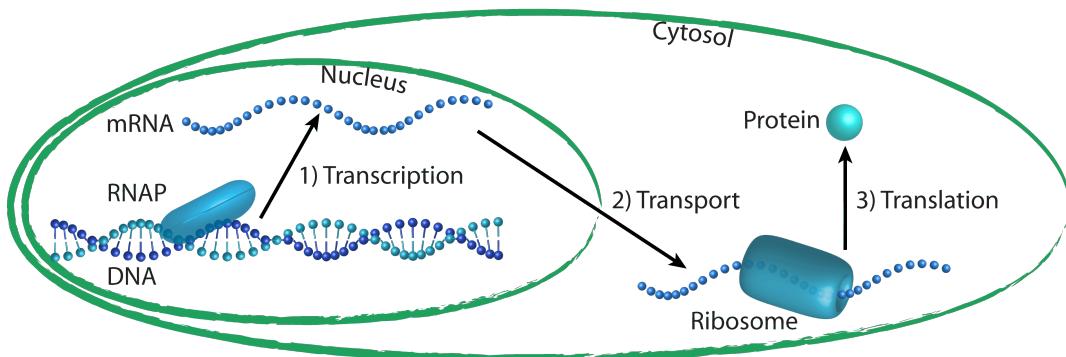


Figure 2.5.2: An informal pictorial diagram of eukaryotic gene expression taking place in three steps: 1) transcription, 2) transport and 3) translation.

Listing 2.5.1: A CBS program for gene expression.

```

1 c [
2   n[ gene + rnap -> gene + rnap + mrna ] |
3   n[ mrna ] -> mrna |
4   rs + mrna -> rs + prot
5 ]

```

The remainder of this section gives some further, larger examples of CBS models. These examples serve to demonstrate additional features of CBS, namely compartments and modules, and they also form the basis of the presentation of LBS in the next chapter.

2.5.1 Gene Expression

We first consider a basic model of gene expression. We abstract away from individual parts and consider a gene as an atomic entity. In order to illustrate the use of compartments, we furthermore consider eukaryotic rather than prokaryotic gene expression. Eukaryotes have a nucleus and gene expression involves the additional step of transporting mRNA from the nucleus into the cytosol as outlined in Figure 2.5.2.

A corresponding CBS program is shown in Listing 2.5.1. The program consists of three reactions composed in parallel and taking place inside a cytosol *compartment* *c*.

The first reaction models transcription. It is located inside a nested nucleus compartment n and produces mRNA from a gene and an RNAP. The second reaction models transport of mRNA out of the nucleus into the enclosing cytosol compartment, and the third reaction models translation of mRNA into protein by a ribosome. Observe how compartments are used both at the level of individual species and at the level of entire programs; in contrast, compartments in BIOCHAM, Little b and Antimony are restricted to individual species or reactions.

Semantically, compartments give rise to renamings of e.g. Petri net places such that species which have the same name but are located in different compartments are represented by different places. Compartments distribute over parallel compositions and reactant/product sums. In this example we could have omitted the cytosol compartment in which case a default top level compartment would be assumed.

2.5.2 A MAPK Cascade

We now shift the focus to a MAPK cascade which is ubiquitous in many signalling pathways. For the next example we choose an adapted (but not identical) version of a previously published Ras/Raf/MEK/ERK cascade [28]. An informal graphical representation is shown in Figure 2.5.3 and has three levels: in level one Ras (the MAPK4) phosphorylates Raf; in level two phosphorylated Raf (the MAPK3) phosphorylates MEK twice; and in level three, doubly-phosphorylated MEK (the MAPK2) phosphorylates ERK (the MAPK) twice. Each phosphorylation step involves three reactions: binding of the kinase and ligand, phosphorylation of the bound ligand, and dissociation of the phosphorylated ligand from its kinase. We furthermore include the corresponding dephosphorylation steps.

A CBS model of this MAPK cascade is shown in Listing 2.5.2. Each phosphorylation/dephosphorylation cycle is modelled separately using *module definitions*, and the main body of the program in line 42 simply invokes the modules in parallel. Such a modular approach simplifies the presentation and should be contrasted with other rule-based approaches using e.g. BIOCHAM where the program would consist of one long, unstructured list of reactions.

We do however observe a high degree of redundancy. All five modules have the same structure, consisting of two sets of three reactions for binding, modification and unbinding. A shorter version of the program could in principle be obtained through appropriate derived forms for enzymatic reactions. But it appears unlikely that a small,

Listing 2.5.2: A modular CBS program for the Ras/Raf/MEK/ERK MAPK cascade.

```

1  module rafCycle {
2      Ras + Raf -> Ras–Raf | 
3      Ras–Raf -> Ras–Raf{m} | 
4      Ras–Raf{m} -> Ras + Raf{m} | 
5      PP2A1 + Raf{m} -> PP2A1–Raf{m} | 
6      PP2A1–Raf{m} -> PP2A1–Raf | 
7      PP2A1–Raf -> PP2A1 + Raf
8  };
9  module mekCycle1 {
10     Raf{m} + MEK -> Raf{m}–MEK | 
11     Raf{m}–MEK -> Raf{m}–MEK{S218} | 
12     Raf{m}–MEK{S218} -> Raf{m} + MEK{S218} | 
13     PP2A2 + MEK{S218} -> PP2A2–MEK{S218} | 
14     PP2A2–MEK{S218} -> PP2A2–MEK | 
15     PP2A2–MEK -> PP2A2 + MEK
16  };
17  module mekCycle2 {
18     Raf{m} + MEK{S218} -> Raf{m}–MEK{S218} | 
19     Raf{m}–MEK{S218} -> Raf{m}–MEK{S218, S222} | 
20     Raf{m}–MEK{S218, S222} -> Raf{m} + MEK{S218, S222} | 
21     PP2A2 + MEK{S218, S222} -> PP2A2–MEK{S218, S222} | 
22     PP2A2–MEK{S218, S222} -> PP2A2–MEK{S218} | 
23     PP2A2–MEK{S218} -> PP2A2 + MEK{S218}
24  };
25  module erkCycle1 {
26     MEK{S218, S222} + ERK -> MEK{S218, S222}–ERK | 
27     MEK{S218, S222}–ERK -> MEK{S218, S222}–ERK{T185} | 
28     MEK{S218, S222}–ERK{T185} -> MEK{S218, S222} + ERK{T185} | 
29     MKP3 + ERK{T185} -> MKP3–ERK{T185} | 
30     MKP3–ERK{T185} -> MKP3–ERK | 
31     MKP3–ERK -> MKP3 + ERK
32  };
33  module erkCycle2 {
34     MEK{S218, S222} + ERK{T185} -> MEK{S218, S222}–ERK{T185} | 
35     MEK{S218, S222}–ERK{T185} -> MEK{S218, S222}–ERK{T185, Y187} | 
36     MEK{S218, S222}–ERK{T185, Y187} -> 
37         MEK{S218, S222} + ERK{T185, Y187} | 
38     MKP3 + ERK{T185, Y187} -> MKP3–ERK{T185, Y187} | 
39     MKP3–ERK{T185, Y187} -> MKP3–ERK{T185} | 
40     MKP3–ERK{T185} -> MKP3 + ERK{T185}
41  };
42  rafCycle | mekCycle1 | mekCycle2 | erkCycle1 | erkCycle2

```

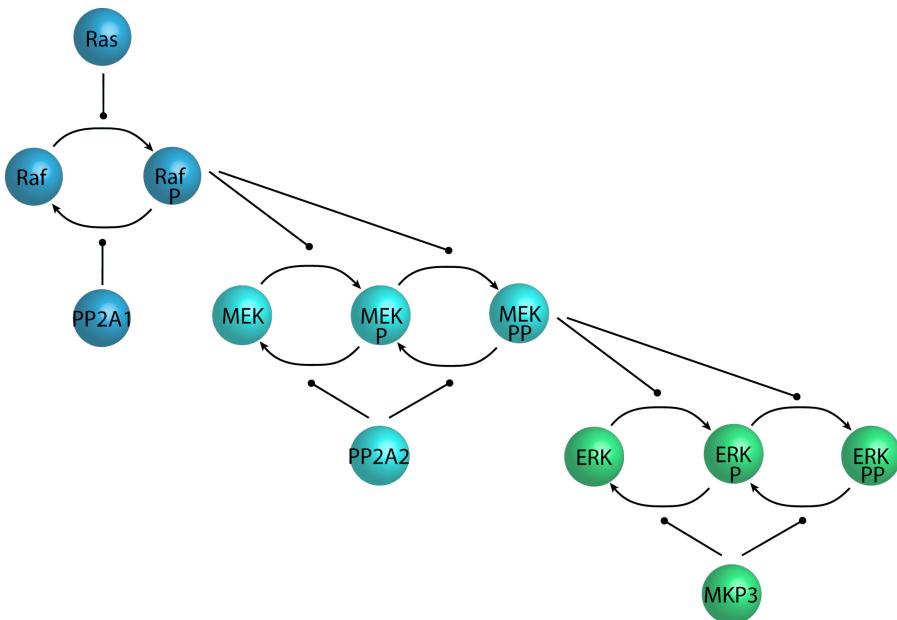


Figure 2.5.3: A Ras/Raf/MEK/ERK MAPK cascade represented by five phosphorylation/dephosphorylation cycles. Each phosphorylation and dephosphorylation step covers three underlying reactions for binding, phosphorylation/dephosphorylation, and unbinding.

fixed set of derived forms can cater for all the variants that a modeller may encounter. We may for example wish to consider variants of the above model in which all the binding reactions are reversible, or in which binding and phosphorylation are combined into a single reaction. Hence we seek to address the problem of reusability through language-level support for parameterised modules in LBS.

2.5.3 A Scaffolded MAPK Cascade.

The next example is based on a scaffolded MAPK cascade from the yeast pheromone pathway [51]. This model is simpler than the previous one in that we only consider a single phosphorylation site in each species and each reaction represents an autophosphorylation involving only a single reactant. The model is however more complicated in that relatively large scaffolding complexes are used. An informal graphical representation is shown in Figure 2.5.4 and the corresponding CBS program is shown in Listing 2.5.3.

The first five reactions in lines 1 – 12 model the formation of the scaffold complex and correspond to the left part of Figure 2.5.4. The last three reactions in lines 14 – 21 model the actual MAPK cascade, with each reaction phosphorylating a single atomic

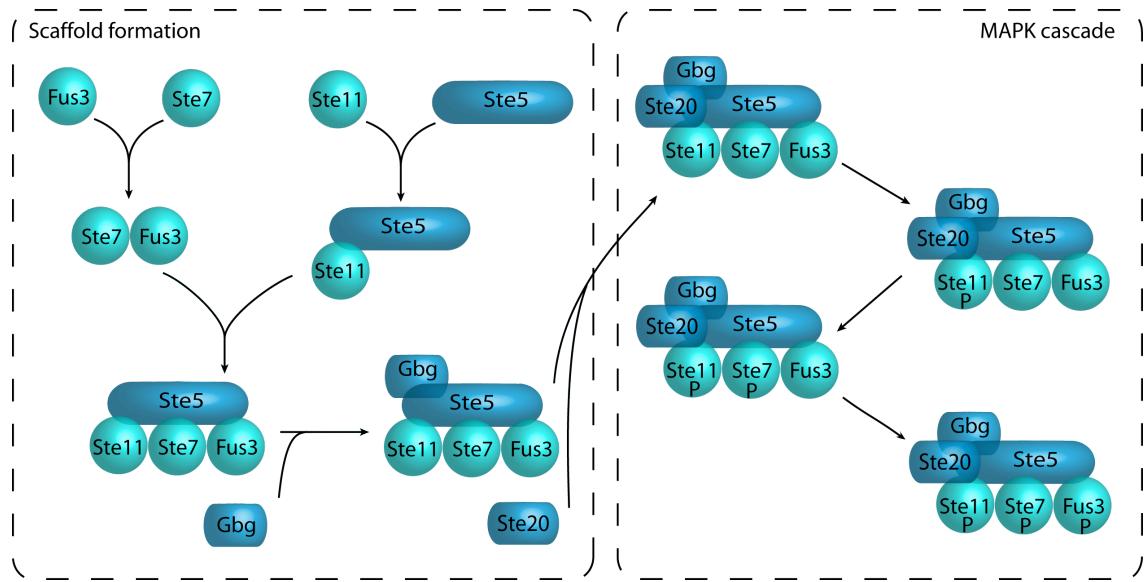


Figure 2.5.4: Scaffold formation and a scaffolded MAPK cascade [51].

Listing 2.5.3: A CBS program for a scaffolded MAPK cascade in yeast.

```

1 Ste5 + Ste11 -> Ste5-Ste11 |
2
3 Ste7 + Fus3 -> Ste7-Fus3 |
4
5 Ste5-Ste11 + Ste7-Fus3 ->
6   Ste5-Ste11-Ste7-Fus3 |
7
8 Ste5-Ste11-Ste7-Fus3 + Gbg ->
9   Ste5-Ste11-Ste7-Fus3-Gbg |
10
11 Ste5-Ste11-Ste7-Fus3-Gbg + Ste20 ->
12   Ste5-Ste11-Ste7-Fus3-Gbg-Ste20 |
13
14 Ste5-Ste11-Ste7-Fus3-Gbg-Ste20 ->
15   Ste5-Ste11{p}-Ste7-Fus3-Gbg-Ste20 |
16
17 Ste5-Ste11{p}-Ste7-Fus3-Gbg-Ste20 ->
18   Ste5-Ste11{p}-Ste7{p}-Fus3-Gbg-Ste20 |
19
20 Ste5-Ste11{p}-Ste7{p}-Fus3-Gbg-Ste20 ->
21   Ste5-Ste11{p}-Ste7{p}-Fus3{p}-Gbg-Ste20

```

species in a complex reactant, and correspond to the right part of Figure 2.5.4. The scaffold is formed by the atomic species Ste5, Ste20 and Gbg, and the species Fus3, Ste7 and Ste11 serve the MAPK, MAPK2 and MAPK3 roles, respectively. All species except Ste20 and Gbg have a single modification site, p, which can be either phosphorylated or unphosphorylated, indicated by the presence of the site name as before.

As in the previous example one immediately notices a high level of redundancy, but here for a different reason. As is common in signalling pathways, some reactions change just a single state of modification in a large complex, yet unaffected parts of the complexes are listed repeatedly. We address this problem with dedicated language constructs for species expressions in LBS.

Chapter 3

LBS by Example

The previous chapter introduced a number of existing formalisms, including CBS which forms the foundation of LBS. We demonstrated how CBS is limited by the lack of parameterised modules, and by the lack of support for handling combinatorial explosion and large complexes in a concise manner. This chapter introduces LBS informally through examples showing how LBS constructs can be used to improve, and go beyond, the CBS models given in the previous chapter. Two LBS case studies, on the yeast pheromone and the ErbB signalling pathways, are also outlined.

3.1 Gene Expression

We start with the model of eukaryotic gene expression shown earlier in Figure 2.5.2 on page 23. First the exact CBS model is replicated in LBS, demonstrating the concept of new species and compartment definitions. We then go beyond the CBS model by showing how to model the expression of multiple genes in a compact manner using parameterised modules.

3.1.1 New Species and Compartment Definitions

CBS has a static semantics which catches typos by requiring that only species names in a given set are used in programs. In LBS we include both *new species definitions* and *new compartment definitions* directly in the language. New species definitions include a list of modification sites, if any, together with their type, and new compartment definitions include a specification of the parent, if any, and an optional volume. The volume is used when compartments are referred to in rate expressions. The seman-

Listing 3.1.1: An LBS program for gene expression.

```

1 spec gene = new{}, mrna = new{}, prot = new{};
2 spec rnap = new{}, rs = new{};
3 comp c = new comp;
4 comp n = new comp inside c;
5
6 c[
7   n[gene + rnap -> gene + rnap + mrna] |
8   n [mrna] -> mrna |
9   rs + mrna -> rs + prot
10 ]

```

tics of LBS requires that species are only used with their defined modification types and that compartments are only used inside their defined parents. New species and compartment definitions are demonstrated by the program in Listing 3.1.1 which is identical to the corresponding CBS program in Listing 2.5.1 on page 23 except for the added definitions in the first four lines.

The expression `new{}` is formally a *species expression* which evaluates to a *species value* with no modification sites and with a globally unique name that is used in e.g. the underlying Petri net semantics. New species are bound to *species identifiers* such as `mrna` which do not themselves hold any identity of a species; we may hence bind the same identifier to an entirely different species in another part of the program. This allows different modules, possibly developed by different people, to use the same species identifier for mRNA species which are semantically and biologically different, and subsequently combine the modules into a single program without unintended cross-talk. On the other hand, when species *are* intended to be shared between modules, the species should be defined globally or made parameters of modules as we demonstrate next.

3.1.2 Parameterised Modules

We extend the basic gene expression program to express *two* proteins, `prot1` and `prot2`, from two different genes, `gene1` and `gene2`. We do so by abstracting the gene expression process into a *parameterised module* and invoking the module twice with the relevant parameters. The result is shown in Listing 3.1.2.

The RNAP and the ribosome are defined globally in lines 1 – 2, meaning that they

Listing 3.1.2: A modular LBS program for gene expression instantiated with two genes and two target proteins.

```

1 spec rnap = new {};
2 spec rs    = new {};
3
4 module m(comp nuc; spec gene, prot) {
5   spec mrna = new {};
6   nuc [gene + rnap -> gene + rnap + mrna] |
7   nuc [mrna] -> mrna |
8   rs + mrna -> rs + prot
9 };
10
11 spec gene1 = new {}, prot1 = new {};
12 spec gene2 = new {}, prot2 = new {};
13 comp c = new comp;
14 comp n = new comp inside c;
15
16 c[ m(n, gene1, prot1) | m(n, gene2, prot2) ]

```

will be shared between all instances of the module defined in lines 4 – 9. This is biologically meaningful since the same RNAP and ribosome species are used for transcription and translation independently of the gene in question. The module is parameterised on the nucleus compartment, the gene and the target protein. The body is similar to before, except that a new mRNA species is defined locally in line 5. This means that each instance of the module uses semantically distinct mRNA, which again is biologically meaningful. Lines 11 – 14 define the genes and proteins to be expressed together with the relevant compartments, and line 16 is a parallel composition of two module invocations inside the cytosol compartment. We could choose to define the nucleus compartment globally in this particular case, but instead give it as a common parameter in both module invocations in order to illustrate how this can be done in the more general case.

3.2 A MAPK Cascade

Parameterised modules can also be used to improve the CBS model of the MAPK cascade in Figure 2.5.3 on page 26. In order to do so, two further LBS features, parametric

type and subtyping, are needed and we start by demonstrating these. We then go beyond the CBS model and show how to handle combinatorial explosion both *explicitly* through the use of nondeterminism and *implicitly* by incorporating the approach of κ and BioNetGen into LBS. Finally, we show how a variant operator can be used to generate multiple variants of the MAPK cascade model for subsequent computational investigation.

3.2.1 Parametric Type and Subtyping

Recall how all the cycle modules used by the CBS program in Listing 2.5.2 on page 25 have the same structure, each with two sets of reactions for representing, respectively, phosphorylation and dephosphorylation. The LBS program in listing 3.2.1 shows how a general, parameterised cycle module can be defined, which in turn relies on two general modules for phosphorylation and dephosphorylation.

The phosphorylation module named ph in lines 1-5 contains three reactions: binding of a kinase k and substrate s, phosphorylation of s in the bound state, and unbinding after phosphorylation. The two species are formal parameters, but in contrast to earlier examples, the formal parameter s has an *annotation* specifying that it must have a modification site m. The dephosphorylation module named dph in lines 7-11 follows a similar structure but is parameterised on a phosphatase p rather than a kinase. The cycle module in lines 13-15 is parameterised on a kinase, a phosphatase and a substrate and invokes the phosphorylation and dephosphorylation modules in parallel. The invocations provide annotations for matching up the modification sites in the actual parameters with the corresponding modification sites in the formal parameters, which in this case is trivial since there is only the single choice, m. Note that there is scope for further abstraction since the phosphorylation and dephosphorylation modules are very similar. In fact they could be abstracted into a single module, but we refrain from doing so for the sake of clarity.

Lines 17-21 define the new species participating in the program and the remaining lines invoke modules for the appropriate cycles. Let us consider the invocation in line 24 in more detail. The first actual parameter, Raf{m}, provides Raf in its phosphorylated state as the kinase, and the second parameter provides PP2A2 as the phosphatase. The third parameter, MEK:{S218}, provides unphosphorylated MEK as the substrate and the annotation {S218} specifies the target site for phosphorylation. This raises two important points. First, the names of modification sites in the actual and formal an-

Listing 3.2.1: A modular LBS program for the Ras/Raf/MEK/ERK signalling cascade.

```

1  module ph( spec k, s:{m} ) {
2      k + s -> k-s |
3      k-s -> k-s{m} |
4      k-s{m} -> k + s{m}
5  };
6
7  module dph( spec p, s:{m} ) {
8      p + s{m} -> p-s{m} |
9      p-s{m} -> p-s |
10     p-s -> p + s
11 };
12
13 module cycle( spec k, p, s:{m} ) {
14     ph( k, s:{m} ) | dph( p, s:{m} )
15 };
16
17 spec Ras = new{};
18 spec Raf = new{m: bool};
19 spec MEK = new{S218: bool, S222: bool};
20 spec ERK = new{T185: bool, Y187: bool};
21 spec PP2A1 = new{}, PP2A2 = new{}, MKP3 = new{};
22
23 cycle(Ras, PP2A1, Raf:{m}) |
24 cycle(Raf{m}, PP2A2, MEK:{S218}) |
25 cycle(Raf{m}, PP2A2, MEK{S218}:{S222}) |
26 cycle(MEK{S218, S222}, MKP3, ERK:{T185}) |
27 cycle(MEK{S218, S222}, MKP3, ERK{T185}:{Y187})

```

notations differ, resulting in a notion of *parametric type*. The underlying semantics maintains a mapping from formal to actual modification site names when evaluating the body of a module. Second, there are two possible choices of modification sites to be phosphorylated in the actual parameter, namely S218 and S222. The annotation picks out the former, and the latter then plays no role from the perspective of the module. This results in a notion of *subtyping*: any actual parameter will do, as long as it contains at least the sites specified in the annotation and with types that match the corresponding formals. This corresponds to record subtyping in classical programming languages [73]. The module invocation in line 25 is similar but picks out the second site, S222, for phosphorylation, and also specifies that MEK is already phosphorylated on site S218.

In general, parameters may be complexes rather than atomic species. Suppose for example that MEK is in a complex with some other species, a, in the actual parameter in line 25. This can be written as follows:

```

1 ...
2 cycle (Raf{m}, PP2A2, MEK{S218}-a:MEK{S222}) |
3 ...

```

This results in an additional layer of subtyping: any actual parameter will do, as long as it contains at least the atomic species specified in the annotation. The annotation is here extended in order to specify that it is the atomic species MEK rather than a that should be mapped to the substrate. In fact the annotations used in Listing 3.2.1 are abbreviations of this more general form, so e.g. MEK{S218}:{S222} is an abbreviation of MEK{S218}:MEK{S222}. Similarly, the annotated formal parameter s:{m} in the cycle module abbreviates s:s{m}, and formal annotations may in general contain multiple atomic species.

We end the discussion of parameterised modules with an abstraction of the entire MAPK cascade into a module which is itself reusable. This, together with a module invocation, is shown in Listing 3.2.2.

3.2.2 Nondeterminism

Nondeterminism for contextual combinatorial explosion In the CBS MAPK cascade model we assumed that species only participate in reactions when they are atomic or when they are in the context of a fully specified complex. Recall that contextual combinatorial explosion arises from proteins taking part in the same reaction while be-

Listing 3.2.2: A general, modular LBS program for the Ras/Raf/MEK/ERK signalling cascade. The species and module definitions from Listing 3.2.1 are omitted.

```

1 ...
2 module mapk(
3   spec k4, k3:{m}, k2:{m1, m2}, k1:{m1, m2},
4     p3, p2, p1) {
5
6   cycle(k4, p3, k3:{m}) |
7   cycle(k3{m}, p2, k2:{m1}) |
8   cycle(k3{m}, p2, k2{m1}:{m2}) |
9   cycle(k2{m1,m2}, p1, k1:{m1}) |
10  cycle(k2{m1,m2}, p1, k1{m1}:{m2})
11 };
12
13 mapk(Ras, Raf:{m}, MEK:{S218, S222}, ERK:{T185, Y187},
14   PP2A1, PP2A2, MKP3)

```

Listing 3.2.3: Phosphorylation using nondeterministic species.

```

1 spec NMEK = MEK{S218, S222}-(SNil or Raf{m} or PP2A2);
2 spec NERK = ERK-(SNil or MKP3);
3
4 ph(NMEK, NERK:ERK{T185})

```

ing bound in multiple different complexes. In the MAPK cascade example, MEK may for example continue to function as a kinase for ERK when it is bound to its own kinase and/or phosphatase and when ERK is bound to its own phosphatase, and similarly for other participating species.

One approach to handling this complexity is to adopt a κ or BioNetGen approach as indeed we demonstrate later. We start however with a middle ground in which all possible species contexts continue to be specified in reactions, but in a syntactically compact manner through the notion of nondeterministic species. Listing 3.2.3 shows an example of phosphorylation using nondeterministic versions of MEK and ERK

The **or** operator expresses that either of its operands can take part in reactions where the expression is used. The distinguished species **SNil** is a neutral element under the complex formation operator, i.e. the axiom $a-\text{SNil} = a$ holds for any species a . The distributivity axiom $a-(b \text{ or } c) = a-b \text{ or } a-c$ also holds for all species a, b and c . Hence

Listing 3.2.4: The MAPK cascade module instantiated with nondeterministic species.

```

1 ...
2 spec Ras = new {};
3 spec RafA = new{m: bool};
4 spec RafB = new{m: bool};
5 spec RafC = new{m: bool};
6
7 spec MEK1 = new{S218:bool ,S222:bool };
8 spec MEK2 = new{S218:bool ,S222:bool };
9
10 spec ERK1 = new{T185:bool ,Y187:bool };
11 spec ERK2 = new{T185:bool ,Y187:bool };
12
13 spec PP2A1 = new{}, PP2A2 = new{}, MKP3 = new{};
14
15 spec NRaf = RafA:{m} or RafB:{m} or RafC:{m};
16 spec NMEK = MEK1:{ S218 ,S222 } or MEK2:{ S218 ,S222 };
17 spec NERK = ERK1:{ T185 ,Y187 } or ERK2:{ T185 ,Y187 };
18
19 mapk(Ras , NRaf , NMEK, NERK, PP2A1 , PP2A2 , MKP3)

```

line 1 in Listing 3.2.3 expands to a choice of three species, namely $\text{MEK}\{\text{S218},\text{S222}\}$ in isolation or in complex with $\text{Raf}\{m\}$ or PP2A2 , and line 2 expands to a choice of two species, namely ERK in isolation or in complex with MKP3 .

A reaction with nondeterministic species semantically gives rise to a number of parallel reactions, one for each possible choice of species. For example, the first reaction $k + s \rightarrow k-s$ in the ph module now gives rise to a parallel composition of 6 reactions:

```

1 MEK{ S218 ,S222 } + ERK -> MEK{ S218 ,S222 }-ERK |
2 MEK{ S218 ,S222 }-Raf{m} + ERK -> MEK{ S218 ,S222 }-Raf{m}-ERK |
3 MEK{ S218 ,S222 }-PP2A2 + ERK -> MEK{ S218 ,S222 }-PP2A2-ERK |
4
5 MEK{ S218 ,S222 } + ERK-MKP3 -> MEK{ S218 ,S222 }-ERK-MKP3 |
6 MEK{ S218 ,S222 }-Raf{m} + ERK-MKP3 -> MEK{ S218 ,S222 }-Raf{m}-ERK-MKP3 |
7 MEK{ S218 ,S222 }-PP2A2 + ERK-MKP3 -> MEK{ S218 ,S222 }-PP2A2-ERK-MKP3

```

The two other reactions in the ph module have similar expansions, and the ph module invocation hence results in a total of 18 reactions.

Nondeterminism for species variant combinatorial explosion The above example demonstrates how nondeterminism can be used to drastically reduce the size of programs in which the combinatorial explosion is contextual. Nondeterminism can also be used to handle species variant combinatorial explosions, i.e. arising from variants of individual proteins which largely react in the same way. For example, Raf has three variants RafA, RafB and RafC; MEK has two variants MEK1 and MEK2; and ERK has two variants ERK1 and ERK2 [72]. As mentioned in Chapter 2, rule-based languages such as κ and BioNetGen do *not* per se have any dedicated means of handling this source of combinatorial explosion, but a recent meta-language provides some level of syntactical support [29]. Indeed this meta-language, together with our need to handle contextual combinatorial explosion, are the two motivating factors for the introduction of nondeterminism into LBS. Listing 3.2.4 shows how the MAPK cascade module can be used with species variants. In order to be of interest, one would expect that *some* reactions distinguish between the variants, but we omit this aspect in the present example.

Each individual member of the nondeterministic species in lines 15-17 is given a separate annotation at time of definition rather than at time of module invocation. The reason is that the members do not have any common atomic species, and in general they may not have common modification sites either, so it is necessary to identify the atomic species and sites to be mapped from the corresponding formals on an individual basis; recall here that e.g. RafA:{m} and RafB:{m} abbreviate respectively RafA:RafA{m} and RafB:RafB{m}, so the annotations do indeed differ between different members of the nondeterministic species. For that reason also semantically, annotations are associated with species rather than with module invocations, and the mappings between formals and actuals are maintained locally within individual species rather than globally. Invocation of the mapk module results in 102 reactions as opposed to the 30 reactions in the original model.

The mechanism of nondeterministic selection An important point about Listing 3.2.4 is that nondeterministic species are expanded *at the level of reactions* and not at the level of modules. This means that the mapk module invocation is *not* equivalent to a parallel composition of module invocations for each choice of species. Such an interpretation would result in 360 reactions, but $360 - 102 = 258$ of these would be duplicates, effectively adding up the rates of duplicated reactions, which is certainly not what we intend. In this respect LBS has a *call-by-name* semantics. On the other

hand, species identifiers are resolved at time of module invocation where actual species parameters are evaluated to sets of species values, so in this respect LBS has a *call-by-value* semantics.

An additional subtlety arises in reactions where the same species occurs multiple times as a reactant or product. Consider for example the following:

```

1 spec a1 = new{} , a2 = new{} , b = new{};
2 spec a = a1 or a2;
3 a + a + b -> a-a-b

```

There are two reasonable, but very different, possibilities for expansion of the reaction.

The first requires that the same choice for *a* is made within the scope of the reaction:

```

1 a1 + a1 + b -> a1-a1-b |
2 a2 + a2 + b -> a2-a2-b

```

The second possibility allows different copies of the same identifier to take different values, but with the correspondence between the occurrences on the reactant and product sides being preserved:

```

1 a1 + a1 + b -> a1-a1-b |
2 a1 + a2 + b -> a1-a2-b |
3 a2 + a1 + b -> a2-a1-b |
4 a2 + a2 + b -> a2-a2-b

```

The correct expansion depends on the specific application. Although the first may seem most appropriate in the general case, the second is for example useful for modelling the combinatorial dimerisation of different variants of ErbB receptors [29] (note that two of the resulting reactions are equivalent, effectively duplicating their rates). In order to cater for these different possibilities, LBS has two reaction arrows. The basic reaction arrow, \rightarrow , which has been used in the examples so far, results in the first expansion, and we call this a *selection arrow*. The double-headed reaction arrow, $\rightarrow\rightleftharpoons$, results in the second expansion, and we call this an *identity-preserving arrow*.

In order to give a uniform semantical treatment of nondeterminism, and to enable other expansions than the two described above, LBS has a dedicated **force** operator for forcing nondeterministic choice. For example, the program:

```

1 spec a = force a1 or a2;
2 P

```

results in a parallel composition of *P* with a binding of *a* to *a1* in one parallel component and *P* with a binding to *a2* in the other parallel component. Reactions using either of

the two arrows are then derived forms expressed in terms of the force operator and a third *deterministic reaction arrow*, \Rightarrow , which requires that reactants and products do not contain nondeterministic species. For example, the program:

```
1 a + a + b -> a-a-b
```

abbreviates the program:

```
1 spec a = force(a);
2 spec b = force(b);
3 a + a + b => a-a-b
```

and the program:

```
1 a + a + b ->> a-a-b
```

abbreviates the program:

```
1 spec a1 = force(a);
2 spec a2 = force(a);
3 spec b1 = force(b);
4 a1 + a2 + b1 -> a1-a2-b1
```

Nondeterministic species expressions which are not bound to identifiers are not allowed in reactions with any of the three arrows. This is because the implicit forcing is done on identifiers rather than on general species expressions, which allows the identity between different occurrences of e.g. the species a to be preserved after nondeterministic selection.

Restricting combinations of nondeterministic choices In the above examples we assume that reaction rates are independent of the combinations of nondeterministic choices. Some combinations may however need different rates than others, and some combinations may not react at all. Consider the following example of a nondeterministic degradation reaction:

```
1 spec a1 = new{}, a2 = new{}, b1 = new{}, b2 = new{};
2 spec s = (a1 or a2)-(b1 or b2);
3 s ->{0.1}
```

A rate of 0.1 is given in curly brackets. Suppose now that the combination with a1 and b1 takes place at the lower rate of 0.01. This can be expressed in a compact manner using the *restriction operator*, **not**, which semantically is interpreted as a set difference operator:

```

1 spec a1 = new{}, a2 = new{}, b1 = new{}, b2 = new{};
2 spec s = (a1 or a2)–(b1 or b2) not a1–b1;
3 s –>{0.1} |
4 a1–b1 –>{0.01}

```

Different rates can also be selected based on the state of modification of species by using variables and *conditionals* in rate expressions. The following example shows reactions which have a high rate when either an a or b is phosphorylated, and a low rate otherwise:

```

1 spec a1 = new{m: bool}, a2 = new{m: bool};
2 spec b1 = new{n: bool}, b2 = new{n: bool};
3 spec s = (a1{m=$x} or a2{m=$x})–(b1{n=$y} or b2{n=$y});
4 s –>{if $x or $y then 0.1 else 0.01}

```

In non-quantitative cases, conditionals can be used directly in *boolean guards* of reactions. Following on from the above example, the nondeterministic reaction below expands to only the reactions in which either an a or b is phosphorylated:

```
s –> if $x or $y
```

The **not** operator can only be used to restrict combinations within *the same nondeterministic species*. In contrast, conditionals can be used to restrict combinations across different nondeterministic species, although only based on the internal state of species. One can however encode species identity as internal state and thus use conditionals to restrict different combinations of species based on identity.

3.2.3 A Modification Site Type with Binding

All of the previous examples have used boolean modification site types for representing phosphorylation state, and complexes have been understood as multisets of modified atomic species. But LBS allows arbitrary choices of modification site types. One such choice leads to a representation of complexes at the more detailed level of bindings. This in turn enables a translation to κ and BioNetGen, and we give a formal definition of the former translation in Chapter 6.

In contrast to solutions using nondeterminism, the support for handling combinatorial explosion in κ and BioNetGen is largely transparent to the modeller. The supporting tools work without generating the set of all possible fully specified complexes. This set may be too large for analysis or simulation using traditional methods; it may

even be infinite. From the perspective of κ and BioNetGen, their integration into the LBS framework allows features such as compartments and modularity to be exploited.

Listing 3.2.5 shows how the idea of explicit binding can be incorporated into LBS. The *concrete* syntax of species modifications differs from that used in the previous examples in order to closer match that of κ and BioNetGen as outlined in Section 2.2; the distinguished binding, ?, used in the last four lines, is a wild card meaning *any binding*. We preserve the explicit separation of different complexes in reactants and products, hence following the syntax of BioNetGen more than that of κ ; semantically, whether or not separate complexes can match a single connected complex depends on whether a translation to κ or BioNetGen is chosen.

The crucial point, however, is that the *abstract* syntax remains unaffected by the introduction of explicit binding: species modifications can still be considered values of an appropriate type assigned to modification sites. As shown in the species definitions in lines 17-23, the type is here called **binding**. Its values are formally *pairs* of an internal state and a binding.

The default value, given to sites which are not mentioned explicitly such as the site m of Ras in line 25, is the pair consisting of the wild cards *any internal state* and *any binding*. Furthermore, either or both of the internal and binding states may be omitted. If the binding state is omitted, such as for the site m of Raf in line 26, the *unbound* state is assumed. More subtly, however, if the internal state is omitted, the *identity* internal state is assumed which has the effect of preserving any previous internal state. This is important in e.g. the second reaction in the ph module, line 3, which binds the site m of the kinase k; it must do so without affecting the original modification of this site. For example, the module invocation in line 26 specifies that the site must be phosphorylated, while the module invocation in line 25 specifies that the site can have any internal state.

The outlined choices of default values for sites not mentioned explicitly and for derived forms achieve the standard semantics of κ in scenarios where species are only ever bound to identifiers at the time of creation. The choices also result in a reasonable semantics in the more general case, which includes parameterised modules, as shown in the example.

Note, finally, that in contrast to earlier examples without explicit binding, choices must now be made about which specific sites species bind on. For example, Raf uses the site m, which is itself subject to phosphorylation in the MAPK cascade, to bind MEK, whereas MEK uses an additional site, m, which is never phosphorylated, to bind ERK.

Listing 3.2.5: A MAPK cascade model at the level of protein binding.

```

1  module ph( spec k:{m} , s:{m} ) {
2      k{m} + s{m~u} -> k{m!1}-s{m~u!1} |
3      k{m!1}-s{m~u!1} -> k{m!1}-s{m~p!1} |
4      k{m!1}-s{m~p!1} -> k + s{m~p}
5  };
6
7  module dph( spec p:{m} , s:{m} ) {
8      p{m} + s{m~p} -> p{m!1}-s{m~p!1} |
9      p{m!1}-s{m~p!1} -> p{m!1}-s{m~u!1} |
10     p{m!1}-s{m~u!1} -> p{m} + s{m~u}
11 };
12
13 module cycle( spec k:{m} , p:{m} , s:{m} ) {
14     ph( k:{m} , s:{m} ) | dph( p:{m} , s:{m} )
15 };
16
17 spec Ras = new{m:binding};
18 spec Raf = new{m:binding};
19 spec MEK = new{S218:binding , S222:binding , m:binding};
20 spec ERK = new{T185:binding , Y187:binding};
21 spec PP2A1 = new{m:binding};
22 spec PP2A2 = new{m:binding};
23 spec MKP3 = new{m:binding};
24
25 cycle(Ras:{m} , PP2A1:{m} , Raf:{m}) |
26 cycle(Raf{m~p}:{m} , PP2A2:{m} , MEK{S222~u?}:{S218}) |
27 cycle(Raf{m~p}:{m} , PP2A2:{m} , MEK{S218~p?}:{S222}) |
28 cycle(MEK{S218~p? , S222~p? , m}:{m} , MKP3:{m} , ERK{Y187~u?}:{T185}) |
29 cycle(MEK{S218~p? , S222~p? , m}:{m} , MKP3:{m} , ERK{T185~p?}:{Y187})

```

Listing 3.2.6: An extension of the MAPK program in Listing 3.2.4 (not repeated here) with variations for generating one semantical object for each possible initial condition.

```

1 ... |
2 (init RafA 500 || PNil) |
3 (init RafB 500 || PNil) |
4 (init RafC 500 || PNil) |
5
6 (init MEK1 500 || PNil) |
7 (init MEK2 500 || PNil) |
8
9 (init ERK1 500 || PNil) |
10 (init ERK2 500 || PNil)

```

We also note that only LBS programs which are written using the binding modification site types can be translated to meaningful κ programs. The problem of translating general LBS programs in which complexes are represented as multisets is an interesting one that we have left for future work.

3.2.4 Model Variation

Given an LBS program it is sometimes of interest to vary it in a number of ways and examine the resulting effect on behaviour. In support of a structured approach to variations, LBS has a *variation operator*, \parallel , which semantically gives the union of its operands, i.e. programs evaluate to *sets* of semantical objects. Listing 3.2.6 shows how the variation operator can be used to generate a semantical object for each possible combination of the given initial conditions of species variants in the MAPK cascade program.

Initial condition statements such as `init RafA 500` are first-class programs and specify a given initial population or concentration for a species. If no initial conditions are specified in a program, the 0 initial population or concentration is assumed for all participating species. The distinguished program **PNil** is a neutral element under parallel composition, i.e. the axiom $P \parallel \text{PNil} = P$ holds for all programs P ; also the distributivity axiom $P_1 \parallel (P_2 \parallel P_3) = (P_1 \parallel P_2) \parallel (P_1 \parallel P_3)$ holds for all programs P_1 , P_2 and P_3 . Hence the parallel composition shown above is a power set construction and expands to a variation composition of all $2^7 = 128$ possible combinations of initial conditions in parallel with the program represented by dots in line 1, in this case the previously

Listing 3.3.1: An LBS program for a scaffolded MAPK cascade in yeast.

```

1 spec Fus3 = new{p:bool}, Ste7 = new{p:bool}, Ste11=new{p:bool};
2 spec Ste5 = new{p:bool}, Ste20 = new{}, Gbg = new{};
3
4 Ste5 + Ste11 -> Ste5-Ste11 as a;
5 Ste7 + Fus3 -> Ste7-Fus3 as b;
6 a + b -> a-b as c;
7 c + Gbg -> c-Gbg as d;
8 d + Ste20 -> d-Ste20 as e;
9
10 e -> e<Ste11{p}> as f;
11 f -> f<Ste7{p}> as g;
12 g -> g<Fus3{p}>

```

defined MAPK cascade.

3.3 A Scaffolded MAPK Cascade

The CBS scaffolded MAPK cascade model suffers from redundancy due to frequent repetition of large complexes. In this section we show how the model can be written more concisely in LBS through the use of species expressions. We also show how a notion of output species parameters can be used to link the scaffolded MAPK cascade module with a separate module for scaffold formation in a natural manner.

3.3.1 Species Expressions

New species, species identifiers and complexes are technically considered species expressions. Species identifiers can be bound to any species expressions, not just the new atomic ones, allowing large complexes to be defined once and used repeatedly. Species expressions also include a construct for *updating* the modification state of atomic species inside a complex. We illustrate this in Listing 3.3.1 which gives a more concise version of the CBS scaffolded MAPK cascade previously shown in Listing 2.5.3 on page 27.

The first two lines consist of new species definitions as before, but now some of the species are defined with a modification site called p of boolean type. Lines 4-8 represent scaffold formation, but now the intermediate complexes are bound to identifiers

using the `as` keyword. This *in-line* approach to binding is an abbreviation for binding a species expression to an identifier, then using the identifier in subsequent reactions, so e.g. the program:

```
1 Ste5 + Ste11 -> Ste5-Ste11 as a; ...
```

is an abbreviation for the program:

```
1 Ste5 + Ste11 -> Ste5-Ste11 |
2 spec a = Ste5-Ste11;
3 ...
```

Reactions which have in-line definitions are composed in *sequence*, using the ; operator rather than the parallel one, as the order of such reactions matters since identifiers defined in one reaction can only be used in the following ones.

The species bound to `e` in line 8 is the full scaffold complex in its unphosphorylated form. Lines 10-12 represent the actual MAPK cascade. In line 10, the complex bound to `e` becomes the same complex, but updated by changing the phosphorylation state of site `p` in `Ste11` to true. The result is then bound to a new identifier, `f`. The last two lines follow a similar pattern. When updates are made on atomic species we use an abbreviation and write e.g. `Fus3{p}` instead of `Fus3<Fus3{p}>`; this abbreviation has been used extensively in the previous section but is not used in the above example.

The reactions in the above LBS program avoid the redundancy which impairs the reactions in the corresponding CBS program. This improves readability. It also facilitates the process of program revision since adding e.g. a new phosphorylated site to the definition of `Ste5` only involves a subsequent change to the first reaction. Contrast this to the corresponding CBS program in which the same revision requires two changes in each of the eight reactions.

There are two further, perhaps less commonly used, species expression operators which enable complex species to be taken apart. Assuming the definition of `g` given above, the *selection* expression `g.Ste7` results in the atomic species from `g` identified by `Ste7`; the *removal* expression `g\Ste7` results in the complex species `g` without `Ste7`. Hence the reaction `g -> g.Ste7 + g\Ste7` represents dissociation of `Ste7` from `g` and could in this case be written explicitly, if more laboriously, as follows:

```
1 Fus3-Ste7{p}-Ste11{p}-Ste5-Ste20-Gbg ->
2   Ste7{p} + Fus3-Ste11{p}-Ste5-Ste20-Gbg
```

However, the species selection and removal operators are needed language constructs, not just notational conveniences. A species identifier used as the target of the selection

Listing 3.3.2: A modular LBS program for scaffold formation and a scaffolded MAPK cascade in yeast.

```

1 spec Fus3 = new{p:bool}, Ste7 = new{p:bool};
2 spec Ste11 = new{p:bool}, Ste5 = new {p:bool};
3
4 module formation(specout e : Fus3–Ste7–Ste11–Ste5) {
5   spec Ste20 = new{}, Gbg = new{};
6   Ste5 + Ste11 → Ste5–Ste11 as a;
7   Ste7 + Fus3 → Ste7–Fus3 as b;
8   a + b → a–b as c;
9   c + Gbg → c–Gbg as d;
10  d + Ste20 → d–Ste20 as e
11 };
12
13 module mapk(spec a : k1{m}–k2{m}–k3{m}; specout d : a) {
14   a → a<k3{m=tt}> as b;
15   b → b<k2{m=tt}> as c;
16   c → c<k1{m=tt}> as d
17 };
18
19 formation(spec link1);
20 mapk(link1 : Fus3{p}–Ste7{p}–Ste11{p}, spec link2);
21 ...

```

and removal operators could be a formal parameter and, as a consequence of subtyping, its complete make-up in terms of atomic species may not generally be known.

If the complex bound to *g* is a homo-multimer and has several copies of e.g. Ste7, then selection and removal operate on all copies. This convention also applies to the update operator: if the target of an update contains multiple copies of the given species name, all copies are updated accordingly. It is however possible to make distinctions between different copies of the same species in homo-multimers, and we give an example of this when presenting the formal semantics of LBS.

3.3.2 Output Species Parameters

Manipulations of large complexes are often spread across multiple modules. Sometimes there is a natural input-output relationship between these modules where a species which is constructed in one module may be the starting point for further manipulation

in another. This applies for example to the scaffolded MAPK cascade program in Listing 3.3.1 which can benefit from a decomposition into two modules, one for scaffold formation, and one for the actual MAPK cascade. The fully formed scaffold in its unphosphorylated state can be considered as an output of the first module and as an input to the second. Although one could simply pass this connecting species as a common parameter to both modules, this would involve the entire scaffold to be written out at the time of module invocation, thus repeating the definitions already given during scaffold formation. In order to avoid this, we introduce the notion of *output species*, and a modular version of the yeast MAPK cascade using this idea is shown in Listing 3.3.2.

The first two lines define four new species while the remaining two species used in the program are defined locally in the formation module. The formation module has a single formal parameter which specifies that the species *e* defined in the module body is given as an output, and the associated annotation specifies that the output contains the species Fus3, Ste7, Ste11 and Ste5. In fact the output also contains Ste20 and Gbg, but these are not exposed, which gives rise to a notion of subtyping similar to that of standard species parameters.

The mapk module has a parameter *a* and also an output species parameter *d* which is defined in the module body and is specified by the annotation to contain at least the species of *a*. In line 19 the formation module is invoked and results in a binding of the identifier link1 to the output scaffold species. In line 20 this is passed on as a parameter to the mapk module which in turn results in a binding of the identifier link2 to the phosphorylated scaffold. In the full model of the yeast pheromone pathway, Ste5 dissociates from the scaffold link2 resulting from the MAPK cascade. We can deduce by inspection of the program that the complex bound to link2 does indeed contain the species Ste5, since link1 contains Ste5 and link2 contains at least the same species as link1.

3.4 Case Study: The Yeast Pheromone Pathway

3.4.1 Overview of the Pathway

The yeast pheromone pathway controls the cell mating response to pheromone signals. An informal pictorial diagram of the pathway, divided into 7 modules, is presented in [51]. Two of the modules involve scaffold formation and a MAPK cascade similar to those depicted in Figure 2.5.4 and modelled in LBS in the previous section. Two

other modules, upstream of the MAPK cascade, are concerned with receptor activation and a G-protein cycle; the remaining modules are concerned with down-stream effects on gene expression. An ODE model of the pathway is also presented in [51]. It is based on a total of 47 reactions and 35 species.

3.4.2 The LBS Model

The corresponding LBS model is listed in Appendix A. It is composed from 7 modules corresponding to those of the original informal diagram, but it also includes two other small, nested modules. One of these (lines 107-109) is a parameterised module with a single reaction for complex degradation, and this is invoked four times. There is otherwise no reuse of modules in this model.

In addition to demonstrating how modularity naturally reflects the informal reasoning of biologists, the LBS model benefits from compartments and inline species definitions, and from output species parameters for connecting modules and hiding local species definitions following the approach outlined in the previous section. The model also uses three features of LBS not yet introduced, namely *enzymatic reactions*, *reversible reactions* and *general rate expressions*. These are all demonstrated by lines 83-84, repeated below:

```

1 rate v46 = k46 * ( Fus3{p}^2 / (4^2 + Fus3{p }^2) );
2 Fus3{p} ~ Sst2 <->[v46]{k47} Sst2{p}
```

A general rate expression is bound to the rate identifier v46. The following reversible reaction uses this general rate expression for the forward direction, and the mass-action constant k47 for the backward direction; expressions enclosed in square brackets are interpreted as general rates, and expressions enclosed in curly brackets are interpreted as rate constants. The tilde symbol, \sim , is used for enzymatic reactions, in this case with Fus3{p} as the enzyme.

3.4.3 Model Validation

The LBS model was validated by a translation to SBML and subsequently to ODEs using the COPASI tool [46]. The resulting ODEs were determined by manual inspection to coincide with the published ODEs. As an additional step of validation, the SBML model was simulated and the resulting graphs were determined to match the published graphs by visual inspection.

3.5 Case Study: The ErbB Pathway

3.5.1 Overview of the Pathway

The ErbB pathway controls cell division and has been widely studied due to its implication in human cancers [90]. The pathway includes four receptors, ErbB1 to ErbB4, which can dimerise after ligand binding. ErbB1 binds e.g. the ligand EGF, ErbB3 and ErbB4 bind e.g. the ligand HRG, and ErbB2 has no known ligand but nevertheless plays a role in dimerisation with other receptors. Dimerised receptors bind to scaffolds and activate down-stream MAPK and PI3K/Akt cascades, which in turn result in a change of gene expression.

An ODE model of the ErbB pathway is presented in [21]. In addition to receptor binding and dimerisation, scaffold formation, MAPK and PI3K/Akt cascades, it includes receptor internalisation. It does however not include gene activation. It is nevertheless one of the largest published ODE models with 499 species and 828 reversible reactions. These large numbers arise mainly from species variant combinatorial explosion with respect to the ErbB1-ErbB4 receptors, and from contextual combinatorial explosion in e.g. scaffold formation.

3.5.2 The LBS Model

A corresponding LBS model is listed in Appendix B. It contains a total of 197 reactions, which is a substantial reduction from the 828 reactions in the original model. The reduction is achieved through the use of nondeterminism and, to a lesser extent, through the use of modification site variables and parameterised modules. Modification site variables are used in lines 426-429. The model has a total of 26 module definitions and 31 module invocations.

Reusable modules for phosphorylation/dephosphorylation cycles are exploited in the MAPK and PI3K/Akt cascades (lines 744-800 and 969-1015) following the approach shown in Section 3.2; these reusable modules are however not shared between the two cascades due to differences in the way that phosphorylation is modelled. Reusable modules are also exploited for reactions which take place in two different compartments, namely in the plasma membrane and in the endosomal membrane (lines 439-452 and 634-667). Finally, a reusable module is used for reactions which take place with different rates for two different reactants (lines 942-962).

The remaining modules are used for structuring the model as in the yeast pheromone

pathway model, and the main body of the model (lines 1076-1090) consists of a large parallel composition of module invocations.

The model uses two language constructs not previously introduced. One is used for writing homomultimers as illustrated e.g. by the species expression $(\text{EGF} - \text{ATP} - \text{ErbB1})$ in line 412 which is simply the homodimer of $(\text{EGF} - \text{ATP} - \text{ErbB1})$. Another construct allows the renaming of species names and modification site names in species using the $::$ operator. This can e.g. be used to refer to all members of a nondeterministic species with a common name, or to distinguish between different members of a homomultimer. The former is illustrated by the definition in line 305:

```
spec ErbB234 = (ErbB2:{ p1 } or ErbB3:{ p1 } or ErbB4:{ p1 })::ErbB234{ p1 };
```

All the members of the nondeterministic species bound to ErbB234 can now be referenced collectively through the name ErbB234, which is conveniently chosen to coincide with the identifier to which the expression is bound. This renaming construct can be defined as a derived form using modules as we show in Subsection 4.5.2.

3.5.3 The Modelling Process and Model Validation

The original ErbB model is available as supplementary material of [21] in an SBML file; this was the starting point for developing the LBS model. The SBML model essentially encapsulates a list of species and reactions, but the species are encoded as identifiers which bear no resemblance to their human-readable, biological names. For example, the EGF species is called “mw07c8092f_eb20_4e3f_968c_9ae4601fa697”. Fortunately, the human-readable names of species are included as annotations in the SBML file.

The first step of the modelling process was to implement a tool for translating the annotated SBML file into an LBS program consisting of one big parallel composition of reactions. A parallel composition of this size has in turn required an optimisation of the LBS compiler with tail recursion in order to avoid a stack overflow.

There are however inconsistencies in the SBML annotations: some species which have identical human-readable names, or more precisely, represent the same multisets of atomic species, have different identifiers in the SBML file. The tool for translating the SBML file into LBS registers such inconsistencies and compensates by adding a distinguished atomic species, called COPY, to one of the clashing species. The tool verifies that there is a one-to-one correspondence between the resulting LBS species names and the SBML species identifiers.

The second step was to manually inspect the LBS reactions in which species were adjusted with the COPY species. In some cases the inconsistencies are due to mistakes in the SBML annotations. For example, there are two distinct SBML species identifiers which are both annotated with the human-readable name PI3K, but one of these should instead be annotated with the name ErbB4{p1}–ErbB2{p1}–Gab1{p1}–GAP–Grb2–PI3K. In other cases, the inconsistencies are due to two complex species with the same multiset representation having different binding structure. These inconsistencies can be resolved by e.g. using modification sites of the **binding** type introduced in Subsection 3.2.3, but we have not done this because of time constraints. Further information about the correct binding structure may also be needed, perhaps in dialogue with the authors of the original model who have already been helpful in answering our questions. Hence some of the species in the LBS model in Appendix B still contain the COPY atomic species.

After these manual adjustments of the LBS model, a second tool was implemented and used to check that the one-to-one correspondence between LBS species and SBML species identifiers is preserved. The tool relies on the reactions in the LBS model and the SBML file to be given in the same order; without this assumption, the verification problem may be intractable (the general graph isomorphism problem, for example, has no known polynomial-time algorithm).

The third step was to add structure to the parallel compositions of reactions by using species expressions, nondeterminism and modularity. Since this alters the ordering of reactions compared to the original SBML file, a third tool was implemented to verify that the revised program has the same “normal form” as the parallel composition of reactions resulting from the second step; the normal form of an LBS program is here the expansion to a single, ordered parallel composition.

The fourth and final step was to adjust a small number of reactions containing species that are marked as having constant concentrations in the original SBML model. There is no corresponding feature in LBS, so the adjustment was done by manually adding the relevant species as products in reactions where the species occur as reactants.

The resulting LBS program is the one listed in Appendix B. It corresponds to the original SBML model by construction and by the three automated validation steps. As an additional validation step, the model was simulated and the resulting graphs were determined to coincide with the graphs from a simulation of the original SBML model by visual inspection. We note that in contrast, an entirely manual reproduction of the

model, without any tool support or automated validation steps, would be extremely difficult to get right.

The resulting LBS model is highly structured compared to the list of reactions captured by the original SBML model. More can however be done once the COPY species have been eliminated since these prevent further symmetries from being exploited using nondeterminism. Parts of the LBS model remain relatively unstructured for other reasons. There are several blocks consisting of reactions which appear to be similar, yet they are not amenable to abstraction using nondeterminism because of small deviations which appear to be arbitrary. Such blocks of reactions are marked with comments, preceded by //, in the LBS model. One example is the definition of the nondeterministic species in line 700. The receptor ErbB4 is not included in this nondeterministic expression because the corresponding reaction involves the world compartment, rather than the plas compartment which is used in the reactions for the other receptors. Such deviations may be due to biological reality, but they may also be due to errors in the model and should hence be investigated. Shedding light onto potential errors is one important benefit of a structured approach to modelling: deviations and inconsistencies are difficult to detect from an unstructured list of reactions involving very large complexes.

Chapter 4

The Abstract Syntax of LBS

The previous chapter gave an informal introduction to the *concrete* syntax and main features of LBS. This chapter formally defines the *abstract* syntax of LBS which forms the basis of the general semantics given in the next chapter. The formal definition of the concrete syntax and its mapping into the abstract syntax are omitted, since both can be deduced without surprises from the examples and from the abstract syntax.

In order to achieve our aim of generality, the abstract syntax is parameterised on a set of *modification site types* ρ and *modification site expressions* e_m . We divide the language into four main syntactic categories, for compartments, species, programs and definitions, and consider each in turn. But first we introduce the notation used in this chapter and the following.

4.1 Notation

We let \mathbb{R} denote the set of *real numbers* and \mathbb{N} denote the set of *natural numbers*. We write \underline{x} for *lists*, $\underline{x}.i$ for the *ith element* (starting from 1) of a list, $|\underline{x}|$ for the *length* of a list and ε for the *empty* list. When a list should be thought of as representing a set, we write \underline{x} instead of \underline{x} . The set of *indices* of a list \underline{x} is $\{i \mid 1 \leq i \leq |\underline{x}|\}$. The *sublist* of \underline{x} consisting of the elements at some subset I of the indices of \underline{x} is written $\underline{x}.I$. The *Cartesian product* $\underline{x} \times_{\circ} \underline{y}$, where \circ is a pairing operator on elements of the respective lists, is the list of length $|\underline{x}| \cdot |\underline{y}|$ s.t. $(\underline{x} \times_{\circ} \underline{y}).((i-1)|\underline{x}| + j) \stackrel{\Delta}{=} \underline{x}.i \circ \underline{y}.j$. The *concatenation* of lists \underline{x} and \underline{y} is written $\underline{x}\underline{y}$, and the *prefix* and *postfix* of an element a to a list \underline{x} are written $a\underline{x}$ and $\underline{x}a$, respectively.

We write $\{x_i\}_{i \in I}$ for a finite *indexed set* and omit I and/or i and write $\{x_i\}_I$, $\{x_i\}$ or $\{x\}$ when they are understood from the context. The *power set* of a set S is written

2^S . The set of *multisets* of a set S is denoted by $MS(S)$ and is defined as the set of total functions from S to the natural numbers, i.e. $MS(S) \stackrel{\Delta}{\simeq} S \rightarrow \mathbb{N}$. We adopt the usual multiset notation and write e.g. $x + 2 \cdot y$ for the multiset containing the element x and two copies of y , and we write $MS(\underline{x})$ for the multiset representation of a list \underline{x} . We also use the standard notation $\prod_{i \in I} X_i$ for dependent sets.

We write $x \stackrel{\Delta}{\simeq} y$ for definitions where x equals y if y is defined, and where x is undefined otherwise. When a notion of well-typedness applies to y , we furthermore write $x \stackrel{\Delta_t}{\simeq} y$ for definitions where x equals y if y is defined and well-typed, and where x is undefined otherwise.

Partial finite *functions* $f : X \hookrightarrow_{\text{fin}} Y$ are denoted by finite indexed sets of pairs $\{x_i \mapsto y_i\}$ where $f(x_i) = y_i$. The *empty* function is correspondingly denoted by \emptyset . The *domain of definition* and *image* of a function f are denoted by $\text{dom}(f)$ and $\text{im}(f)$, respectively. For functions f and g we define the *update* of f by g , written $f\langle g \rangle$, as follows:

$$f\langle g \rangle(x) \stackrel{\Delta}{\simeq} \begin{cases} f(x) & \text{if } x \in \text{dom}(f) \setminus \text{dom}(g) \\ g(x) & \text{if } x \in \text{dom}(g) \end{cases}$$

If g consists of a single binding $x \mapsto y$ we write $f\langle x \mapsto y \rangle$ instead of $f\langle \{x \mapsto y\} \rangle$. We specify the type of a partial function f by writing $f(x) = y$ where x and y are given variables ranging over two sets; these sets are then understood to form the domain and image of f .

When an element of a list or an indexed set is referred to without explicit quantification in a semantical definition, the index is assumed to be universally quantified over a set which is understood from the context. Under such circumstances we often omit the index and write e.g. x instead of $\underline{x}.i$. If \circ is an operation on the elements of lists \underline{x} and \underline{y} both of length n , we write $\underline{x} \circ \underline{y}$ for the list of length n in which the i th element is $\underline{x}.i \circ \underline{y}.i$.

4.2 Compartments

4.2.1 Compartment Expressions

The abstract syntax for basic compartment expressions is shown in Table 4.2.1, where id_c ranges over the set of *compartment identifiers* and $w \in \mathbb{R}$ ranges over compartment volumes. New compartments are created using the new compartment expression which

Table 4.2.1: The abstract syntax for compartment expressions.

$e_c ::=$	COMPARTMENT EXPRESSION
new comp vol w inside e_c	NEW COMPARTMENT
\top	WORLD COMPARTMENT
id_c	COMPARTMENT IDENTIFIER
1_c in e_c	NIL COMPARTMENT

explicitly records a parent compartment and a volume. In cases where a compartment is used at the top level, the world compartment can be specified as a parent, hence allowing compartment hierarchies to be terminated. A compartment is generally used in multiple contexts by binding it to an identifier at time of creation.

The nil compartment functions as a neutral element for the composition of compartment lists. It is paired with a parent compartment, which is necessary for type-checking of compartment hierarchies. Nil compartments can for example be used to decrease the depth of a module hierarchy when passed as parameters to modules.

Although the world compartment figures as a general compartment in the abstract syntax, it is only intended for use as a parent of new compartments and of the nil compartment. It is not intended for use in e.g. reactions, and its proper usage is enforced in the semantics for programs. One could enforce this intended usage syntactically by introducing separate production rules for top level and nested new compartment and nil compartment expressions. However, whether or not a compartment features at the global top level of a program is not generally known at time of definition: take for example compartment definitions inside a module, where the parent compartment may be a formal parameter.

4.2.2 Derived Compartment Expressions

The volume in new compartment expressions may be omitted, in which case a default volume of 1.0 is assumed. The parent compartment in new compartment and nil compartment expressions may also be omitted, in which case the world parent compartment is assumed.

Table 4.3.1: The abstract syntax for boolean expressions.

$e_b ::=$	BOOLEAN EXPRESSION
tt	TRUE
ff	FALSE
$x : \text{bool}$	TYPED VARIABLE
$e_b \text{ or } e'_b \mid \text{not } e_b$	BOOLEAN OPERATORS

4.3 Species

4.3.1 Modification Site Expressions

Recall that the abstract syntax for species expressions is parameterised on a set of modification site types ρ and a set of modification site expressions e_m . Since boolean expressions are of widespread practical use as demonstrated in the examples, we assume that the set of modification types contains the boolean type **bool**, and that the set of modification site expressions contains the boolean expressions e_b generated by the grammar in Table 4.3.1 where x ranges over the set of *variables*.

The boolean expressions contain the usual **tt/ff** base values and a minimal set of connectives from which the full set of boolean connectives can be defined as derived forms in the usual manner. Variables are used to create species expressions which can match multiple concrete species. We assume that the set of variables is closed by prefixing of underscore-terminated binary strings, i.e. that b_x is a variable for all $b \in \{0, 1\}^*$; this is needed to confine variables to their appropriate namespace when defining the semantics. The type annotation of variables is likewise used for technical convenience.

4.3.2 Species Expressions

The abstract syntax for species expressions is shown in Table 4.3.2, where n_s ranges over the set of *species names*, n_m ranges over the set of *modification site names* and id_s ranges over the set of *species identifiers*. Species names identify atomic species independently of any modification sites, while species identifiers refer to possibly complex species including both the names and modification states of atomic species in the com-

plex. We assume for technical reasons that both the set of species identifiers and the set of binary strings is contained in the set of species names. We assume furthermore, as for variables, that the set of species identifiers is closed by prefixing of underscore-terminated binary strings, i.e. that b_id_s is also a species identifier for all $b \in \{0, 1\}^*$.

The grammar distinguishes between species expressions e_s and extended species expressions e_{s+} which add the new atomic species expression. This is because new species expressions only make sense in the context of definitions, where the resulting new species value can be bound to an identifier. Species bound to an identifier can then be used in multiple contexts and given an initial population through the construct given in the abstract syntax for programs. Technically, separating out the new species expression alleviates the need to consider fresh names in the semantics for the remaining expressions and for certain cases of programs; this significantly simplifies the presentation.

New atomic species are created by specifying a name and a type consisting of a partial finite function from modification site names to modification site types. The modification sites are assigned default expressions appropriate for the corresponding type, e.g. **ff** in the case of the **bool** type. In contrast to new compartment expressions, a new species expression explicitly includes a species name. Often this name is the same as the identifier to which the new species expression is assigned, which is reflected in a derived form of definitions. Although semantically the underlying unique species name will be freshly generated, the specified name is used to identify specific atomic species in subsequent species selection, removal and update expressions. Species names rather than general species expressions are used here for two reasons. First, the update expression updates a specific atomic species in a complex. Second, atomic species names are local to a species, meaning that the same atomic species name in two different species may map to different underlying fresh species names. This is used to cater for nondeterminism in the context of parametric types in species parameters of modules. Similar considerations of nondeterminism apply to compartments, which are only used in species expressions indirectly through compartment identifiers rather than through general compartment expressions.

The species annotations necessary to match the names and sites of actual parameters to those of formal parameters are handled in the abstract syntax for species expressions rather than in the abstract syntax for module invocation in programs. This too is because of nondeterminism where separate annotations may be required for each member of a nondeterministic species expression as shown in Listing 3.2.4.

Table 4.3.2: The abstract syntax for species expressions.

$e_{s+} ::= e_s$	EXTENDED SPECIES EXPRESSION
new n_s, σ	NEW ATOMIC SPECIES
$e_s ::=$	SPECIES EXPRESSION
$id_c[e_s]$	LOCATION
$e_s - e'_s$	COMPOSITION
$e_s.id_c[n_s]$	SELECTION
$e_s \setminus id_c[n_s]$	REMOVAL
$e_s \langle id_c[n_s, \alpha] \rangle$	UPDATE
$e_s \text{ or } e'_s$	CHOICE
$e_s \text{ not } e'_s$	CHOICE RESTRICTION
$e_s : \xi$	ANNOTATION
id_s	IDENTIFIER
$\mathbf{0}_s$	NIL
$\xi ::= \underline{id_c[n_s, n_m]}$	ANNOTATION
$\sigma ::= \{n_m \mapsto \rho\}$	MODIFICATION TYPE
$\alpha ::= \{n_m \mapsto e_m\}$	MODIFICATION ASSIGNMENT

4.3.3 Derived Species Expressions

The derived form $n.e_s$, where $n \in \mathbb{N}$, is used for writing homomultimers and abbreviates:

$$\underbrace{e_s - \cdots - e_s}_{n \text{ times}}$$

Two further derived forms, used repeatedly in the previous chapter, allow updates and annotations of atomic species without having to repeat atomic species names. Specifically, the expressions:

$$id_s\{\alpha\} \quad \text{and} \quad id_s : \underline{n_m}$$

abbreviate respectively the expressions:

$$id_s\langle \epsilon[id_s, \alpha] \rangle \quad \text{and} \quad id_s : \underline{\epsilon[id_s, n_m]}$$

4.4 Programs

4.4.1 Basic Programs

The abstract syntax for programs is shown in Table 4.4.1, where $n \in \mathbb{N}$, id_m ranges over the set of *module identifiers* and id_a ranges over the set of *algebraic rate function identifiers*. Definitions, ranged over by D , are treated in the next section. Module invocations include actual parameters for compartments, species, rates and output species, and as already pointed out, the annotations of actual species parameters necessary to match the formal parameters are handled in the abstract syntax for species expressions.

Reaction rate expressions can either be constant rate expressions, given inside curly brackets, or general algebraic rate expressions, given inside square brackets. Note that constant rate expressions are represented by algebraic expressions in the abstract syntax because this allows for a uniform treatment of defined constants and conditionals. Semantically however, constant rate expressions are required to evaluate to constants.

Algebraic rate expressions include rate constants, compartments and species, where the latter two are interpreted as respectively a volume and a population. Algebraic rate expressions also include a number of basic functions and arithmetic operators which feature regularly in the biological literature; these are inspired by similar features found in BioPEPA. Custom rate functions which are parameterised on compartments, species

Table 4.4.1: The abstract syntax for basic programs.

$P ::=$	PROGRAM
$\underline{n} \cdot e_s \Rightarrow^{e_r} \underline{n}' \cdot e'_s \text{ if } e_b$	REACTION
$\mathbf{0}_p$	NIL PROGRAM
$P \mid P'$	PARALLEL COMPOSITION
$P \parallel P'$	VARIATION COMPOSITION
$id_c[P]$	LOCATED PROGRAM
$D ; P$	DEFINITION
$id_m(\underline{e_c}; \underline{e_{s+}}; \underline{e_a}; \mathbf{out} \underline{id_s}) ; P$	MODULE INVOCATION
$id_s = \mathbf{force} e_s ; P$	NONDETERMINISTIC SELECTION
$\mathbf{init} e_s = r$	INITIAL POPULATION
$e_r ::=$	RATE EXPRESSION
$\{e_a\}$	CONSTANT RATE EXPRESSION
$[e_a]$	ALGEBRAIC RATE EXPRESSION
$e_a ::=$	ALGEBRAIC RATE EXPRESSION
r	CONSTANT
id_c	VOLUME
e_s	POPULATION
$\mathbf{if} e_b \mathbf{then} e_a \mathbf{else} e'_a$	CONDITIONAL
$id_a(\underline{e_c}; \underline{e_s}; \underline{e_a})$	FUNCTION INVOCATION
$\exp(e_a) \mid \log(e_a) \mid \sin(e_a) \mid \cos(e_a)$	STANDARD FUNCTIONS
$e_a + e'_a \mid e_a - e'_a$	ARITHMETIC OPERATORS
$e_a \times e'_a \mid e_a / e'_a \mid e_a \hat{e}'_a$	

Table 4.4.2: The abstract syntax for derived programs.

$P ::= \dots$	DERIVED PROGRAM
$e''_s \sim n \cdot e_s A^{e_r} n' \cdot e'_s \text{ if } e_b; P$	GENERAL REACTION
$e''_s \sim n \cdot e_s A_2^{e_r, e'_r} n' \cdot e'_s \sim e'''_s \text{ if } e_b, e'_b; P$	GENERAL REVERSIBLE REACTION
$A ::=$	REACTION ARROWS
\Rightarrow	DETERMINISTIC ARROW
\rightarrow	SELECTION ARROW
$\rightarrow\!\!\!$	IDENTITY-PRESERVING ARROW
$A_2 ::= \Leftrightarrow \mid \leftrightarrow \mid \rightleftharpoons$	REVERSIBLE REACTION ARROWS
$e_s ::= \dots$	DERIVED SPECIES EXPRESSIONS
$e_s \text{ as } id_s$	IN-LINE DEFINITION

and algebraic rate expressions can be defined and invoked repeatedly. These also allow the definition of common rate functions for e.g. Michaelis-Menten or Hill kinetics. Conditionals enable different rates to be chosen depending on the state of modification of reactants as recorded by match variables as outlined in Subsection 3.2.2.

Only the simplest possible reaction is included in the abstract syntax for programs. Species expressions are assumed to be deterministic, requiring any nondeterministic selection to be carried out in advance through the use of the force operator; there are no in-line species definitions; and there are no reversible or enzymatic reactions.

4.4.2 Derived Programs

More complicated reactions are generated by the abstract syntax for derived programs in Table 4.4.2, all of which are defined in terms of basic programs in the next chapter. The dots in the grammar indicate extension of the grammar for basic programs.

Derived programs include the two additional reaction arrows which cater for nondeterministic species and which implicitly force nondeterministic selection in two different manners, as exemplified in Subsection 3.2.2. Enzymatic reactions are given by a list of enzymes to the left of the tilde symbol. All types of reactions can be reversible with any combination of constant rate expressions and general algebraic rate expressions for each of the two directions. Finally, species expressions in derived reactions may contain in-line definitions which go into scope in the sequential program following the reaction.

Further derived forms arise by omitting the enzyme or boolean expression parts of reactions. The absence of an enzyme part is understood as an enzyme part with an empty list of species, and the absence of a boolean expression part is understood as a boolean expression part with the expression `tt`. Stoichiometry in reactions can be omitted, in which case stoichiometry 1 is assumed. Finally, the sequential programs following reactions and module invocations can be omitted when there are no in-line species definitions or output species parameters, respectively. In these cases the nil sequential program is assumed.

4.5 Definitions

4.5.1 Basic Definitions

The abstract syntax for definitions is shown in Table 4.5.1 and should be self-explanatory. Formal species parameters have annotations ξ as defined in the abstract syntax for species expressions. Together with the corresponding annotation of actual species parameters, this is sufficient to construct a mapping that allows use of the species inside the module body.

4.5.2 Derived Definitions

Recall from the abstract syntax for species that a new species expression includes a species name. But in many cases this name is identical to the identifier that the new species expression is bound to. The name can then be omitted, i.e. the expression:

$$id_s = \mathbf{new} \sigma$$

abbreviates the expression:

$$id_s = \mathbf{new} id_s, \sigma$$

Table 4.5.1: The abstract syntax for definitions.

$D ::=$	DEFINITION
$id_s = e_{s+}$	SPECIES
$id_c = e_c$	COMPARTMENT
$id_a(id_c; id_s : \xi; id_a) = e_a$	FUNCTION
$id_m(id_c; id_s : \xi; id_a; \mathbf{out} id'_s : e_s) = P$	MODULE

This is the reason that the set of species names is assumed to contain the set of species identifiers.

Another derived form allows the renaming of species names and modification site names in species as demonstrated in Subsection 3.5.2. Formally, the expression:

$$id_s = e_s :: \xi; P$$

abbreviates a module definition and invocation:

$$id_m(id_s : \xi) = P; id_m(e_s)$$

for some arbitrary module identifier, id_m .

Chapter 5

The General Semantics of LBS

This chapter defines a denotational framework for compositionally assigning semantical objects such as Petri nets to LBS programs. Our aim is to abstract away from the specific kind of semantical object under consideration.

Assumptions We achieve our aim of abstraction by assuming a given *concrete semantics* structure $(S, |_S, \mathbf{0}_S, R_S, I_S)$ consisting of:

- A set S of *semantical objects* ranged over by O .
- A partial binary *composition function* $|_S$ on semantical objects.
- A distinguished *nil semantical object* $\mathbf{0}_S \in S$.
- A partial *reaction assignment function* of the form $R_S(R, b) = O$ assigning a semantical object to a given reaction R , named b , in a normal form, defined below (b is used to name e.g. Petri net transitions).
- A partial *initial condition assignment function* of the form $I_S(v_{\text{gns}}, r) = O$ assigning a semantical object to an initial population or concentration r of species v_{gns} in a ground normal form, defined below.

The last implies that semantical objects have a representation of initial conditions, e.g. an initial marking in the case of a Petri net. Specific examples of concrete semantics are given in the next chapter.

Recall that the abstract syntax is parameterised on modification site types ρ and modification site expressions e_m . We assume the following relations and functions pertaining to these:

- A *typing relation* of the form $e_m : \rho$ giving types to modification site expressions. This is used for determining well-typedness of species expressions.
- A *default expression function* of the form $\text{default}(\rho) = e_m$ giving default expressions to types. This is used for assigning modification site expressions to unassigned sites in species expressions.
- A *variable function* of the form $FV(e_m) = \{x_i : \rho_i\}$ giving the set of (typed) variables in a modification site expression. This is used for assigning semantical objects to reactions in some of the concrete semantics.
- An *expression denotation function* of the form $\llbracket e_m \rrbracket_m \Gamma_x = v_m$ for evaluating a modification site expression to a value v_m in a given set $\llbracket \rho \rrbracket_t$ where $e_m : \rho$, given a *variable environment* of the form $\Gamma_x(x : \rho) = v_m$ assigning values $v_m \in \llbracket \rho \rrbracket_t$ to typed variables. This is used for assigning semantical objects to reactions in some of the concrete semantics.
- An *update function* of the form $e_m \langle e'_m \rangle = e''_m$ for updating one modification site expression with another. This is used in the semantics of species update expressions. While this operation is trivial for e.g. boolean expressions in which the original expression is simply disregarded, the situation is more subtle for e.g. binding expressions.
- A *seal function* of the form $\text{seal}(e_m, b) = e'_m$ for confining names in modification site expressions to a *namespace* given by a binary string $b \in \{0, 1\}^*$. The namespace is used to avoid capture of e.g. variables in actual species parameters when used inside the body of a module.

In the case where only the boolean modification site type is given, and where the set of modification site expressions is hence the set of boolean expressions, the above functions can be defined as follows:

- $e_m : \mathbf{bool}$ for all e_m
- $\text{default}(\mathbf{bool}) \stackrel{\Delta}{=} \mathbf{ff}$
- $FV(e_m)$ is defined inductively as follows:
 - $FV(\mathbf{tt}) \stackrel{\Delta}{=} \emptyset$
 - $FV(\mathbf{ff}) \stackrel{\Delta}{=} \emptyset$

- $FV(x : \mathbf{bool}) \stackrel{\Delta}{\simeq} \{x : \mathbf{bool}\}$
- $FV(e_b \mathbf{or} e'_b) \stackrel{\Delta}{\simeq} FV(e_b) \cup FV(e'_b)$
- $FV(\mathbf{not} e_b) \stackrel{\Delta}{\simeq} FV(e_b)$

- $\llbracket e_b \rrbracket_m \Gamma_x = \llbracket e_b \rrbracket_b \Gamma_x$ is defined inductively as follows:
 - $\llbracket \mathbf{tt} \rrbracket_b \Gamma_x \stackrel{\Delta}{\simeq} \mathbf{tt}$
 - $\llbracket \mathbf{ff} \rrbracket_b \Gamma_x \stackrel{\Delta}{\simeq} \mathbf{ff}$
 - $\llbracket x : \mathbf{bool} \rrbracket_b \Gamma_x \stackrel{\Delta}{\simeq} \Gamma_x(x)$
 - $\llbracket e_b \mathbf{or} e'_b \rrbracket_b \Gamma_x \stackrel{\Delta}{\simeq} \begin{cases} \mathbf{tt} & \text{if } \llbracket e_b \rrbracket_b \Gamma_x = \mathbf{tt} \text{ or } \llbracket e'_b \rrbracket_b \Gamma_x = \mathbf{tt} \\ \mathbf{ff} & \text{otherwise} \end{cases}$
 - $\llbracket \mathbf{not} e_b \rrbracket_b \Gamma_x \stackrel{\Delta}{\simeq} \begin{cases} \mathbf{tt} & \text{if } \llbracket e_b \rrbracket_b \Gamma_x = \mathbf{ff} \\ \mathbf{ff} & \text{otherwise} \end{cases}$
- $e_m \langle e'_m \rangle = e'_m$
- $seal(e_m, b)$ is defined inductively as follows:
 - $seal(\mathbf{tt}, b) \stackrel{\Delta}{\simeq} \emptyset$
 - $seal(\mathbf{ff}, b) \stackrel{\Delta}{\simeq} \emptyset$
 - $seal(x : \mathbf{bool}, b) \stackrel{\Delta}{\simeq} b _ x : \mathbf{bool}$
 - $seal(e_b \mathbf{or} e'_b, b) \stackrel{\Delta}{\simeq} seal(e_b, b) \mathbf{or} seal(e'_b, b)$
 - $seal(\mathbf{not} e_b, b) \stackrel{\Delta}{\simeq} \mathbf{not} seal(e_b, b)$

Overview As for the abstract syntax, the semantics is presented in four sections each treating one of the four syntactic categories in detail. An overview of the denotation functions and associated symbols is given in Tables 5.0.1 and 5.0.2. The environments are partial finite functions from appropriate sets of identifiers and other relevant parameters to appropriate sets of values. For the rate function and module environments these values are themselves functions mapping actual parameters to some other appropriate values. The binary string b is a parameter of some of the denotation functions which pass it on to the $seal$ and R_S functions. Freshness of b is ensured by appropriate extensions as denotation functions are computed. This follows the approach of CBS, except that in CBS, fresh names are computed bottom-up, whereas we compute them top-down in order to avoid some unpleasant technicalities.

Table 5.0.1: Denotation functions.

Function signature	Denotation of
$\llbracket e_c \rrbracket_c \Gamma_c, b = v_c$	Compartment expressions
$\llbracket e_s \rrbracket_s \Gamma_c, \Gamma_s = \underline{v}_s$	Species expressions
$\llbracket e_{s+} \rrbracket_{s+} \Gamma_c, \Gamma_s, b = \underline{v}_s$	Extended species expressions
$\llbracket e_a \rrbracket_a \Gamma_c, \Gamma_s, \Gamma_a, \underline{v}_c = v_a$	Algebraic rate expressions
$\llbracket e_m \rrbracket_m \Gamma_x = v_m$	Modification site expressions
$\llbracket P \rrbracket_p \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b, \underline{v}_c = \{(O_i, \Gamma_{soi})\}$	Programs
$\llbracket D \rrbracket_d \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b = \Gamma'_c, \Gamma'_s, \Gamma'_a, \Gamma'_m, \Gamma_{so}$	Definitions

Table 5.0.2: Symbols in the denotation function signatures.

Symbol	Description
v_c	Compartment value
v_s	Species value
v_a	Algebraic rate value
v_m	Modification site value
O	Semantical object
b	Binary string
Γ_c	Compartment environment
Γ_s	Species environment
Γ_a	Algebraic rate function environment
Γ_x	Variable environment
Γ_m	Module environment
Γ_{so}	Output species environment

5.1 Compartments

5.1.1 Compartment Values

We let n_c range over a given set of *compartment names* which is assumed to include the set of binary strings and contain the nil compartment name $\mathbf{1}_c$. In contrast to compartment identifiers, which are language constructs used for binding compartment values, compartment names other than $\mathbf{1}_c$ are used to uniquely and globally identify a compartment. Compartment values are of the following form:

$$\begin{aligned} v_c ::= & \quad \text{COMPARTMENT VALUE} \\ | (n_c, w, v'_c) & \quad \text{NESTED COMPARTMENT} \\ | \top & \quad \text{WORLD COMPARTMENT} \end{aligned}$$

We let V_c denote the set of all compartment values generated by this grammar. Parent compartments v'_c are recorded as values rather than names, since the name $\mathbf{1}_c$ does not identify a value uniquely. Compartment volumes w represent the volume of a compartment in the “biological sense” that the volume of a child compartment does not count towards the volume of its enclosing parent.

5.1.2 The Denotation Function

A *compartment environment* is a partial finite function of the form $\Gamma_c(id_c) = v_c$ mapping compartment identifiers to compartment values. The denotation function for compartment expressions is of the form:

$$[\![e_c]\!]_c \Gamma_c, b = v_c$$

and is defined inductively as follows:

- $[\![\mathbf{new} \, \mathbf{comp} \, \mathbf{vol} \, w \, \mathbf{inside} \, e_c]\!]_c \Gamma_c, b \stackrel{\Delta}{\simeq} (b, w, v_c)$ where
 - $v_c \stackrel{\Delta}{\simeq} [\![e_c]\!]_c \Gamma_c, 0b$
- $[\![\top]\!]_c \Gamma_c, b \stackrel{\Delta}{\simeq} \top$
- $[\![\mathbf{1}_c \, \mathbf{inside} \, e_c]\!]_c \Gamma_c, b \stackrel{\Delta}{\simeq} (\mathbf{1}_c, 0.0, v_c)$ where
 - $v_c \stackrel{\Delta}{\simeq} [\![e_c]\!]_c \Gamma_c, b$
- $[\![id_c]\!]_c \Gamma_c, b \stackrel{\Delta}{\simeq} \Gamma_c(id_c)$

The denotation function is partial since it is not defined for identifiers which do not have bindings in the given environment. New compartments are named by the binary string argument to the denotation function. The denotation of the parent compartment is computed recursively, but with a 0 prefixed to the fresh name, hence resulting in a new fresh name. Nil compartment values are arbitrarily given the volume 0.0.

5.1.3 Well-Typedness of Compartment Value Lists

Compartments generally occur in the context of lists of other compartments, and we are only interested in such lists which respect the hierarchy captured in compartment values. Formally, we say that a *list* (n_c, w, v_c) is *well-typed* if $\underline{v}_c.i = (n_c, w, v_c).(i-1)$ for $i \in \{2 \dots |\underline{v}_c|\}$. Any other lists, including those which contain the world compartment in any other position than possibly the first, are ill-typed.

Compartment lists in turn occur in the context of sets of other compartment lists, and we are only interested in such sets where all compartment lists agree on parent compartments. To formalise this, we define a function of the form $\text{parent}(\underline{v}_c) = \{\underline{v}_{c_i}'\}$ which gives the set of legal parent compartments of a compartment list:

$$\text{parent}(\underline{v}_c) \stackrel{\Delta}{=} \begin{cases} V_c & \text{if } \underline{v}_c = \epsilon \\ \{\underline{v}_c'\} & \text{if } |\underline{v}_c| > 0 \text{ and } \underline{v}_c.1 = (n_c, w, v_c') \\ \emptyset & \text{if } |\underline{v}_c| > 0 \text{ and } \underline{v}_c.1 = \top \end{cases}$$

In words, the empty list of compartments can be put inside any compartment; a non-empty list of compartments can only be put inside the compartment specified by the first element of the list unless this is the world compartment, in which case it can be put nowhere. Formally, we then say that a *set* $\{\underline{v}_{c_i}\}$ is *well-typed* if all \underline{v}_{c_i} are well-typed and either $\text{parent}(\underline{v}_{c_i}) = \emptyset$ for all i or $\bigcap_i \text{parent}(\underline{v}_{c_i}) \neq \emptyset$.

The forest structure of well-typed sets of compartment value lists The motivation for defining well-typedness of sets of compartment value lists is that only physically meaningful compartment hierarchies should be allowed in programs. By this we mean that sets of compartment value lists should form a forest structure, here a directed acyclic graph in which each node has at most one parent.

Observe first that one can obtain a directed graph from a well-typed set of compartment value lists in which nodes are compartment values and edges are determined by left-to-right neighbourhood in lists. Formally, given a well-typed set $\{\underline{v}_{c_i}\}$ we define

$G\{\underline{v}_{c_i}\} \stackrel{\Delta}{\simeq} (V, E)$ where

$$\begin{aligned} V &\stackrel{\Delta}{\simeq} \{\underline{v}_{c_i}.j\} \\ E &\stackrel{\Delta}{\simeq} \{(\underline{v}_{c_i}.j, \underline{v}_{c_i}.(j+1))\} \end{aligned}$$

(Recall from our notational convention that the above definitions give indexed sets where i is an index into the set of compartment value lists and j is an index into list positions). We then have that these graphs are indeed forests:

Proposition 5.1.1. $G\{\underline{v}_{c_i}\}$ is a forest.

The proof is given in Appendix C.

5.1.4 Normal Forms of Compartment Value Lists

Parent compartments in compartment values are necessary for type-checking in the general semantics, and volumes are necessary in algebraic rate expressions. But from the view of any concrete semantics, we are interested in a normal form of compartment lists in which only compartment names are retained and in which the nil compartments are removed. The *normal form function* is of the form $nf(\underline{v}_c) = \underline{n}_c$ and is defined as $nf(\underline{n}_c, w, \underline{v}_c) \stackrel{\Delta}{\simeq} \underline{n}_c.I$ where $I \stackrel{\Delta}{\simeq} \{i \mid \underline{n}_c.i \neq \mathbf{1}_c\}$ if \underline{v}_c is a well-typed compartment value list and is undefined otherwise. The graph arising from the normal form of a set of compartment value lists is also a forest if all non-nil compartment values have distinct compartment names, i.e. if the same non-nil compartment name does not occur with different parents or with different volumes. This is always the case for graphs in which compartment values arise from compartment expressions.

5.2 Species

5.2.1 Species Values

Recall from the abstract syntax for species expressions that ξ is an annotation used to provide a match between actual and formal species parameters. Recall also that ρ ranges over modification site types, that n_m ranges over modification site names, and that n_s ranges over species names. *Species values* are generated by the following

grammar where we use $Q \subsetneq_{\text{fin}} \mathbb{N}$ to range over sets of list indices:

$v_s ::= v_{us}^{i:\xi}$	SPECIES VALUE
$v_{us} ::= \underline{v_c}[n_s, \alpha_\sigma]$	UNBOXED SPECIES VALUE
$\alpha_\sigma ::= \{n_m \mapsto (\rho, e_m)\}$	TYPED ASSIGNMENT
$\iota ::= \{\underline{id_c}[n_s] \mapsto (Q, \iota_m)\}$	SPECIES INTERFACE
$\iota_m ::= \{n_m \mapsto n'_m\}$	MODIFICATION SITE INTERFACE

An *unboxed species value* represents a possibly complex species by a list of located atomic species, each of which is represented by a name and a *typed assignment* mapping modification site names to pairs of modification types and expressions. *Species values* add annotations and interfaces. Annotations are as in the abstract syntax for species expressions: they are used for selecting the located atomic species and modification sites in an actual species parameter which should be mapped from the corresponding atomic species and modification sites in a formal parameter. *Interfaces* capture this mapping from formals to actuals and can hence be viewed as a product of module invocation. The need for a local mapping from formals to actuals arises from our having nondeterministic species, where different members of the set of values denoted by a nondeterministic actual species parameter may require different mappings from formals to actuals as demonstrated in Listing 3.2.4.

Interfaces map formal located names to pairs consisting of a set of position indices in the associated unboxed species values and a *modification site interface*. The sets of indices are used to cater for the general case of homo-multimers in which there are multiple instances of some atomic species. Modification site interfaces map formal modification site names to actual modification site names in the unboxed species value. Interfaces may expose only a subset of species indices in an unboxed species value, and for each exposed set of species indices, the associated modification site interface may expose only a subset of the modification sites recorded in the unboxed species value. Hence interfaces give rise to a notion of subtyping. For species values which have not been subjected to module invocations, the interface exposes all atomic species and all modification sites. Interfaces also support a notion of parametric type since they provide means of renaming atomic species and modification site names.

Examples of species values Examples of some species values arising from Listing 3.3.2 on page 46 are shown informally in Figure 5.2.1 where we let f be the pair

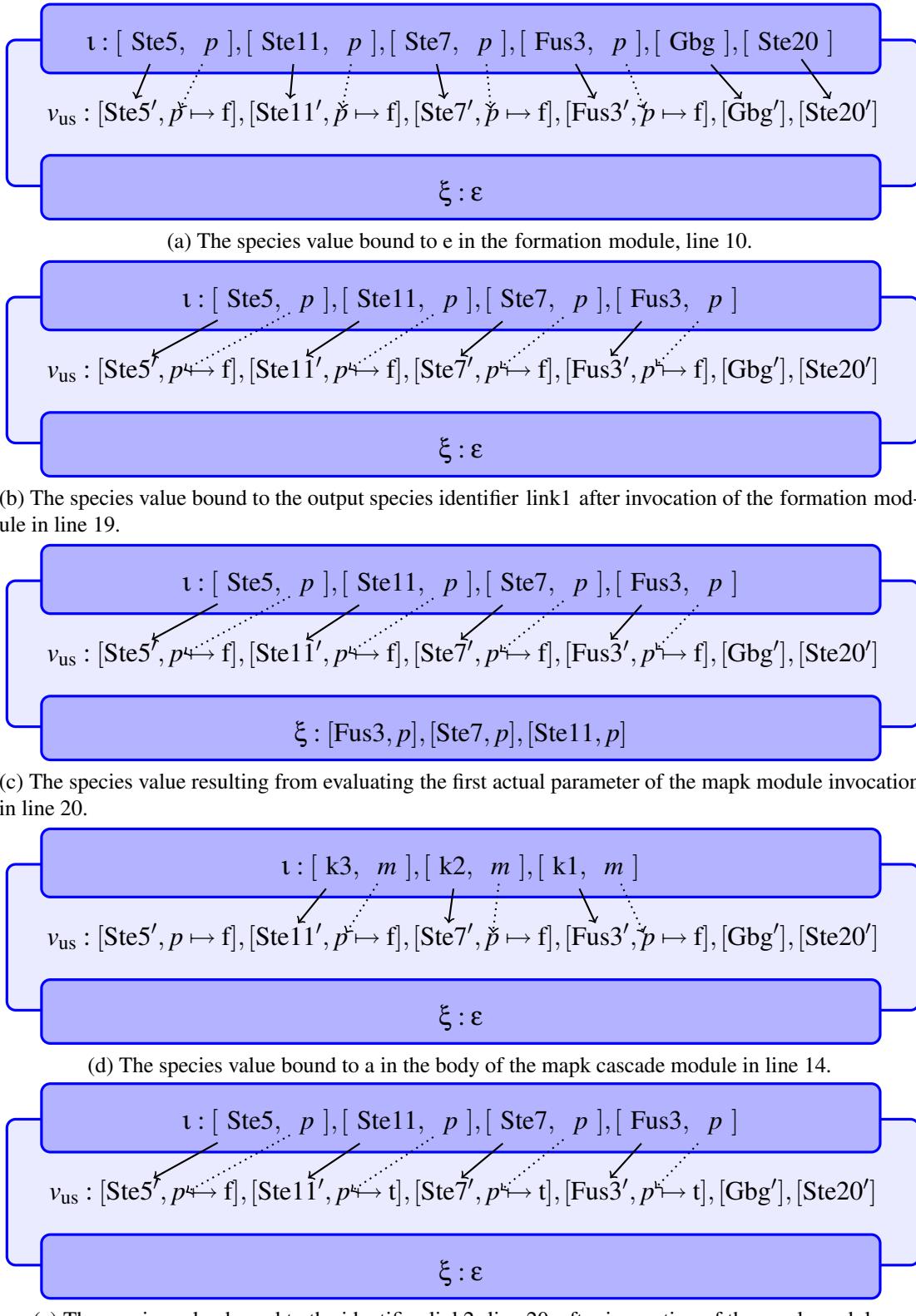


Figure 5.2.1: Examples of species values from Listing 3.3.2 represented in an informal graphical notation.

(**bool,ff**) and t be the pair (**bool,tt**). Interfaces are depicted in the top part of each figure, with solid lines representing the mapping from atomic species names to indices, and dotted lines representing the embedded mapping between modification site names. Note that none of these examples are homo-multimers, so interfaces map to singleton sets of indices. Unboxed species values are depicted in the center part of each figure, and annotations are depicted at the bottom.

Figure 5.2.1a shows a complex species value before it has been subjected to any module invocation, hence all primitive species and their modification sites are exposed by the interface. The species names in the unboxed value are primed, indicating that these are fresh. Figure 5.2.1b shows the species value after it has been output from the formation module where Gbg and Ste20 have been removed from the interface. In Figure 5.2.1c the annotation of the actual species parameter of the mapk module has been recorded in the species value. Figure 5.2.1d shows the species value after the interface has been updated based on the annotation in Figure 5.2.1c and the corresponding formal annotation, $\xi' : [k1, m], [k2, m], [k3, m]$; together these provide a mapping from e.g. k1 to Fus3, which is traced through the interface in Figure 5.2.1c down to the fourth index of the unboxed species value. The annotation has now served its purpose and is discarded. Finally, figure 5.2.1e shows the species value where three atomic species have been phosphorylated, and following output from the mapk module, the interface of this species value has been restored to the interface of the original input species value in Figure 5.2.1d.

A smaller example, which illustrates how homomultimers can be represented, is shown in Figure 5.2.2a; here the same atomic species name, s , maps to two occurrences of the same underlying fresh species name, s' . An interface may however also map different located names to indices with the same located fresh species names as shown in Figure 5.2.2b. This allows multiple instances of the same atomic species within a homo-multimer to be distinguished.

5.2.2 Well-Typedness of Species Values

For an unboxed species value $v_c[n_s, \alpha_\sigma]$ we require that the set of lists of compartment values is well-typed and hence forms a forest structure, and that assignments respect their associated type. These conditions can be phrased formally as follows:

1. $\{\underline{v_c.i}\}$ is a well-typed set of compartment value lists
2. $\forall (\rho, e_m) \in im(\alpha_\sigma). e_m : \rho$

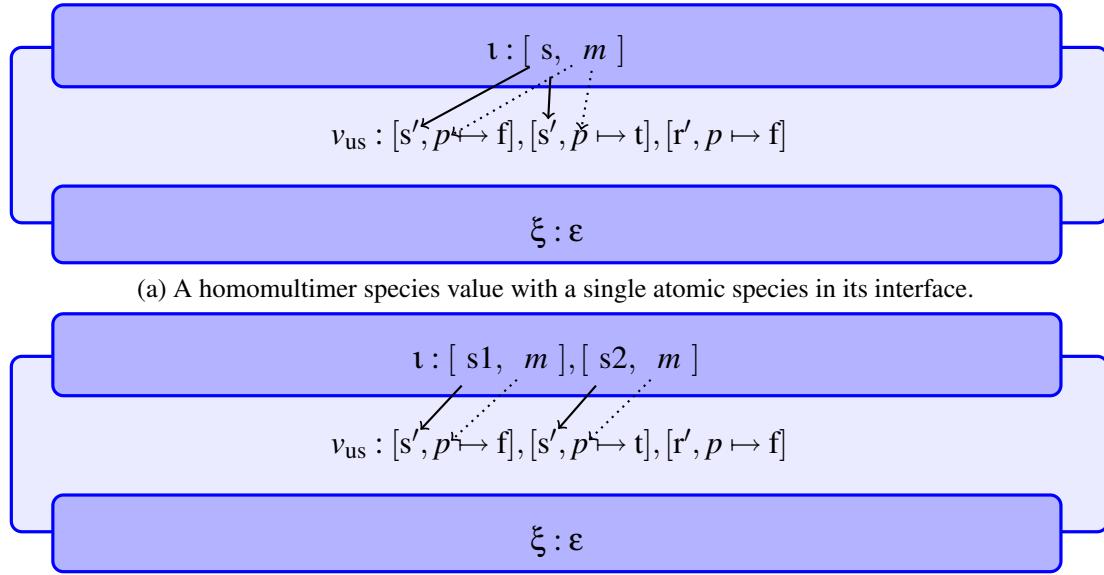


Figure 5.2.2: Examples of homomultimer species values.

For a species value $v_{us}^{\iota:\xi}$ we furthermore require that the interface maps to 1) non-empty and 2) disjoint sets of indices; that 3) all indices in a set exist in the unboxed species value and 4) contain species with identical located fresh names and modification site names; that 5) the modification site interfaces map to sites which exist in the assignments at the corresponding indices; that 6) the annotation only mentions located species and sites which exist in the interface. These conditions can be summarised formally as follows, where $v_c[n_s, \alpha_\sigma] = v_{us}$ and $\underline{id}_c[n_s, \underline{n_m}] = \xi$.

1. $\forall (Q, \iota_m) \in im(\iota). |Q| > 0$
2. $\forall l, l' \in dom(\iota). l \neq l' \Rightarrow ind(\iota(l)) \cap ind(\iota(l')) = \emptyset$
where $ind(Q, \iota_m) \stackrel{\Delta}{=} Q$
3. $\forall (Q, \iota_m) \in im(\iota). Q \subseteq_{fin} \{1, \dots, |v_{us}|\}$
4. $\forall (Q, \iota_m) \in im(\iota). \forall q, q' \in Q. v_c[n_s].q = v_c[n_s].q' \wedge t(\underline{\alpha_\sigma}.q) = t(\underline{\alpha_\sigma}.q')$
where $t(\{n_m \mapsto (\rho, e_m)\}) \stackrel{\Delta}{=} \{n_m \mapsto \rho\}$
5. $\forall (Q, \iota_m) \in im(\iota). \forall q \in Q. im(\iota_m) \subseteq_{fin} dom(\underline{\alpha_\sigma}.q)$
6. $\underline{id}_c[n_s] \in dom(\iota) \wedge \forall (Q, \iota_m). (Q, \iota_m) = \iota(\underline{id}_c[n_s]) \Rightarrow \{\underline{n_m}.i\} \subseteq_{fin} dom(\iota_m)$

5.2.3 The Denotation Function

We now turn to the semantics for species expressions. A *species environment* is a partial finite function of the form $\Gamma_s(id_s) = \underline{v}_s$ mapping species identifiers to lists of species values. The denotation function for species expressions is of the form:

$$\llbracket e_s \rrbracket_s \Gamma_c, \Gamma_s = \underline{v}_s$$

and is parametric on compartment and species environments. The denotation of a species is a *list* of species values. More than one species value may arise because of nondeterminism, and we use lists rather than sets to cater for output species in module parameters, as will be apparent in the semantics for programs; however, for most purposes we may think of these lists as sets, hence the wavy underline notation.

The definition of the denotation function for species expressions is given in the following. In order to simplify notation, we write Γ instead of Γ_c, Γ_s for cases where the environments are not used. Let us also reiterate the subtle notational convention that given e.g. a list \underline{v}_{us} we write v_{us} for $\underline{v}_{us}.i$, and that i is implicitly assumed to be universally quantified over the indices of \underline{v}_{us} in definitions; see for example the last 3 lines of the first case below.

- $\llbracket id_c[e_s] \rrbracket_s \Gamma_c, \Gamma_s \stackrel{\Delta}{\simeq} \underline{v}_{us}^{1:\xi}$ where
 - $v_c \stackrel{\Delta}{\simeq} \Gamma_c(id_c)$
 - $\underline{v}_{us1}^{\underline{v}_c 1:\xi_1} \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_s \Gamma_c, \Gamma_s$
 - $v_{us} \stackrel{\Delta}{\simeq_t} \underline{v}_c \underline{v}_{c1}[n_s, \alpha_\sigma]$ where $\underline{v}_{c1}[n_s, \alpha_\sigma] \stackrel{\Delta}{\simeq} v_{us1}$
 - $\iota \stackrel{\Delta}{\simeq} \{id_c \underline{id}_{c1}[n_s] \mapsto \iota_1(\underline{id}_{c1}[n_s]) \mid \underline{id}_{c1}[n_s] \in dom(\iota_1)\}$
 - $\xi \stackrel{\Delta}{\simeq} \underline{id}_c \underline{id}_{c1}[n_s, n_m]$ where $\underline{id}_{c1}[n_s, n_m] \stackrel{\Delta}{\simeq} \xi_1$
- $\llbracket e_{s1} - e_{s2} \rrbracket_s \Gamma \stackrel{\Delta}{\simeq_t} \underline{v}_{s1} \times_\circ \underline{v}_{s2}$ where
 - $\underline{v}_{s1} \stackrel{\Delta}{\simeq} \llbracket e_{s1} \rrbracket_s \Gamma$
 - $\underline{v}_{s2} \stackrel{\Delta}{\simeq} \llbracket e_{s2} \rrbracket_s \Gamma$
 - $v_{us1}^{\underline{v}_c 1:\xi_1} \circ v_{us2}^{\underline{v}_c 2:\xi_2} \stackrel{\Delta}{\simeq} \underline{v}_{us}^{1:\xi}$ where
 - * $v_{us} \stackrel{\Delta}{\simeq} v_{us1} v_{us2}$

$$*\ \mathfrak{i}(l) \stackrel{\Delta}{\simeq} \begin{cases} \mathfrak{i}_1(l) & \text{if } l \in \text{dom}(\mathfrak{i}_1) \setminus \text{dom}(\mathfrak{i}_2) \\ (A(Q_2), \mathfrak{i}_{m2}) & \text{if } l \in \text{dom}(\mathfrak{i}_2) \setminus \text{dom}(\mathfrak{i}_1) \wedge \\ & (Q_2, \mathfrak{i}_{m2}) = \mathfrak{i}_2(l) \\ (Q_1 \cup A(Q_2), \mathfrak{i}_m) & \text{if } l \in \text{dom}(\mathfrak{i}_1) \cap \text{dom}(\mathfrak{i}_2) \wedge \\ & (Q_1, \mathfrak{i}_{m1}) = \mathfrak{i}_1(l) \wedge (Q_2, \mathfrak{i}_{m2}) = \mathfrak{i}_2(l) \wedge \\ & \mathfrak{i}_m = \mathfrak{i}_{m1} = \mathfrak{i}_{m2} \end{cases}$$

where $A(Q) = \{q + |\nu_{us}| \mid q \in Q\}$

$$*\ \underline{\xi} \stackrel{\Delta}{\simeq} \underline{\xi}_1 \underline{\xi}_2$$

- $\llbracket e_s . \underline{id}_c[n_s] \rrbracket_s \Gamma \stackrel{\Delta}{\simeq} \nu_{us}^{1:\xi}$ where

- $\nu_{us} \stackrel{\Delta}{\simeq} \nu_{us1}^{1:\xi_1} \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_s \Gamma$
- $(Q, \mathfrak{i}_m) \stackrel{\Delta}{\simeq} \mathfrak{i}_1(\underline{id}_c[n_s])$
- $\nu_{us} \stackrel{\Delta}{\simeq} \nu_{us1} . Q$
- $\mathfrak{i} \stackrel{\Delta}{\simeq} \{\underline{id}_c[n_s] \mapsto (\{1 \dots |Q|\}, \mathfrak{i}_m)\}$
- $\xi \stackrel{\Delta}{\simeq} \xi_1 . \{q \mid \exists \underline{n}_m . \xi_1 . q = (\underline{id}_c[n_s], \underline{n}_m)\}$

- $\llbracket e_s \setminus \underline{id}_c[n_s] \rrbracket_s \Gamma \stackrel{\Delta}{\simeq} \nu_{us}^{1:\xi}$ where

- $\nu_{us} \stackrel{\Delta}{\simeq} \nu_{us1}^{1:\xi_1} \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_s \Gamma$
- $(Q, \mathfrak{i}_m) \stackrel{\Delta}{\simeq} \mathfrak{i}_1(\underline{id}_c[n_s])$
- $\nu_{us} \stackrel{\Delta}{\simeq} \nu_{us1} . (\{1 \dots |\nu_{us1}|\} \setminus Q)$
- $\mathfrak{i} \stackrel{\Delta}{\simeq} \{l \mapsto (A(Q'), \mathfrak{i}_m) \mid l \in \text{dom}(\mathfrak{i}_1) \setminus \{\underline{id}_c[n_s]\} \wedge (Q', \mathfrak{i}_m) = \mathfrak{i}_1(l)\}$

where

- * $A(Q') \stackrel{\Delta}{\simeq} \{q' - |\{q \in Q \mid q \leq q'\}| \mid q' \in Q'\}$
- $\xi \stackrel{\Delta}{\simeq} \xi_1 . \{q \mid \neg \exists \underline{n}_m . \xi_1 . q = (\underline{id}_c[n_s], \underline{n}_m)\}$

- $\llbracket e_s \langle \underline{id}_c[n_s, \alpha] \rangle \rrbracket_s \Gamma \stackrel{\Delta}{\simeq} \nu_c[n'_s, \alpha_{\sigma''}]^{1:\xi}$ where

- $\nu_c \stackrel{\Delta}{\simeq} \nu_c[n'_s, \alpha_{\sigma'}]^{1:\xi} \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_s \Gamma$
- $(Q, \mathfrak{i}_m) \stackrel{\Delta}{\simeq} \mathfrak{i}(\underline{id}_c[n_s])$

$$-\underline{\alpha_\sigma''}.q \stackrel{\Delta}{\simeq} \begin{cases} \underline{\alpha_\sigma'}.q(\alpha \circ (\iota_m^{-1})) & \text{if } q \in Q \\ \underline{\alpha_\sigma'}.q & \text{otherwise} \end{cases}$$

where

$$*\alpha_\sigma'(\alpha')(n_m) \stackrel{\Delta}{\simeq} \begin{cases} \alpha_\sigma'(n_m)\langle\alpha'(n_m)\rangle & \text{if } n_m \in \text{dom}(\alpha_\sigma') \cap \text{dom}(\alpha') \\ \alpha_\sigma'(n_m) & \text{if } n_m \in \text{dom}(\alpha_\sigma') \setminus \text{dom}(\alpha') \end{cases}$$

for all α_σ' , α' and n_m

- $\llbracket e_{s1} \text{ or } e_{s2} \rrbracket_s \Gamma \stackrel{\Delta}{\simeq} \underline{v_{s1}} \underline{v_{s2}}$ where

$$\begin{aligned} -\underline{v_{s1}} &\stackrel{\Delta}{\simeq} \llbracket e_{s1} \rrbracket_s \Gamma \\ -\underline{v_{s2}} &\stackrel{\Delta}{\simeq} \llbracket e_{s2} \rrbracket_s \Gamma \end{aligned}$$

- $\llbracket e_{s1} \text{ not } e_{s2} \rrbracket_s \Gamma \stackrel{\Delta}{\simeq} \underline{v_s}$ where

$$\begin{aligned} -\underline{v_{s1}} &\stackrel{\Delta}{\simeq} \llbracket e_{s1} \rrbracket_s \Gamma \\ -\underline{v_{s2}} &\stackrel{\Delta}{\simeq} \llbracket e_{s2} \rrbracket_s \Gamma \\ -\underline{v_s} &\stackrel{\Delta}{\simeq} \underline{v_{s1}}.\{q \mid \neg \exists q'. \underline{v_{s2}}.q' = \underline{v_{s1}}.q\} \end{aligned}$$

- $\llbracket e_s : \xi \rrbracket_s \Gamma \stackrel{\Delta}{\simeq}_t \underline{v_{us}^{1:\xi}}$ where

$$-\underline{v_{us}^{1:\xi_1}} \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_s \Gamma$$

- $\llbracket id_s \rrbracket_s \Gamma_c, \Gamma_s \stackrel{\Delta}{\simeq} \Gamma_s(id_s)$
- $\llbracket \mathbf{0}_s \rrbracket_s \Gamma \stackrel{\Delta}{\simeq} \underline{\varepsilon}$

The denotation function is partial because some species expressions do not result in lists of well-typed species values or in environments which are functions, or because some operations are undefined for some of the intermediate objects which arise. Given suitable environments, we say that a species expression is *well-typed* if its denotation is defined.

The denotation function for extended species expressions is of the form:

$$\llbracket e_{s+} \rrbracket_{s+} \Gamma_c, \Gamma_s, b = \underline{v_s}$$

It is parametric on compartment and species environments, and also on a binary string b used to create fresh names for new species. Here is the definition:

- $\llbracket e_s \rrbracket_{s+\Gamma_c, \Gamma_s, b} \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_{s, \Gamma_c, \Gamma_s}$
- $\llbracket \text{new } n_s, \sigma \rrbracket_{s+\Gamma_c, \Gamma_s, b} \stackrel{\Delta}{\simeq} \underbrace{\varepsilon[b, \alpha_\sigma]}_{\stackrel{\Delta}{\simeq}}^{1:\varepsilon}$ where
 - $\alpha_\sigma \stackrel{\Delta}{\simeq} \{n_m \mapsto (\rho, \text{default}(\rho)) \mid n_m \in \text{dom}(\sigma) \wedge \rho = \sigma(n_m)\}$
 - $\iota \stackrel{\Delta}{\simeq} \{\varepsilon[n_s] \mapsto (\{1\}, \{n_{mi} \mapsto n_{mi}\}) \text{ where } \{n_{mi} \mapsto \rho_i\} \stackrel{\Delta}{\simeq} \sigma\}$

Explanation of the denotation function In the case of located species, the compartment value assigned to the compartment identifier is looked up in the compartment environment and the species value denoting the nested species expression is obtained recursively. The denotation of the located species expression is obtained from this species value by adding the compartment value to the left of every located atomic species in the unboxed species value, by adding the compartment identifier to the left of each list of compartment identifiers in the domain of the interface, and by likewise adding the compartment identifier to each list of compartments in the annotation. Note that the interface records the compartment identifier rather than the compartment value. The expression is well-typed when the compartment identifier is defined in the given environment and when the resulting sets of compartment value lists are well-typed.

The denotation of a composite species expression is given by a Cartesian product of the denotations of the two operands. The corresponding pairing operation on species values concatenates the two unboxed species values and composes the interfaces in a manner that reflects this concatenation. The composed interface essentially maps located names to the union of indices given by the individual interfaces, with the twist that indices from the second interface are increased by the length of the first unboxed species value. This adjustment of indices is handled by an auxiliary function A . Annotations are composed simply by list concatenation. The resulting species value is well-typed when the two components agree on atomic species names and modification site interfaces for any common members of their interface.

In the case of species selection, the resulting unboxed species value is obtained by selecting the indices determined by the interface of the target species on the given located name. The resulting interface maps the given located name to a set of consecutive indices together with the original modification site interface. The resulting annotation is obtained by selecting for those entries which contain the given located name. The expression is well-typed when the located name is in the domain of the interface, which is necessary for the list selection operations to be well-defined. The

case of species removal follows a similar idea, although special care is needed for the appropriate adjustment of indices using an auxiliary function A .

In the case of species update, the interface and annotation of the denotation of the operand are preserved, and an updated unboxed species is obtained by updating the assignments at the indices with the given located name. The modification site names in the given update expression are first renamed by function composition with the inverse of the modification site interface, and the result is given as an argument to an update operation on typed assignments which is defined as an auxiliary function. Here, the assignment to modification sites which are not mentioned in the update are preserved. For sites which are both in the original assignment and in the update, the expression update function (which is a parameter of the general semantics) is used; the update function is assumed extended to pairs of modification site types and expressions. The expression is well-typed if the located species name is in the domain of the target species interface, if the domain of the update is in the domain of the relevant modification site interface, and if the update respects the relevant species types.

The remaining cases are simpler. For nondeterministic species expressions, the species value lists obtained from the denotations of the operands are simply concatenated. The species annotation expression replaces the annotation in the denotation of the nested species expression with a new annotation. For this to be well-typed, the new annotation must mention only located names and sites which exist in the domain of the interface of the operand. In the case of species identifier expressions, the corresponding value is simply looked up in the species environment which must be defined for the given identifier in order for the expression to be well-typed. Finally, the nil species evaluates to a singleton list containing just the empty species value.

For extended species expressions, the new species expression evaluates to a singleton unboxed species with a fresh species name given by the binary string parameter to the denotation function, together with a typed assignment which extends the given type with default modification expressions. The interface simply maps the given species name in the empty list of compartment identifiers to the first and only index of the unboxed species value, together with the identity interface on modification site names.

5.2.4 Normal Forms of Species Values and Further Functions

Normal form species values Interfaces, annotations and parent definitions in compartment values are needed for determining well-typedness and for making module

invocation work, as detailed in the next section, but they are not needed for normal form reactions. All that is needed here is a *normal form*:

$$v_{\text{ns}} ::= \underline{n_c}[n_s, \alpha_\sigma]$$

of species values. The *normal form function* then takes the form $\text{nf}(v_s, \underline{v_c}) = v_{\text{ns}}$ where $\underline{v_c}$ is a list of compartment values which is required to go all the way from the world compartment down to the compartment enclosing the species value. The function is used in the semantics for programs and is applied to species values in a located reaction. In the following definition we use the normal form function for compartment value lists defined in the previous section:

$$\text{nf}(\underline{v'_c}[n_s, \alpha_\sigma]^{1,\xi}, \underline{v_c}) \stackrel{\Delta}{\simeq} \underline{\text{nf}(v_c v'_c)[n_s, \alpha_\sigma]}$$

The function is defined only when $\underline{v_c} v'_c$ is a well-typed list of compartment values. Although retaining the full list of compartment names (rather than just the enclosing compartment) is unnecessary for some concrete semantics such as Petri nets, this information may be relevant in other cases. For example, it allows the compartment forest structure of programs to be obtained through the general semantical framework by defining an appropriate concrete semantics for representing and composing forests.

Ground normal form species values We introduce one further *ground normal form* of species values:

$$\begin{aligned} v_{\text{gns}} &::= \underline{n_c}[n_s, \beta_\sigma] \\ \beta_\sigma &::= \{n_m \mapsto (\rho, v_m)\} \end{aligned}$$

Here modification sites map to pairs of modification types and values, rather than to pairs of modification types and expressions with variables. Ground normal form species values are used in the semantics for programs in the case of initial conditions; indeed, as described above, the general semantics is parameterised on a function of the form $I_S(v_{\text{gns}}, r) = O$ for assigning semantical objects to initial populations or concentrations of ground normal form species values. Recall also that the general semantics is parameterised on a function of the form $\llbracket e_m \rrbracket_m \Gamma_x = v_m$ which assigns values to modification site expressions given a variable environment. This function can be extended in an evident manner to a function from normal form species values to ground normal form species values; this is needed when defining concrete semantics.

Further functions on species values We define two functions on species values which are required for the semantics of module invocations. For the first function, the intuition is that one can update an interface ι_1 given the associated annotation ξ_1 together with a second matching annotation ξ_2 obtained from a corresponding formal species parameter. The updated interface is then used to access the species within the body of the module. An example of this is shown in the transition from the species value in Figure 5.2.1c to that in Figure 5.2.1d on page 73. Formally, the function takes the form $close(v_{s1}, \xi_2) = v_{s2}$ and is defined as follows:

$$close(v_{us}^{\iota_1:\xi_1}, \xi_2) \stackrel{\Delta}{\simeq} v_{us}^{\iota_2:\epsilon}$$

where, for $\underline{id_c}[n_s, n_m]_1 \stackrel{\Delta}{\simeq} \xi_1$ and $\underline{id_c}[n_s, n_m]_2 \stackrel{\Delta}{\simeq} \xi_2$,

$$\iota_2 \stackrel{\Delta}{\simeq} \{ \underline{id_c}[n_s]_2.i \mapsto (Q, \{ n_{m2}.i.j \mapsto \iota_m(n_{m1}.i.j) \}) \mid (Q, \iota_m) = \iota_1(\underline{id_c}[n_s]_1.i) \}$$

The function is only defined if the lists ξ and ξ' have the same length, and if all of the embedded lists $n_{m1}.i$ and $n_{m2}.i$ also have the same length.

The second function enables one species value to take on the interface and annotation of another species value. This is needed in the semantics for output species in programs, and an example is shown in the transition from the species value in Figure 5.2.1a to that in Figure 5.2.1b, and from the species value in Figure 5.2.1d to that in Figure 5.2.1e. Formally, we first need two supporting functions. The first gives the located name of an unboxed species value at a given index, and the second counts the number of previous occurrences of the located species name at a given index:

$$\begin{aligned} l^q(v_{us}) &\stackrel{\Delta}{\simeq} \underline{id_c}[n_s] \text{ where } \exists \alpha_\sigma. \underline{id_c}[n_s, \alpha_\sigma] = v_{us}.q \\ c^q(v_{us}) &\stackrel{\Delta}{\simeq} |\{q' \mid l^q(v_{us}) = l^{q'}(v_{us}) \wedge q' < q\}| \end{aligned}$$

The function of interest then takes the form $adapt(v_{s1}, v_{s2}) = v_{s3}$ and is defined as follows:

$$adapt(v_{us1}^{\iota_1:\xi_1}, v_{us2}^{\iota_2:\xi_2}) \stackrel{\Delta_t}{\simeq} v_{us1}^{\iota_3:\xi_2}$$

where, for all $l \in \text{dom}(\iota_2)$:

$$\begin{aligned} (\underline{Q}_2, \iota_{m2}) &\stackrel{\Delta}{\simeq} \iota_2(l) \\ \underline{Q}_3 &\stackrel{\Delta}{\simeq} \{q \mid \exists q_2 \in Q_2. l^q(v_{us1}) = l^{q_2}(v_{us2}) \wedge c^q(v_{us1}) = c^{q_2}(v_{us2})\} \\ \iota_3(l) &\stackrel{\Delta}{\simeq} (\underline{Q}_3, \iota_{m2}) \end{aligned}$$

Here the new interface ι_3 maps a located species name l to the indices in v_{us1} which have the same located fresh name l' as the indices in v_{us2} mapped from l by ι_2 . If there are multiple choices of such indices, the index which results in the same number of previous occurrences of the fresh name l' in the two unboxed species values is chosen. The function is defined only when the resulting species value is well-typed, which is the case whenever the resulting sets of indices are non-empty and the modification site interfaces map to sites which exist in the resulting unboxed species value.

5.2.5 Species Value Design Choices

We end the treatment of species values with some remarks about possible alternative representations. First note that we choose to include modification types in species values. However, since new species values are always created with fresh names and there are no expressions which allow modification sites to change type, species values with identical names also have identical modification types. Hence it would also be possible to maintain modification types separately from species values as indeed was done in a previous version of the language [70], with the benefit of reduced redundancy. But this approach would have the downside of cluttering the presentation of the semantics with an additional environment needing to be maintained.

Alternative representations of species interfaces are also possible. We choose for example to include compartment identifiers in the renaming of interfaces, which allows compartments to differ between different members of a nondeterministic species in the same way that atomic species may differ. But interfaces could instead provide local mappings for only species and modification site names, and require compartments to be evaluated externally. This would ensure that two species with the same location in their interface are indeed in the same location. Such guarantees cannot be made when location is included in the interface.

Another design choice involves the relatively relaxed conditions on interfaces. For example, an interface may map a located name to indices in which the compartment structure is completely different, and different located names may map to indices with the same species names. The latter allows the elements of a homomultimer to be distinguished within the same species as demonstrated by the earlier example in Figure 5.2.2b. This is one reason why unboxed species values are lists rather than multisets. The ordering of atomic species within a complex is however also significant when giving a concrete semantics in terms of κ as we see in the next chapter; the concrete

semantics in terms of Petri nets, ODEs and CTMCs, on the other hand, disregard the ordering.

Annotations are maintained explicitly in species values. This incurs some overhead in the semantics for species expressions since all operations must take annotations into account. Alternatively, the interface could be represented by lists and the annotation could be captured by an appropriate ordering and restriction of the interface. This would give a more compact semantics at the cost of reduced transparency. It is however impossible for another reason pertaining to output species: these can adopt the interface of actual species values at time of module invocation using the *adapt* function, so interfaces of actual species parameters must be preserved.

5.3 Programs

5.3.1 Normal Form Reactions

Recall that the general semantics is parameterised on a structure $(S, |_S, \mathbf{0}_S, R_S, I_S)$ and that R_S is a function of the form $R_S(R, b) = O$ assigning a semantical object to a reaction R , named b , in a suitable normal form. More precisely, R takes the form:

$$\underline{n} \cdot v_{ns} \xrightarrow{v_r} \underline{n'} \cdot v'_{ns} \text{ if } e_b$$

where v_{ns} and v'_{ns} are normal form species values as defined in the semantics for species and v_r is a *rate value*, i.e. a rate expression in which species expressions have been evaluated to their normal forms, compartment expressions have been replaced by their resulting volumes, and rate function invocations have been evaluated. Rate values, and their underlying algebraic rate values, are generated by the grammar in Table 5.3.1.

The denotation function for algebraic rate expressions is of the form:

$$[[e_a]]_a \Gamma_c, \Gamma_s, \Gamma_a, \underline{v_c} = v_a$$

Here Γ_a is an *algebraic rate function environment* of the form $\Gamma_a(id_a) = f$ where f in turn is a function of the form $f(\underline{v_c}, \underline{v_s}, \underline{v_a}, \underline{v'_c}) = v_a$ mapping actual parameters, together with a list $\underline{v'_c}$ of parent compartment values at time of invocation, to algebraic rate values. The denotation function is defined below, but with some standard cases for functions and arithmetic operators omitted. We adopt a convention here and throughout where any parameters of a denotational function that are not explicitly used by a given case are represented by Γ ; in the second case below, for example, Γ hence represents the parameters Γ_s , Γ_a and $\underline{v_c}$.

Table 5.3.1: The abstract syntax for rate values.

$v_r ::=$	RATE VALUE
$\{v_a\}$	RATE CONSTANT RATE VALUE
$[v_a]$	ALGEBRAIC RATE VALUE
$v_a ::=$	ALGEBRAIC RATE VALUE
r	CONSTANT
v_{ns}	POPULATION
if e_b then v_a else v'_a	CONDITIONAL
$\exp(v_a)$ $\log(v_a)$ $\sin(v_a)$ $\cos(v_a)$	FUNCTIONS
$v_a + v'_a$ $v_a - v'_a$	ARITHMETIC OPERATORS
$v_a \times v'_a$ v_a / v'_a $v_a \hat{v}'_a$	
• $\llbracket r \rrbracket_a \Gamma \stackrel{\Delta}{\simeq} r$	
• $\llbracket id_c \rrbracket_a \Gamma, \Gamma_c \stackrel{\Delta}{\simeq} w$ where	
- $(n_c, w, v_c) \stackrel{\Delta}{\simeq} \Gamma_c(id_c)$	
• $\llbracket e_s \rrbracket_a \Gamma_c, \Gamma_s, \Gamma_a, \underline{v_c} \stackrel{\Delta}{\simeq} nf(\underline{v_s} \cdot 1, \underline{v_c})$ where	
- $\underline{v_s} \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_s \Gamma_c, \Gamma_s$	
if $ \underline{v_s} = 1$	
- $v_c \stackrel{\Delta}{\simeq} \Gamma_c(id_c)$	
- $\underline{v_s} \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_s \Gamma_c, \Gamma_s$	
- $v_a \stackrel{\Delta}{\simeq} \llbracket e_a \rrbracket_a \Gamma_c, \Gamma_s, \Gamma_a, \underline{v'_c}$	
if $ \underline{v_s} = 1$	

- $\llbracket \text{if } e_b \text{ then } e_a \text{ else } e'_a \rrbracket_a \Gamma \stackrel{\Delta}{\simeq} \text{if } e_b \text{ then } \llbracket e_a \rrbracket_a \Gamma \text{ else } \llbracket e'_a \rrbracket_a \Gamma$
- $\llbracket \exp(e_a) \rrbracket_a \Gamma \stackrel{\Delta}{\simeq} \exp(\llbracket e_a \rrbracket_a \Gamma)$
- $\llbracket e_a + e'_a \rrbracket_a \Gamma \stackrel{\Delta}{\simeq} \llbracket e_a \rrbracket_a \Gamma + \llbracket e'_a \rrbracket_a \Gamma$

The case of compartments is only defined for non-world compartment expressions, reflecting our convention that the world compartment should only be used as a parent in definitions of new compartments. The case of species is only defined when the species expression does not contain nondeterminism because any nondeterministic choice is forced in the appropriate derived forms of reactions. In the case of algebraic rate function invocation, the semantic function is looked up in the algebraic rate function environment and applied to the actual parameters after these have been evaluated.

The remaining cases simply evaluate components recursively. Note in particular that conditionals are preserved in rate values, since a full evaluation requires an assignment to variables. As for normal form species expressions, this assignment is left as a concern for the concrete semantics because certain semantical objects, such as coloured Petri nets, have their own distinct way of handing variables.

5.3.2 The Denotation Function for Basic Programs

The denotation function for basic programs is of the form:

$$\llbracket P \rrbracket_p \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b, \underline{v}_c = \{(O_i, \Gamma_{soi})\}$$

Here Γ_m is a *module environment* of the form $\Gamma_m(id_m) = g$ where g in turn is a function of the form $g(\underline{v}_c, \underline{v}_s, \underline{v}_a, \underline{id}_s, b, \underline{v}_c) = \{(O_i, \Gamma_{soi})\}$ mapping actual parameters to a set of pairs of semantical objects and *output species environments* Γ_{soi} which have the same form as species environments. Note that we obtain a *set* of semantical objects and output species environments in order to account for variation composition. The output species environments allows the formal output species, defined inside a module, to become available in the program following module invocation where they are bound to the corresponding actual output species identifiers. Note also that g is parameterised on a fresh name b and a list \underline{v}_c of parent compartments. The latter is because parent compartments for a module are determined dynamically rather than statically.

The denotation function is defined below and relies on the function

$$\delta_n^m \stackrel{\Delta}{\simeq} \{0\}^{n-1} 1 \{0\}^{m-n}$$

for constructing a binary string of length m with zeros everywhere except in the n th position which holds a one.

- $\llbracket n \cdot e_s \Rightarrow^{e_r} n' \cdot e'_s \text{ if } e_b \rrbracket_p \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b, v_c \stackrel{\Delta}{\simeq} \{(O, \emptyset)\}$ where
 - $v_r \stackrel{\Delta}{\simeq} \llbracket e_r \rrbracket_a \Gamma_c, \Gamma_s, \Gamma_a, v_c$
 - $\underline{v}_s \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_s \Gamma_c, \Gamma_s$
 - $\underline{v}'_s \stackrel{\Delta}{\simeq} \llbracket e'_s \rrbracket_s \Gamma_c, \Gamma_s$
 - $v_{ns} \stackrel{\Delta}{\simeq} nf(\underline{v}_s \cdot 1, v_c)$
 - $v'_{ns} \stackrel{\Delta}{\simeq} nf(\underline{v}'_s \cdot 1, v_c)$
 - $O \stackrel{\Delta}{\simeq} R_S(n \cdot v_{ns} \Rightarrow^{v_r} n' \cdot v'_{ns} \text{ if } e_b, b)$
 - if $|v_s| = |\underline{v}'_s| = 1$
- $\llbracket \mathbf{0}_p \rrbracket_p \Gamma \stackrel{\Delta}{\simeq} \{(\mathbf{0}_S, \emptyset)\}$
- $\llbracket P \mid P' \rrbracket_p \Gamma, b \stackrel{\Delta}{\simeq} \{(O_i|_S O'_j, \Gamma_{soi}\langle\Gamma'_{soj}\rangle)\}$ where
 - $\{(O_i, \Gamma_{soi})\} \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_p \Gamma, 0b$
 - $\{(O'_j, \Gamma'_{soj})\} \stackrel{\Delta}{\simeq} \llbracket P' \rrbracket_p \Gamma, 1b$
- $\llbracket P \parallel P' \rrbracket_p \Gamma, b \stackrel{\Delta}{\simeq} \{(O_i, \Gamma_{soi})\} \cup \{(O'_j, \Gamma'_{soj})\}$ where
 - $\{(O_i, \Gamma_{soi})\} \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_p \Gamma, 0b$
 - $\{(O'_j, \Gamma'_{soj})\} \stackrel{\Delta}{\simeq} \llbracket P' \rrbracket_p \Gamma, 1b$
- $\llbracket id_c[P] \rrbracket_p \Gamma, \Gamma_c, v_c \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_p \Gamma, \Gamma_c, v'_c v_c$ where
 - $v'_c \stackrel{\Delta}{\simeq} \Gamma_c(id_c)$
- $\llbracket id_m(e_c; \underline{e}_{s+}; e_a; \text{out } id_s); P \rrbracket_p \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b, v_c \stackrel{\Delta}{\simeq} \{(O_i|_S O'_{j_i}, \Gamma_{soi}\langle\Gamma'_{soj_i}\rangle)\}$ where
 - $m \stackrel{\Delta}{\simeq} |e_c| + |\underline{e}_{s+}| + 2$
 - $b_i \stackrel{\Delta}{\simeq} \delta_i^m$
 - $b'_j \stackrel{\Delta}{\simeq} \delta_{|\underline{e}_{s+}|+j}^m$

- $b^1 \stackrel{\Delta}{\simeq} \delta_{|\underline{e}_{s+}| + |\underline{e}_c| + 1}^m$
- $b^2 \stackrel{\Delta}{\simeq} \delta_{|\underline{e}_{s+}| + |\underline{e}_c| + 2}^m$
- $\underline{v}_s.i \stackrel{\Delta}{\simeq} [\![\underline{e}_{s+}.i]\!]_s \Gamma_c, \Gamma_s, b_i b,$
- $\underline{v}_c.j \stackrel{\Delta}{\simeq} [\![\underline{e}_c.j]\!]_c \Gamma_c, b'_j b$
- $\underline{v}_a.k \stackrel{\Delta}{\simeq} [\![\underline{e}_a.k]\!]_a \Gamma_c, \Gamma_s, \Gamma_a, \underline{v}_c$
- $\{(O_i, \Gamma_{soi})\} \stackrel{\Delta}{\simeq} \Gamma_m(id_m)(\underline{v}_c, \underline{v}_s, \underline{v}_a, id_s, \underline{v}_c, b^1 b)$
- $\{\{(O'_j, \Gamma'_{soj})\}_i\} \stackrel{\Delta}{\simeq} [\![P]\!]_p \Gamma_c, \Gamma_s \langle \Gamma_{soi} \rangle, \Gamma_a, \Gamma_m, b^2 b, \underline{v}_c$
- $[\![D ; P]\!]_p \Gamma, b, \underline{v}_c \stackrel{\Delta}{\simeq} \{(O_i, \Gamma'_{so} \langle \Gamma_{soi} \rangle)\}$ where
 - $\Gamma', \Gamma'_{so} \stackrel{\Delta}{\simeq} [\![D]\!]_d \Gamma, 0b$
 - $\{(O_i, \Gamma_{soi})\} \stackrel{\Delta}{\simeq} [\![P]\!]_p \Gamma', 1b, \underline{v}_c$
- $[\![id_s = \mathbf{force}(e_s) ; P]\!]_p \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b, \underline{v}_c \stackrel{\Delta}{\simeq} \{(O_{j_1}.1 | s \dots | s O_{j_m}.m, \emptyset)\}$ where
 - $\underline{v}_s \stackrel{\Delta}{\simeq} [\![e_s]\!]_s \Gamma_c, \Gamma_s$
 - $\{\{(O_j, \Gamma_{soj})\}_i\}.i \stackrel{\Delta}{\simeq} [\![P]\!]_p \Gamma_c, \Gamma_s \langle id_s \mapsto \underline{v}_s.i \rangle, \Gamma_a, \Gamma_m, \delta_i^{|\underline{v}_s|} b, \underline{v}_c$
- $[\![\mathbf{init} e_s = r]\!]_p \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b, \underline{v}_c \stackrel{\Delta}{\simeq} \{(O, \emptyset)\}$ where
 - $\underline{v}_s \stackrel{\Delta}{\simeq} [\![e_s]\!]_s \Gamma_c, \Gamma_s$
 - $v_{ns} \stackrel{\Delta}{\simeq} nf(\underline{v}_s.1, \underline{v}_c)$
 - $O \stackrel{\Delta}{\simeq} I_S(v_{ns}, r)$

if $|\underline{v}_s| = 1$

We furthermore define $[\![P]\!]_p \stackrel{\Delta}{\simeq} [\![P]\!]_p \emptyset, \emptyset, \emptyset, \emptyset, \varepsilon, \top$ for programs which constitute a complete model, i.e. which have no free identifiers.

Explanation of the denotation function The case of reactions relies on the given concrete semantic function for assigning a semantical object to the reaction evaluated to its normal form. This normal form reaction is in turn obtained by evaluating the species expressions to their normal forms, which involves completing the compartment

hierarchy in species values, and by evaluating rate expressions to rate values. The latter assumes the denotation function for algebraic rate expressions to be extended to rate expressions in an evident manner. There is one explicit condition for well-typedness, namely that species expressions must be deterministic, i.e. evaluate to singleton lists of species values. There is also the implicit condition that the concrete semantic function must be defined for the computed normal form reaction, which may e.g. fail if non-mass-action rates are used with a concrete ODE semantics.

The denotation of the nil program is simply the singleton set with the nil semantical object and the empty output species environment.

The denotation of a parallel composition is the pairwise composition of all semantical objects in the denotations of the operands, together with the pairwise update of output species environments from the first component with those of the second. The fresh name prefixes are extended appropriately. This case is well-typed when the composition operation, which is a parameter of the general semantic function, is defined.

The denotation of a variation composition is similar to that of parallel composition but results in a union of semantical objects rather than a Cartesian product.

In the case of located programs, the compartment identifier is looked up in the compartment environment and appended to the list of compartment values used to compute the denotation of the nested program. The denotation is defined when the compartment identifier is in the given compartment environment and when the resulting list of compartment values is well-typed.

The case of module invocation evaluates the actual parameters and passes the resulting values as parameters to the function denoting the module as given by the module environment. This function takes two additional parameters, namely the parent compartments at time of invocation and a fresh name string. From the function we obtain a set of semantical objects together with output species environments with bindings for the actual output species parameters. The sequential program is then evaluated in the species environment updated with the appropriate bindings for the output species. The result is the set of all pairwise compositions of semantical objects from the module and from the sequential program, together with the pairwise update of output species environments from the module with those from the sequential program. Hence the sequential program is treated as a parallel program with respect to semantical objects.

Special care must be taken to ensure the proper extension of fresh name strings for evaluating compartment expressions, species expressions, the module body and the

sequential program. The crucial characteristic of these strings is that none is a postfix of another, ensuring that there is no way of extending one string to match another. So far we have achieved this in the semantics of binary operators by prefixing respectively a 0 and a 1 to the fresh name string. But here we are faced with lists of expressions to be evaluated. We then achieve the desired property by letting all prefixes be of length $|e_s| + |e_{s+}| + 2$, where the plus two term accounts for the module body and for the sequential program. For the i th compartment expression we choose a prefix in which the i th symbol is 1 and the remaining symbols are 0s, and a similar construction is used for the remaining prefixes. The denotation function for module invocation is defined when the module identifier is in the given environment and the associated function is defined for the given arguments.

The case of definitions relies on the denotation function for definitions to obtain an updated collection of environments in which the sequential program following the definition is evaluated.

The case of nondeterministic selection evaluates the species expression, and for each resulting species value, it evaluates the sequential program. As for module invocation, special care is needed to ensure that the fresh name strings are extended appropriately. The resulting set of semantical objects consists of all possible compositions of semantical objects associated with each species value, and is hence effectively a Cartesian product. The output species environments resulting from repeated evaluation of the sequential program are disregarded, since there does not appear to be any meaningful way to reconcile them. They all have the same domain, but generally differ in their images, since each is a result of evaluating the same sequential program with different bindings for the forced species.

Finally, the case of initial population or concentration definitions evaluates the given species expression, obtains the corresponding normal form based on the current parent compartments, and uses the concrete semantic function to obtain a semantical object.

5.3.3 The Definition of Derived Programs

Next we define the denotation of derived forms in terms of basic programs. We start by considering in-line species definitions which intuitively give rise to a basic reaction without in-line definitions in parallel with the following program put in scope of the relevant definitions; an example of this is given in Subsection 3.3.1. The formal presen-

tation relies on an auxiliary *definition extraction function* of the form $\llbracket e_s \rrbracket_{ds} \underline{D} = e'_s, \underline{D}'$ where e_s is a derived species expression as defined in the abstract syntax for derived programs, e'_s is a basic species expression and $\underline{D}, \underline{D}'$ are lists of extracted species definitions. Selected cases of the definition are shown below; the remaining cases are similar:

- $\llbracket id_c[e_s] \rrbracket_{ds} \underline{D} \stackrel{\Delta}{\simeq} id_c[e'_s], \underline{D}'$ where
 - $e'_s, \underline{D}' \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_{ds} \underline{D}$
- $\llbracket e_s - e'_s \rrbracket_{ds} \underline{D} \stackrel{\Delta}{\simeq} e''_s - e'''_s, \underline{D}''$ where
 - $e''_s, \underline{D}' \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_{ds} \underline{D}$
 - $e'''_s, \underline{D}'' \stackrel{\Delta}{\simeq} \llbracket e'_s \rrbracket_{ds} \underline{D}'$
- $\llbracket id_s \rrbracket_{ds} \underline{D} \stackrel{\Delta}{\simeq} \begin{cases} e_s, \underline{D} & \text{if } \exists i. \underline{D}.i = (id_s = e_s) \\ id_s, \underline{D} & \text{otherwise} \end{cases}$
- $\llbracket e_s \text{ as } id_s \rrbracket_{ds} \underline{D} \stackrel{\Delta}{\simeq} e'_s, (id_s = e'_s) \underline{D}'$ where
 - $e'_s, \underline{D}' \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_{ds} \underline{D}$

We extend the function to the form $\llbracket e_s \rrbracket_{ds} \underline{D} = e'_s, \underline{D}'$ in order to rename species identifiers in reactant and product lists:

- $\llbracket \varepsilon \rrbracket_{ds} \underline{D} \stackrel{\Delta}{\simeq} \varepsilon, \underline{D}$
- $\llbracket e_s e'_s \rrbracket_{ds} \underline{D} \stackrel{\Delta}{\simeq} e''_s e'''_s, \underline{D}''$ where
 - $e''_s, \underline{D}' \stackrel{\Delta}{\simeq} \llbracket e_s \rrbracket_{ds} \underline{D}$
 - $e'''_s, \underline{D}'' \stackrel{\Delta}{\simeq} \llbracket e'_s \rrbracket_{ds} \underline{D}'$

We also assume the function extended to rate expressions in an evident manner.

The definition of identity-preserving arrows, outlined informally in Subsection 3.2.2, relies on an auxiliary *linearisation function* for renaming identifiers in species expressions in a linear manner. Informally, the renaming is such that all species identifiers in the reactants become distinct, all species identifiers in the products become distinct, and the i th occurrences of a given identifier in the original reactants and products are given the same name.

The linearisation function is of the form $\text{lin}(e_s, M) \stackrel{\Delta}{\simeq} e'_s, M'$ where M and M' are multisets of species identifiers. Two key cases of the definition are given below where, for $i \in \mathbb{N}$, $bs(i)$ is the binary string representation of i :

- $\text{lin}(id_s, M) \stackrel{\Delta}{\simeq} bs(M(id_s))_id_s, M + id_s$
- $\text{lin}(e_s - e'_s, M) \stackrel{\Delta}{\simeq} e''_s - e'''_s, M''$ where
 - $e''_s, M' \stackrel{\Delta}{\simeq} \text{lin}(e_s, M)$
 - $e'''_s, M'' \stackrel{\Delta}{\simeq} \text{lin}(e'_s, M')$

The base case prefixes an identifier with the binary string representation of the number of the identifier's previous occurrences, and adds the identifier to the multiset. The case of complex formation evaluates the first expression in the given multiset, resulting in a new multiset in which the second expression is evaluated. The remaining cases which are not shown here simply evaluate components recursively in the original multisets.

We extend the linearisation function to the form $\text{lin}(\underline{e}_s, M) = \underline{e}'_s, M'$ in order to rename species identifiers in reactant and product lists:

- $\text{lin}(\varepsilon, M) \stackrel{\Delta}{\simeq} \varepsilon, M$
- $\text{lin}(e_s \underline{e}'_s, M) \stackrel{\Delta}{\simeq} e''_s \underline{e}'''_s, M''$ where
 - $e''_s, M' \stackrel{\Delta}{\simeq} \text{lin}(e_s, M)$
 - $\underline{e}'''_s, M'' \stackrel{\Delta}{\simeq} \text{lin}(\underline{e}'_s, M')$

We also assume the linearisation function extended to rate expressions in an evident manner.

The derived forms are then defined by a denotation function of the form:

$$\llbracket P \rrbracket_{\text{dp}} = P'$$

where P is a derived form program and P' is a basic program. In the following, we assume a function of the form $\odot \underline{D}; P = P'$ which, given a list \underline{D} of definitions and a program P , gives a program P' in which the definitions in \underline{D} have been composed sequentially following the order of \underline{D} and have scope P . We assume a function of the form $\text{order}(\mathcal{D}) = \underline{D}$ which orders a set \mathcal{D} of definitions in some arbitrary but definite order. The function FS gives the set of species identifiers in a species expression and is defined along standard lines.

$$\begin{aligned}
& - \llbracket e''_s \sim n \cdot e_s \ A_2^{e_r, e'_r} \ n' \cdot e'_s \sim e'''_s \text{ if } e_b, e'_b ; P \rrbracket_{dp} \stackrel{\Delta}{\simeq} \\
& \quad \llbracket e''_s \sim n \cdot e_s \ A(A_2)^{e_r} \ n' \cdot e'_s \text{ if } e_b ; \mathbf{0}_p \rrbracket_{dp} \mid \\
& \quad \llbracket e'''_s \sim n' \cdot e'_s \ A(A_2)^{e'_r} \ n \cdot e_s \text{ if } e'_b ; P \rrbracket_{dp}
\end{aligned}$$

where

$$\begin{aligned}
& - A(\Leftrightarrow) \stackrel{\Delta}{\simeq} \Rightarrow \\
& - A(\leftrightarrow) \stackrel{\Delta}{\simeq} \rightarrow \\
& - A(\Leftrightarrow\Rightarrow) \stackrel{\Delta}{\simeq} \rightarrow
\end{aligned}$$

$$\begin{aligned}
& - \llbracket e''_s \sim n \cdot e_s \ A^{e_r} \ n' \cdot e'_s \text{ if } e_b ; P \rrbracket_{dp} \stackrel{\Delta}{\simeq} \llbracket 1 \cdot e''_s \ n \cdot e_s \ A^{e_r} \ 1 \cdot e''_s \ n' \cdot e'_s \text{ if } e_b ; P \rrbracket_{dp} \\
& - \llbracket n \cdot e_s \ A^{e_r} \ n' \cdot e'_s \text{ if } e_b ; P \rrbracket_{dp} \stackrel{\Delta}{\simeq} \llbracket n \cdot e''_s \ A^{e'_r} \ n' \cdot e'''_s \text{ if } e_b \rrbracket_{dp} \mid \odot D'' ; P
\end{aligned}$$

where

$$\begin{aligned}
& - e''_s, D = \llbracket e_s \rrbracket_{ds} \varepsilon \\
& - e'_r, D' \stackrel{\Delta}{\simeq} \llbracket e_r \rrbracket_{ds} D \\
& - e'''_s, D'' = \llbracket e'_s \rrbracket_{ds} D' \\
& - \llbracket n \cdot e_s \rightarrow^{e_r} n' \cdot e'_s \text{ if } e_b \rrbracket_{dp} \stackrel{\Delta}{\simeq} \\
& \quad \odot order\{id_s = \mathbf{force} id_s \mid id_s \in FS(e_s, e'_s)\} ; n \cdot e_s \Rightarrow^{e_r} n' \cdot e'_s \text{ if } e_b \\
& - \llbracket n \cdot e_s \rightarrow^{e_r} n' \cdot e'_s \text{ if } e_b \rrbracket_{dp} \stackrel{\Delta}{\simeq} \\
& \quad \odot order\{bs(i).id_s = \mathbf{force} id_s \mid id_s \in dom(M) \wedge i \in M(id_s)\} ; \\
& \quad \quad \quad n \cdot e''_s \Rightarrow^{e'_r} n' \cdot e'''_s \text{ if } e_b
\end{aligned}$$

where

$$\begin{aligned}
& - e'_r, M' \stackrel{\Delta}{\simeq} lin(e_r, \emptyset) \\
& - e''_s, M'' \stackrel{\Delta}{\simeq} lin(e_s, \emptyset) \\
& - e'''_s, M''' \stackrel{\Delta}{\simeq} lin(e'_s, \emptyset) \\
& - M(id_s) \stackrel{\Delta}{\simeq} \{1 \dots max(M'(id_s), M''(id_s), M'''(id_s))\}
\end{aligned}$$

The first case defines a reversible reaction as the parallel composition of two reactions, one for each direction. The second case defines an enzymatic reaction as a non-enzymatic reaction in which the enzymes are included in both the reactants and the products and hence do not get consumed. This is the interpretation needed in the yeast pheromone case study, although other choices, following e.g. Michaelis-Menten kinetics, could also be made.

The third case defines a reaction with in-line definitions as a reaction where definitions have been extracted, in parallel with the following program which is put in the scope of the extracted definitions. Note that identifiers defined in-line are expanded in the reaction rather than put in scope of the extracted definitions; this is necessary in order to obtain a meaningful interplay with nondeterministic selection.

The last two cases define nondeterministic reaction arrows in terms of the force operator and the deterministic reaction arrow. Note that nondeterministic species must be bound to identifiers in reactions in order to preserve the relationship between identical nondeterministic species in reactants and products. Reactions with explicit nondeterminism are therefore ill-typed. Note also that for reactions with the identity-preserving arrow, a given identifier should generally have the same number of occurrences in the reactants and products to obtain meaningful results, although this condition is not explicitly enforced. In particular, reactions such as $2 s \rightarrow s-s$ and $s + s \rightarrow s-s$ are *not* equivalent according to the above definition of derived forms.

The order of evaluation of derived forms is significant. Specifically, in-line species definitions are expanded before nondeterministic selection. This ensures that e.g. the program:

```
1 s + t -> s-t as a; P
```

expands correctly, i.e. to:

```
1 (spec s = force s;
2   spec t = force t;
3   s + t => s-t) | spec a = s-t; P
```

rather than to:

```
1 spec s = force s;
2 spec t = force t;
3 (s + t => s-t | spec a = s-t; P)
```

5.4 Definitions

The denotation function The denotation function for definitions updates the environments with bindings for a given definition. It takes the following form:

$$\llbracket D \rrbracket_d \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b = \Gamma'_c, \Gamma'_s, \Gamma'_a, \Gamma'_m, \Gamma_{so}$$

The output species environment is *created* by the denotation function and is always empty except for the case of species definitions, where it captures the binding for the defined species. This is in contrast to the other environments which are *updated* by the denotation function. Here is the definition:

- $\llbracket id_s = e_{s+} \rrbracket_d \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b \stackrel{\Delta}{\simeq} \Gamma_c, \Gamma_s \langle id_s \mapsto \underline{v}_s \rangle, \Gamma_a, \Gamma_m, \{id_s \mapsto \underline{v}_s\}$ where

$$- \underline{v}_s \stackrel{\Delta}{\simeq} \llbracket e_{s+} \rrbracket_{s+} \Gamma_c, \Gamma_s, b$$

- $\llbracket id_c = e_c \rrbracket_d \Gamma_c, \Gamma_c \stackrel{\Delta}{\simeq} \Gamma, \Gamma_c \langle id_c \mapsto v_c \rangle, \emptyset$ where

$$- v_c \stackrel{\Delta}{\simeq} \llbracket e_c \rrbracket_c \Gamma_c, b$$

- $\llbracket id_a(id_c; id_s : \xi; id_a) = e_a \rrbracket_d \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b \stackrel{\Delta}{\simeq} \Gamma_c, \Gamma_s, \Gamma_a \langle id_a \mapsto f \rangle, \Gamma_m, \emptyset$ where

$$- f(v_c, \underline{v}_s, v_a, \underline{v}_c') \stackrel{\Delta}{\simeq} \llbracket e_a \rrbracket_a \Gamma'_c, \Gamma'_s, \Gamma'_a, \underline{v}_c'$$

$$- \Gamma'_c \stackrel{\Delta}{\simeq} \Gamma_c \langle \{id_c \mapsto v_c\} \rangle$$

$$- \Gamma'_s \stackrel{\Delta}{\simeq} \Gamma_s \langle \{id_s \mapsto close(\underline{v}_s, \xi)\} \rangle$$

$$- \Gamma'_a \stackrel{\Delta}{\simeq} \Gamma_a \langle \{id_a \mapsto v_a\} \rangle$$

$$- \llbracket id_m(id_c; id_s : \xi; id_a; \text{out } id'_s : e'_s) = P \rrbracket_d \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b \stackrel{\Delta}{\simeq} \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m \langle id_m \mapsto g \rangle, \emptyset$$

where

$$- g(v_c, \underline{v}_s, v_a, \underline{id''_s}, b', \underline{v}_c') \stackrel{\Delta}{\simeq} \{(O_i, \Gamma_{soi})\}$$

$$- \Gamma'_c \stackrel{\Delta}{\simeq} \Gamma_c \langle \{id_c \mapsto v_c\} \rangle$$

$$- \Gamma'_s \stackrel{\Delta}{\simeq} \Gamma_s \langle \{id_s \mapsto seal(close(\underline{v}_s, \xi), b)\} \rangle$$

$$- \Gamma''_s \stackrel{\Delta}{\simeq} \Gamma_s \langle \{id_s \mapsto \underline{v}_s\} \rangle$$

- $\Gamma'_a \stackrel{\Delta}{\simeq} \Gamma_a \langle \{id_a \mapsto seal(v_a, b)\} \rangle$
- $\{(O_i, \Gamma'_{soi})\} \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_p \Gamma'_c, \Gamma'_s, \Gamma'_a, \Gamma_m, b', \underline{v_c}'$
- $\Gamma_{soi} \stackrel{\Delta}{\simeq} \{id''_s \mapsto adapt(\underline{v_s}', v''_s)\}$ where
 - * $\underline{v_s}' \stackrel{\Delta}{\simeq} \Gamma'_{soi}(id'_s)$
 - * $v''_s, \emptyset = \llbracket e'_s \rrbracket_s \Gamma'_c, \Gamma''_s$

Explanation of the denotation function The cases for species and compartment definitions are straightforward since they rely on the respective denotation functions. The case of rate function definitions updates the rate function environment with a new binding to a function f from actual parameters and parent compartments to an algebraic rate value. This algebraic rate value is computed in the environments at time of definition updated with bindings for the actual parameters, and with the parent compartments at time of invocation. The interfaces of the actual species parameters are updated based on the annotations of the corresponding formal parameters using the *close* function defined in Section 5.2 which here is assumed extended to lists of species values. The function f is only defined when the number of actual and formal parameters match, and when the species interface update function is defined.

The case of module definitions updates the module environment with a new binding to a function from actual parameters, a fresh name string and parent compartments, to a set of semantical objects and species output environments. The semantical objects are computed in the environments at time of definition updated with bindings for actual parameters, and with the fresh name string and parent compartments at time of invocation. As for algebraic rate expressions, the interfaces of actual species values are updated. But an additional step is taken to confine species values to a namespace given by the fresh name string at time of definition, ensuring that e.g. variables in actual parameters are not captured inside the module. This is done using the *seal* function on modification site expressions, which is given as a parameter of the general semantics; we assume this to be extended appropriately to lists of species values and also to algebraic rate values. Finally, the resulting output species environment is given by a mapping from actual output species identifiers to the values of the corresponding formal output species identifiers as recorded in the output species environment of the body, but with interfaces updated using the *adapt* function defined in Section 5.2. The function is assumed extended to pairs of species value lists of the same length. Hence

the updates are carried out in a pair-wise manner by matching up corresponding positions in the lists constituting the nondeterministic species values; this is the reason for having nondeterministic species represented by lists rather than sets.

Chapter 6

Some Concrete Semantics of LBS

Practical applications of LBS require specific choices of concrete semantics to be made, and any questions of language expressiveness must also be addressed in the context of a specific concrete semantics. This chapter therefore gives five examples of concrete semantics, namely: basic Petri nets; coloured Petri nets; ordinary differential equations; continuous time Markov chains; and κ . The first four of these follow the ideas in [74], but are adapted to adhere to the general semantics of LBS.

The concrete semantics have not been implemented in the tool. As mentioned in the introduction, a translation to SBML has instead been implemented because this suffices as a proof of concept for our case studies. The implementation does however follow the general semantics presented in the previous chapter (with the exception that fresh names are generated imperatively for the sake of simplicity), and in fact the translation to SBML can be considered a concrete semantics in its own right. The concrete semantics defined in this chapter can therefore readily be incorporated into the tool as needed. We note that although SBML suffices for many practical applications and can indeed be translated to for example Petri nets, the direct definition of a Petri net concrete semantics is necessary when seeking to exploit modularity in analysis. We return to this topic in the next chapter in the context of Petri net flows.

6.1 Preliminaries

The general semantics preserves variables in species modification sites because variables can be exploited by some concrete semantics. But for other concrete semantics this is not the case, and we can instead parameterise the general semantic function on a structure $(S, |_S, \mathbf{0}_S, G_S, I_S)$ which is the same as before, except that G_S is a function

Table 6.1.1: The abstract syntax for ground rate values.

$v_{\text{gr}} ::=$	GROUND RATE VALUE	
$\{r\}$	RATE CONSTANT	
$[v_{\text{ga}}]$	GROUND ALGEBRAIC RATE VALUE	
$v_{\text{ga}} ::=$	GROUND ALGEBRAIC RATE VALUE	
r	CONSTANT	
v_{gns}	POPULATION	
$\exp(v_{\text{ga}}) \mid \log(v_{\text{ga}}) \mid \sin(v_{\text{ga}}) \mid \cos(v_{\text{ga}})$	FUNCTIONS	
$v_{\text{ga}} + v'_{\text{ga}} \mid v_{\text{ga}} - v'_{\text{ga}}$	ARITHMETIC OPERATORS	
$v_{\text{ga}} \times v'_{\text{ga}} \mid v_{\text{ga}} / v'_{\text{ga}} \mid v_{\text{ga}} \hat{\wedge} v'_{\text{ga}}$		

assigning semantical objects to named *ground normal form reactions*. These are normal form reactions in which expressions have been appropriately evaluated based on a variable environment: species values have been evaluated to ground normal form species values as defined previously; rate values have been evaluated to obtain *ground rate values* defined below; and reaction conditionals are omitted because reactions with conditionals which evaluate to **ff** are simply discarded.

In this section we define the general assignment R_S of semantical objects to named normal form reactions in terms of an assignment G_S to named ground normal form reactions, allowing a concrete semantics to be defined in terms of either of these.

6.1.1 Ground Normal Form Reactions

Ground algebraic rate values differ from algebraic rate values in that species values are replaced by ground normal form species values and conditionals are not included. Ground rate values contain ground algebraic rate values rather than algebraic rate values, and for the rate constant case, these must indeed be constants. The formal definition is given by the grammar in Table 6.1.1.

A denotation function of the form $\llbracket v_a \rrbracket_a \Gamma_x = v_{\text{ga}}$ assigning ground algebraic rate values to algebraic rate values, given a variable environment, is defined below. Only

selected cases for functions and arithmetic operators are shown since the remaining cases are similar.

- $\llbracket r \rrbracket_a \Gamma_x \stackrel{\Delta}{\simeq} r$
- $\llbracket v_{ns} \rrbracket_a \Gamma_x \stackrel{\Delta}{\simeq} \llbracket v_{ns} \rrbracket_m \Gamma_x$
- $\llbracket \text{if } e_b \text{ then } v_a \text{ else } v'_a \rrbracket_a \Gamma_x \stackrel{\Delta}{\simeq} \begin{cases} \llbracket v_a \rrbracket_a \Gamma_x & \text{if } \llbracket e_b \rrbracket_b \Gamma_x = \text{tt} \\ \llbracket v'_a \rrbracket_a \Gamma_x & \text{otherwise} \end{cases}$
- $\llbracket \exp(v_a) \rrbracket_a \Gamma_x \stackrel{\Delta}{\simeq} \exp(\llbracket v_a \rrbracket_a \Gamma_x)$
- $\llbracket v_a + v'_a \rrbracket_a \Gamma_x \stackrel{\Delta}{\simeq} \llbracket v_a \rrbracket_a \Gamma_x + \llbracket v'_a \rrbracket_a \Gamma_x$

In the case of normal form species values we assume the denotation function for modification site expressions extended in an evident manner.

Ground normal form reactions are then of the form:

$$G ::= \underline{n \cdot v_{gns}} \xrightarrow{v_{gr}} \underline{n' \cdot v'_{gns}}$$

6.1.2 The General Semantics in Terms of Ground Normal Form Reactions

The idea in the following construction is to obtain a ground normal form reaction for each possible variable environment associated with a normal form reaction, then get the semantical object of each ground normal form reaction, and finally apply the parallel composition operator to these objects. We therefore start by defining a function of the form $R_S(R, b, \Gamma_x) = O$ assigning a semantical object O to a normal form reaction R , named b , given a variable environment Γ_x :

$$R_S(\underline{n \cdot v_{ns}} \xrightarrow{v_r} \underline{n' \cdot v'_{ns}} \text{if } e_b, b, \Gamma_x) \stackrel{\Delta}{\simeq} \begin{cases} G_S(\underline{n \cdot \llbracket v_{ns} \rrbracket_m \Gamma_x} \xrightarrow{\llbracket v_r \rrbracket_a \Gamma_x} \underline{n' \cdot \llbracket v'_{ns} \rrbracket_m \Gamma_x}, b) & \text{if } \llbracket e_b \rrbracket_b \Gamma_x = \text{tt} \\ \mathbf{0}_S & \text{otherwise} \end{cases}$$

If the conditional evaluates to **ff**, the reaction is assigned the nil object, and otherwise the assignment relies on the function G_S for assigning a semantical object to the ground normal form of the reaction. Again we assume the denotation function on modification site expressions to be extended to normal form species values in an evident manner.

We also assume the denotation function for ground algebraic rate values to be extended to ground rate values in an evident manner; note that this function is only defined when ground algebraic rate values which are used as constants do indeed evaluate to constants.

The set of all variable environments associated with a normal form reaction is defined as follows, using the standard notation for dependent sets:

$$VE(R) \stackrel{\Delta}{=} \prod_{(x:\rho) \in FV(R)} \llbracket \rho \rrbracket_t$$

We here assume the variable function FV on modification site expressions to be extended to reactions in an evident manner. Observe that variable environments are restricted to only assign values of the given types to variables, and that for finite types, we get a finite set of variable environments.

In order to construct appropriate binary strings for naming reactions, we assume an arbitrary but fixed total ordering \leq on variable environments Γ_x . In practise this can for example be obtained from a lexicographical ordering on variables together with a suitable ordering on values. We assume an operator \circledcirc_s which gives the parallel composition in some definite order of its operands. Recall also that the function δ_i^m gives a binary string of length m with 0s everywhere except for the i th entry. The assignment R_S can then be defined in terms of G_S as follows:

$$\begin{aligned} R_S(R, b) &\stackrel{\Delta}{=} \\ &\circledcirc_s \{R_S(R, \delta_i^m b, \Gamma_x) \mid \Gamma_x \in VE(R) \wedge i = |\{\Gamma'_x \in VE(R) \mid \Gamma'_x \leq \Gamma_x\}| \wedge m = |VE(R)|\} \end{aligned}$$

6.2 A Basic Petri Net Semantics

6.2.1 Basic Petri Nets

We have already encountered a graphical representation of a basic Petri net in Figure 2.1.1 on page 12, and we have demonstrated how this can be obtained as a composition of two component Petri nets. To recap, *places*, depicted as circles, represent species, and *transitions*, depicted as rectangles, represent reactions. *Flow functions*, depicted as weighted arcs between places and transitions, represent stoichiometry. Finally, a *marking* defines the state of a Petri net by the number of *tokens* contained in each place, representing the number of individual molecules or concentration levels of the corresponding species.

The formal definition of our Petri nets is given below, where V_{gns} is the set of all ground normal form species values v_{gns} .

Definition 1. An LBS-Petri net PN is a tuple $(S, T, F^{\text{in}}, F^{\text{out}}, M^0)$ where

- $S \subsetneq_{\text{fin}} \{MS(v_{\text{gns}}) \mid v_{\text{gns}} \in V_{\text{gns}}\}$ is the set of places.
- $T \subsetneq_{\text{fin}} \{0, 1\}^*$ is the set of transitions.
- $F^{\text{in}}, F^{\text{out}} : T \times S \rightarrow \mathbb{N}$ are the flow-in and flow-out functions, respectively.
- $M^0 \in MS(S)$ is the initial marking.

Recall in the above definition that $MS(\underline{x})$ gives the multiset representation of a list \underline{x} . The set of places hence contains multiset-representations of ground normal form species values, reflecting that the ordering of atomic species within ground normal form species values is insignificant, i.e. that the complex formation operator is commutative. Transitions are binary strings since these are used to name reactions in the general semantics. We use the notation S_{PN} to refer to the places S of a Petri net PN , and similarly for the other Petri net elements. The set of all Petri nets is denoted by \mathcal{PN} .

6.2.2 The Qualitative Semantics of Basic Petri Nets

The qualitative semantics determines how the marking of a Petri net changes over discrete time, and we outlined this “token game” informally in Section 2.1. Formally, the set of all *markings* of a Petri net is the set of multisets of places:

$$\mathcal{M}(PN) \stackrel{\Delta}{\simeq} MS(S_{PN})$$

The behaviour of a Petri net is defined in terms of a transition relation which captures all possible moves in the token game.

Definition 2. Let PN be a Petri net, let $X \in MS(T_{PN})$ and let $M, N \in \mathcal{M}(PN)$. Then define $M \xrightarrow{X} N$ iff

1. $M \geq \sum_{t \in X} F_{PN}^{\text{in}}(t)$
2. $N = M + \sum_{t \in X} F_{PN}^{\text{out}}(t) - F_{PN}^{\text{in}}(t)$

Note that a flow function applied to only one argument, a transition, is interpreted as a function on places, here a marking. The arithmetic operations and relations are understood to be extended to markings in the expected way, e.g. $M \geq N$ iff $M(s) \geq N(s)$ for all s . Condition 1 hence states that the marking M must have sufficient tokens for transitions in X to fire, and condition 2 states that N is the marking resulting from firing the transitions from X in the marking M .

6.2.3 The Concrete Basic Petri Net Semantics of LBS

Definition 3. *The concrete semantics for LBS in terms of Petri nets is given by the tuple $(\mathcal{PN}, |_{\mathcal{PN}}, \mathbf{0}_{\mathcal{PN}}, G_{\mathcal{PN}}, I_{\mathcal{PN}})$ where*

- $PN_1 |_{\mathcal{PN}} PN_2 \stackrel{\Delta}{\simeq} PN$ where
 - $S_{PN} \stackrel{\Delta}{\simeq} S_{PN_1} \cup S_{PN_2}$
 - $T_{PN} \stackrel{\Delta}{\simeq} T_{PN_1} \cup T_{PN_2}$
 - $F_{PN}^{\text{io}}(t, s) \stackrel{\Delta}{\simeq} \begin{cases} F_{PN_1}^{\text{io}}(t, s) & \text{if } t \in T_{PN_1} \wedge s \in S_{PN_1} \\ F_{PN_2}^{\text{io}}(t, s) & \text{if } t \in T_{PN_2} \wedge s \in S_{PN_2} \\ 0 & \text{otherwise} \end{cases} \quad \text{for } \text{io} \in \{\text{in}, \text{out}\}$
 - $M_{PN}^0 \stackrel{\Delta}{\simeq} M_{PN_1}^0 + M_{PN_2}^0$

if $T_{PN_1} \cap T_{PN_2} = \emptyset$
- $\mathbf{0}_{\mathcal{PN}} \stackrel{\Delta}{\simeq} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
- $G_{\mathcal{PN}}(\underline{n} \cdot v_{\text{gns}} \xrightarrow{v_{\text{gr}}} \underline{n}' \cdot v'_{\text{gns}}, t) \stackrel{\Delta}{\simeq} PN$ where
 - $S_{PN} \stackrel{\Delta}{\simeq} \{MS(v_{\text{gns}} \cdot i)\} \cup \{MS(v'_{\text{gns}} \cdot j)\}$
 - $T_{PN} \stackrel{\Delta}{\simeq} \{t\}$
 - $F_{PN}^{\text{in}}(t, s) \stackrel{\Delta}{\simeq} \sum_{MS(v_{\text{gns}} \cdot i) = s} \underline{n} \cdot i$
 - $F_{PN}^{\text{out}}(t, s) \stackrel{\Delta}{\simeq} \sum_{MS(v'_{\text{gns}} \cdot j) = s} \underline{n}' \cdot j$
 - $M_{PN}^0 \stackrel{\Delta}{\simeq} \emptyset$
- $I_{\mathcal{PN}}(v_{\text{gns}}, n) \stackrel{\Delta}{\simeq} (\{MS(v_{\text{gns}})\}, \emptyset, \emptyset, \emptyset, \{MS(v_{\text{gns}}) \mapsto n\})$

The function $I_{\mathcal{PN}}$ is only defined for natural-numbered initial populations, not for real-numbered initial concentrations, because markings in Petri nets are discrete. The parallel composition operator is only defined for Petri nets with disjoint sets of transitions. The transition sets of two Petri nets resulting from the general semantics are however always disjoint because reactions have fresh names. This is in contrast to CBS where a bottom-up approach is taken: the semantics for parallel composition renames transitions before composition.

6.3 A Coloured Petri Net Semantics

Coloured Petri nets (CPNs) allow a single place to represent a species in any of its possible states of modification as we demonstrated in Figure 2.1.2 on page 14. CPNs hence allow for a compact description of models and can potentially lead to more efficient simulation and analysis, and in contrast to basic Petri nets, they are capable of representing species with infinite modification site types such as strings.

6.3.1 Coloured Petri Nets

Recall from Section 2.1 that places in CPNs are assigned *types* (or *colour*), and tokens are structured values of the type assigned to the place in which they reside. In our case, the type of a place is given by a multiset of located atomic species names and their modification site types, hence representing a complex species independently of its state of modification. Tokens are multiset representations of ground normal form species values, and arcs are equipped with multiset representations of normal form species values which are not necessarily ground. This enables a transition to operate selectively on species in a given state of modification, or indeed to ignore the state of certain sites. *Boolean guards* with variables allow transitions to assert further control over tokens.

We give a definition of coloured Petri nets which is tailored to our needs and which avoids some details of the standard definition [48]. For example, the standard definition distinguishes between place names and place types, but for our purposes a place is identified uniquely by its type. Our definition can however be recast in standard terms, as would be necessary for exploiting existing CPN tools.

Formally, we define a *species type* τ as follows:

$$\tau ::= \sum_i \underline{n}_{c_i}[n_{si}, \sigma_i]$$

and we let *Types* be the set of all species types. We define a function of the form $\text{type}(v_{\text{ns}}) = \tau$ giving the type of a normal form species value:

$$\text{type}(\underline{n}_c[n_s, \alpha_\sigma]) \stackrel{\Delta}{=} \sum_i \underline{n}_{c_i}[n_{si}, \sigma_i]$$

where σ_i is $\underline{\alpha_\sigma}.i$ in which each pair of the image has been projected to the type component. We assume a similar definition for a function of the form $\text{type}(v_{\text{gns}}) = \tau$ for ground normal form species values.

The formal definition of our coloured Petri nets is given below, where E_{bool} is set of boolean expressions e_b .

Definition 4. An LBS-coloured Petri net CPN is a tuple $(S, T, F^{\text{in}}, F^{\text{out}}, B, M^0)$ where

- $S \subsetneq_{\text{fin}} \text{Types}$ is a finite set of places.
- $T \subsetneq_{\text{fin}} \{0, 1\}^*$ is a finite set of transitions.
- $F^{\text{in}}, F^{\text{out}} : \prod_{(t, \tau) \in T \times S} MS(\{MS(v_{\text{ns}}) \mid \text{type}(v_{\text{ns}}) = \tau\})$ are the flow-in and flow-out functions, respectively.
- $B : T \rightarrow E_{\text{bool}}$ is the transition guard function.
- $M^0 : \prod_{\tau \in S} MS(\{MS(v_{\text{gns}}) \mid \text{type}(v_{\text{gns}}) = \tau\})$ is the initial marking.

As for basic Petri nets, we use the notation S_{CPN} to refer to the places S of a coloured Petri net CPN , and similarly for the other elements. The set of all coloured Petri nets is denoted by \mathcal{CPN} .

6.3.2 The Qualitative Semantics of Coloured Petri Nets

The set of all *markings* of a coloured Petri net CPN is defined as follows:

$$\mathcal{M}(CPN) \stackrel{\Delta}{=} \prod_{\tau \in S_{CPN}} MS(\{MS(v_{\text{gns}}) \mid \text{type}(v_{\text{gns}}) = \tau\})$$

We furthermore let

$$VE_X \stackrel{\Delta}{=} \{\Gamma_x \mid \text{dom}(\Gamma_x) = X\}$$

be the set of variable environments with domain X , and we let

$$FV(t, CPN) \stackrel{\Delta}{\simeq} FV(F_{CPN}^{\text{in}}(t)) \cup FV(F_{CPN}^{\text{out}}(t)) \cup FV(B_{CPN}(t))$$

be the set of typed variables associated with a transition t in CPN ; here FV is assumed extended in an evident manner. The behaviour of a coloured Petri net is defined in terms of a transition relation as follows.

Definition 5. Let CPN be a coloured Petri net, let $X \in MS(\prod_{t \in T_{CPN}} VE_{FV(t, CPN)})$ and let $M, N \in \mathcal{M}(CPN)$. Then define $M \xrightarrow{X} N$ iff

1. $M \geq \sum_{(t, \Gamma_x) \in X} [F_{CPN}^{\text{in}}(t)]_m \Gamma_x$
2. $N = M + \sum_{(t, \Gamma_x) \in X} [F_{CPN}^{\text{out}}(t)]_m \Gamma_x - [F_{CPN}^{\text{in}}(t)]_m \Gamma_x$
3. $\bigwedge_{(t, \Gamma_x) \in X} [B_{CPN}(t)]_m \Gamma_x = tt$

Recall that the modification site denotation function is a parameter of the species semantics, and in the above definition we assume this function to be extended from modification site expressions to normal form species values and to markings in an evident manner. A flow function applied to a transition is here interpreted as a marking, i.e. a mapping from places to multisets, and the multiset operations are assumed to be appropriately extended. Conditions 1 and 2 then correspond to conditions 1 and 2 in the qualitative semantics of basic Petri nets. Condition 3 states that the guards of all fired transitions must evaluate to tt .

6.3.3 The Concrete Coloured Petri Net Semantics of LBS

Definition 6. The concrete semantics for LBS in terms of coloured Petri nets is given by the tuple $(CPN, |_{CPN}, \mathbf{0}_{CPN}, R_{CPN}, I_{CPN})$ where

- $CPN_1 |_{CPN} CPN_2 \stackrel{\Delta}{\simeq} CPN$ where
 - $S_{CPN} \stackrel{\Delta}{\simeq} S_{CPN_1} \cup S_{CPN_2}$
 - $T_{CPN} \stackrel{\Delta}{\simeq} T_{CPN_1} \cup T_{CPN_2}$
 - $F_{CPN}^{\text{io}}(t, \tau) \stackrel{\Delta}{\simeq} \begin{cases} F_{CPN_1}^{\text{io}}(t, \tau) & \text{if } t \in T_{CPN_1} \wedge \tau \in S_{CPN_1} \\ F_{CPN_2}^{\text{io}}(t, \tau) & \text{if } t \in T_{CPN_2} \wedge \tau \in S_{CPN_2} \\ \emptyset & \text{otherwise} \end{cases} \quad \text{for } \text{io} \in \{\text{in, out}\}$

- $B_{CPN}(t) \stackrel{\Delta}{\simeq} \begin{cases} B_{CPN_1}(t) & \text{if } t \in T_{CPN_1} \\ B_{CPN_2}(t) & \text{if } t \in T_{CPN_2} \end{cases}$
- $M_{CPN}^0 \stackrel{\Delta}{\simeq} M_{CPN_1}^0 + M_{CPN_2}^0$
- if* $T_{CPN_1} \cap T_{CPN_2} = \emptyset$
- $\mathbf{0}_{CPN} \stackrel{\Delta}{\simeq} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
- $R_{CPN}(\underline{n} \cdot v_{ns} \Rightarrow^{vr} \underline{n}' \cdot v'_{ns} \text{ if } e_b, t) \stackrel{\Delta}{\simeq} CPN \text{ where}$
 - $S_{CPN} \stackrel{\Delta}{\simeq} \{type(v_{ns}.i)\} \cup \{type(v'_{ns}.j)\}$
 - $T_{CPN} \stackrel{\Delta}{\simeq} \{t\}$
 - $F_{CPN}^{\text{in}}(t, \tau) \stackrel{\Delta}{\simeq} \sum_{type(v_{ns}.i)=\tau} \underline{n}.i \cdot MS(v_{ns}.i)$
 - $F_{CPN}^{\text{out}}(t, \tau) \stackrel{\Delta}{\simeq} \sum_{type(v'_{ns}.j)=\tau} \underline{n}'.j \cdot MS(v'_{ns}.j)$
 - $B_{CPN}(t) \stackrel{\Delta}{\simeq} e_b$
 - $M_{CPN}^0 \stackrel{\Delta}{\simeq} \emptyset$
- $I_{CPN}(v_{gns}, n) \stackrel{\Delta}{\simeq} (\{type(v_{gns})\}, \emptyset, \emptyset, \emptyset, \emptyset, \{MS(v_{gns}) \mapsto n\})$

The definition is similar to that for basic Petri nets, but differs in the inclusion of guards and in the definition of reactions where species types and normal form values are used rather than ground values.

6.4 An ODE Semantics

The Petri net and coloured Petri net semantics presented above are *qualitative* in that they do not take reaction rates into account. In this section we give a *quantitative* semantics in terms of *ordinary differential equations* (ODEs). ODEs are *continuous* since they define system dynamics in terms of species concentrations. They are also *deterministic* since they, given initial conditions, uniquely determine the state of a system at any point of time in terms of species concentrations.

6.4.1 ODEs

A set of ODEs specifies how the concentration $[s_i]$ of a species s_i changes over time and is traditionally written in the following notation:

$$\begin{aligned}\frac{d[s_1]}{dt} &= p_1 \\ &\vdots \\ \frac{d[s_n]}{dt} &= p_n\end{aligned}$$

where the p_i are real polynomials over $[s_i]$. The initial conditions of a set of ODEs are specified by the concentration of each species at time 0.

Formally, let $(Pol(X), +, \cdot)$ be the ring of real polynomials over variables in the set X . We then define the structure of ODEs with initial conditions as follows:

Definition 7. A structure D of LBS-ODEs with initial conditions is given by a tuple (X, P, I) where

- $X \subsetneq_{\text{fin}} \{MS(v_{\text{gns}}) \mid v_{\text{gns}} \in V_{\text{gns}}\}$ is the set of variables.
- $P : X \rightarrow Pol(X)$ is the assignment of polynomials to variables.
- $I : X \rightarrow \mathbb{R}$ is the initial condition.

The set of all structures of ODEs with initial conditions is denoted by \mathcal{D} , and we denote e.g. X in D by X_D . Although non-linear ODEs cannot generally be solved in closed form, numerical integration methods are available and described in standard text books [7].

6.4.2 The Concrete ODE Semantics of LBS

Given two total functions $f_1 : X_1 \rightarrow Y$ and $f_2 : X_2 \rightarrow Y$ with a binary operator $+$ on the elements of Y , we define $f_1 + f_2 : X_1 \cup X_2 \rightarrow Y$ as follows:

$$(f_1 + f_2)(x) \stackrel{\Delta}{=} \begin{cases} f_1(x) & \text{if } x \in X_1 \setminus X_2 \\ f_2(x) & \text{if } x \in X_2 \setminus X_1 \\ f_1(x) + f_2(x) & \text{if } x \in X_1 \cap X_2 \end{cases}$$

The semantics of LBS in terms of ODEs is defined below.

Definition 8. The concrete semantics for LBS in terms of ODEs is given by the tuple $(\mathcal{D}, |\mathcal{D}, \mathbf{0}_{\mathcal{D}}, G_{\mathcal{D}}, I_{\mathcal{D}})$ where

- $D_1 |_{\mathcal{D}} D_2 \stackrel{\Delta}{\simeq} D$ where
 - $X_D \stackrel{\Delta}{\simeq} X_{D_1} \cup X_{D_2}$
 - $P_D \stackrel{\Delta}{\simeq} P_{D_1} + P_{D_2}$
 - $I_D \stackrel{\Delta}{\simeq} I_{D_1} + I_{D_2}$
- $\mathbf{0}_{\mathcal{D}} \stackrel{\Delta}{\simeq} (\emptyset, \emptyset, \emptyset)$
- $G_{\mathcal{D}}(\underline{n} \cdot v_{\text{gns}} \Rightarrow^{v_{\text{gr}}} \underline{n}' \cdot v'_{\text{gns}}, b) \stackrel{\Delta}{\simeq} D$ where
 - $X_D \stackrel{\Delta}{\simeq} \{MS(v_{\text{gns}} \cdot i)\} \cup \{MS(v'_{\text{gns}} \cdot j)\}$
 - $P_D(s) \stackrel{\Delta}{\simeq} \begin{cases} (N(s) - M(s)) \cdot r \cdot \prod_i (MS(v_{\text{gns}} \cdot i))^{n.i} & \text{if } v_{\text{gr}} = \{r\} \\ v_{\text{ga}} & \text{if } v_{\text{gr}} = [v_{\text{ga}}] \end{cases}$
where
 - * $M(s) \stackrel{\Delta}{\simeq} \sum_{MS(v_{\text{gns}} \cdot i)=s} n.i$
 - * $N(s) \stackrel{\Delta}{\simeq} \sum_{MS(v'_{\text{gns}} \cdot j)=s} n'.j$
 - $I_D(s) = 0$
- $I_{\mathcal{D}}(v_{\text{gns}}, r) \stackrel{\Delta}{\simeq} (\{s\}, \{s \mapsto 0\}, \{s \mapsto r\})$ where
 - $s \stackrel{\Delta}{\simeq} MS(v_{\text{gns}})$

In the case of reactions, rate expressions are constructed from mass-action rate constants in the standard way [89].

6.5 A CTMC Semantics

We now give another quantitative semantics in terms of *continuous time Markov chains* (CTMCs). In contrast to ODEs, CTMCs are *discrete* since they describe the system state in terms of species populations rather than concentrations, and they give rise to *stochastic* behaviour.

6.5.1 CTMCs

The state of a CTMC corresponds to a marking of a Petri net and is hence given by a multiset of ground normal form species values in their multiset form. State transitions are described directly in terms of a transition rate matrix. Here is the formal definition:

Definition 9. An LBS-continuous time Markov chain V with initial state is a tuple (X, Q, I) where

1. $X \subset MS(\{MS(v_{gns}) \mid v_{gns} \in V_{gns}\})$ is the set of states.
2. $Q : X^2 \rightarrow \mathbb{R}$ is the transition rate matrix satisfying
 - (a) $Q(M, N) \geq 0$ for all $M, N \in X$ with $M \neq N$
 - (b) $\sum_{M \neq N} Q(M, N) = -\sum_{M \neq N} Q(N, M)$
3. $I \in X$ is the initial state.

The set of all CTMCs with initial state is denoted by \mathcal{V} , and we denote e.g. X in V by X_V . We refer to the literature [89] for further details on CTMCs and their associated simulation methods.

6.5.2 The Concrete CTMC Semantics of LBS

Definition 10. The concrete semantics for LBS in terms of CTMCs is given by the tuple $(\mathcal{V}, |_\mathcal{V}, \mathbf{0}_\mathcal{V}, G_\mathcal{V}, I_\mathcal{V})$ where

- $V_1 |_\mathcal{V} V_2 \stackrel{\Delta}{\simeq} V$ where
 - $X_V \stackrel{\Delta}{\simeq} \{M + N \mid M \in X_{V_1} \wedge N \in X_{V_2}\}$
 - $Q_V(M, M') \stackrel{\Delta}{\simeq} \begin{cases} Q_{V_1}(M, M') & \text{if } (M, M') \in dom(Q_{V_1}) \setminus dom(Q_{V_2}) \\ Q_{V_2}(M, M') & \text{if } (M, M') \in dom(Q_{V_2}) \setminus dom(Q_{V_1}) \\ Q_{V_1}(M, M') + Q_{V_2}(M, M') & \text{if } (M, M') \in dom(Q_{V_1}) \cap dom(Q_{V_2}) \\ 0 & \text{otherwise} \end{cases}$
 - $I_V \stackrel{\Delta}{\simeq} I_{V_1} + I_{V_2}$
- $\mathbf{0}_\mathcal{V} \stackrel{\Delta}{\simeq} (\emptyset, \emptyset, \emptyset)$

- $G_{\mathcal{V}}(\underline{n} \cdot v_{\text{gns}} \Rightarrow^{\{r\}} \underline{n'} \cdot v'_{\text{gns}}, t) \stackrel{\Delta}{\simeq} V \text{ where}$
 - $X_V \stackrel{\Delta}{\simeq} MS(\{MS(v_{\text{gns}}.i)\} \cup \{MS(v'_{\text{gns}}.j)\})$
 - $Q_V(M', N') \stackrel{\Delta}{\simeq} \begin{cases} r \binom{M'}{M} & \text{if } (M, N) \preceq (M', N') \wedge M \neq N \\ -r \binom{M'}{M} & \text{if } M' = N' \wedge M \neq N \wedge M' \geq M \\ 0 & \text{otherwise} \end{cases}$

where

- * $M(s) \stackrel{\Delta}{\simeq} \sum_{MS(v_{\text{gns}}.i)=s} \underline{n}.i$
- * $N(s) \stackrel{\Delta}{\simeq} \sum_{MS(v'_{\text{gns}}.j)=s} \underline{n'}.j$
- * $\binom{M'}{M} \stackrel{\Delta}{\simeq} \prod_{s \in \text{dom}(M')} \binom{M'(s)}{M(s)}$
- * $(M, N) \preceq (M', N') \text{ iff } \exists L. M' = M + L \wedge N' = N + L$
- $I_V(s) = 0$
- $I_{\mathcal{V}}(v_{\text{gns}}, n) \stackrel{\Delta}{\simeq} V \text{ where}$
 - $X_V \stackrel{\Delta}{\simeq} MS(\{MS(v_{\text{gns}})\})$
 - $Q_V(M, N) \stackrel{\Delta}{\simeq} 0$
 - $I_V(\{MS(v_{\text{gns}})\}) = n$

In the definition of the transition rate matrix in the case of parallel composition, we assume two multisets with different domains to be equal whenever they are equal on their common domain and 0 elsewhere. In the case of reactions, $\binom{x}{y}$ is the binomial coefficient and state transition rates are constructed from rate constants in the standard way [89]. Note that the reaction assignment is only defined when constant rate expressions are used.

6.6 A κ Semantics

In Section 3.2.3 we discussed how an appropriate modification site type, **binding**, can be used to model complexes at the level of bindings rather than at the level of multisets. In this section we formally define expressions of type **binding** and the associated functions required by the general semantics. We then give a concrete semantics of LBS in terms of κ , and a similar semantics can be given in terms of BioNetGen. We start by defining an abstract syntax of κ based on the supplementary material of [37], and also refer the

Table 6.6.1: The abstract syntax for κ rules.

$x ::= \underline{a} \rightarrow^r \underline{a}'$	κ RULE
$a ::= (n_s, \underline{s})$	AGENT
$s ::= (n_m, i, l)$	SITE
$i ::=$	INTERNAL STATE
v	INTERNAL STATE VALUE
?	WILD CARD
$l ::=$	LINK
?	FREE OR BOUND
-	BOUND TO SOMETHING
o	FREE
n	LINK LABEL

reader to [37] for a formal presentation of the semantics of κ which, like CTMCs, is discrete and stochastic.

6.6.1 κ

κ rules The abstract syntax for κ rules is given in Table 6.6.1 where, as before, n_s ranges over the set of atomic species names, n_m ranges over the set of modification site names, $r \in \mathbb{R}$ and $n \in \mathbb{N}$. Furthermore, v ranges over a given set of internal state values as in κ together with the set of compartment name lists; the latter is needed in order to encode LBS compartments in κ . In the context of κ , we refer to atomic species as *agents*.

A κ rule consists of a list of reactant agents and a list of product agents, and the arrow is labelled with a rate constant. An agent is a pair consisting of an agent name and a list of sites, and a site is given by a name, an internal state and a link. An internal state can be a value, such as “phosphorylated” or “unphosphorylated”, or it can be a wild card indicating “any” value. A link can be two kinds of wild cards, the

most permissive being “either free or bound” and the more restricted being “bound to something”; a link can also be free, i.e. unbound, or it can be bound by some specific label. The internal state may be omitted in which case the wild card, ?, is assumed, and the link may be omitted in which case the free link, \circ , is assumed.

We furthermore need the notion of a *ground agent*, a_g , which is an agent without wild cards in its internal states and links.

κ programs We now give the formal definition of our κ programs with initial conditions. These are our semantical objects.

Definition 11. An LBS- κ program K with initial conditions is a pair (X, I) where

1. $X ::= \{x_i\}$ is a set of κ rules.
2. $I ::= a_g$ is the initial condition.

We denote e.g. X in K by X_K , and the set of κ programs is denoted by \mathcal{K} .

Well-typedness An agent is well-typed if each of its site names occurs exactly once; a list of agents is well-typed if all the agents are well-typed and each link label occurs exactly twice; a rule is well-typed if its two lists of agents are well-typed; and finally, a κ program is well-typed if all its rules and its initial condition are well-typed.

Our presentation differs slightly from that in [37]. There an *interface* function is assumed which assigns sites to all agent names. An additional well-typedness condition then requires that an agent only uses sites which are mentioned in its interface. Although an LBS program does contain the information needed to construct interfaces in the translation to κ , doing so is not strictly necessary since, given a set of κ rules, a minimal interface can always be chosen such that the well-typedness condition is satisfied. We hence omit interfaces in our presentation for the sake of simplicity.

A note on semantics Although we omit a formal presentation of the semantics of κ , there is one important point to make, namely that the ordering of agents within rules matters. For example, the rule:

$A(m!1), B(m!1, n \sim u) \rightarrow A(m!1), B(m!1, n \sim p)$

is *not* semantically equivalent to the rule:

$A(m!1), B(m!1, n \sim u) \rightarrow B(m!1, n \sim p), A(m!1)$

When applying the first rule, the site n in a matching instance of B is phosphorylated while preserving any internal state and links not specified in the rule. But in the second rule, the matching instances of the agents A and B are first deleted together with any internal state and links they may have, and two new agents with internal state and links as specified by the RHS are then added.

The reason is that a rule may have multiple copies of the same agent on both its LHS and RHS, and the copies on the two sides must be matched up when applying the rule. The matching is done through a *longest common prefix policy*: instances of agents in the common prefix are simply updated as defined by the RHS, but instances of agents after the common prefix on the LHS are first deleted and then the agents after the common prefix on the RHS are created. The prefix policy takes both agent and site names into account, so also the rule:

$A(m!1), B(m!1, n \sim u) \rightarrow A(m!1), B(m!1, n \sim p)$

is semantically different from the rule:

$A(m!1), B(m!1, n \sim u) \rightarrow A(m!1), B(n \sim p, m!1)$

This may seem counter-intuitive and indeed unnecessary since site names are never duplicated within agents. In LBS, modification sites are not ordered. The translation to κ rules therefore assumes a fixed, global ordering on site names to avoid situations of the kind described above. On the other hand, reactants, products and complex species are represented by lists in LBS, which allows the ordering between agents to be preserved in the translation to κ .

6.6.2 The Concrete κ Semantics of LBS

Modification site expressions of type binding Recall that the abstract syntax for LBS is parameterised on a set of modification site types and expressions. In order for a κ semantics to be of interest, we here assume that the set of modification site types contains exactly the **binding** type. Correspondingly, the set of modification site expressions is assumed to consist of the set of *binding expressions* e_{bd} generated by the grammar in Table 6.6.2, where $b \in \{0, 1\}^*$ is a namespace used to confine link names in a similar manner to variables in the case of boolean expressions. Put informally, only LBS programs which are written in a κ style have meaningful translations to κ .

A binding expression is a pair consisting of an LBS internal state and an LBS link. These are defined as for κ , except that link labels have a namespace consisting of a list

Table 6.6.2: The abstract syntax for modification site expressions e_{bd} of type **binding**.

$e_{\text{bd}} ::= (i^+, l^+)$	LBS BINDING EXPRESSION
$i^+ ::=$	LBS INTERNAL STATE
v	INTERNAL STATE VALUE
$?$	WILD CARD
ϵ	IDENTITY
$l^+ ::=$	LBS LINK
$?$	FREE OR BOUND
$-$	BOUND TO SOMETHING
\circ	FREE
(n, \underline{b})	RESTRICTED LINK LABEL
ϵ	IDENTITY

of binary strings, and both internal states and links include the *identity*. Identities are used in species updates where one may need to update either the internal state or link, but not both, as demonstrated in Subsection 3.2.3.

Recall that the general semantics is parameterised on a number of functions on modification site expressions. For **binding** expressions, these are defined as follows:

- $e_{\text{bd}} : \text{binding}$ for all e_{bd}
- $\text{default}(\text{binding}) \stackrel{\Delta}{=} (? , ?)$
- $FV(e_{\text{bd}}) \stackrel{\Delta}{=} \emptyset$
- $\llbracket e_{\text{bd}} \rrbracket_m \Gamma_x \stackrel{\Delta}{=} e_{\text{bd}}$
- $(i^+, l^+) \langle (i^{+'}, l^{+'}) \rangle \stackrel{\Delta}{=} (i^+ \langle i^{+'} \rangle, l^+ \langle l^{+'} \rangle)$ where
 - $i^+ \langle i^{+'} \rangle \stackrel{\Delta}{=} \begin{cases} i^+ & \text{if } i^{+'} = \epsilon \\ i^{+'} & \text{otherwise} \end{cases}$
 - $l^+ \langle l^{+'} \rangle \stackrel{\Delta}{=} \begin{cases} l^+ & \text{if } l^{+'} = \epsilon \\ l^{+'} & \text{otherwise} \end{cases}$
- $\text{seal}((i^+, l^+), b) \stackrel{\Delta}{=} \begin{cases} (i^+, (n, bb')) & \text{if } l^+ = (n, b') \\ (i^+, l^+) & \text{otherwise} \end{cases}$

The typing relation is trivial since we assume just one modification site type. The free variable and valuation functions are also trivial since κ expressions do not contain variables. One could of course choose to have more complex internal states which include variables, thus obtaining a kind of “coloured” κ as in coloured Petri nets, but they are omitted here for the sake of simplicity. The default expression for unspecified modification site types has a wild card internal state and link, which reflects the use of unspecified sites in κ . The update function overwrites any internal state or links in all cases except when the identity is used for updating. Finally, the seal function simply updates the namespace of any link labels by appending the given binary string to the list of binary strings already present.

We have some simple derived forms for use in updates. Omitting the list of binary strings in a restricted link label is understood as the empty list. As outlined in Subsection 3.2.3, omitting the internal state or link is understood as respectively the identity

internal state and the free link:

$$(l^+) \stackrel{\Delta}{\simeq} (\epsilon, l^+) \text{ and } (i^+) \stackrel{\Delta}{\simeq} (i^+, \circ) \text{ and } () \stackrel{\Delta}{\simeq} (\epsilon, \circ)$$

The concrete semantics Modifications in LBS are represented by finite functions rather than lists as in κ . The translation to κ must therefore “linearise” these functions, for which we assume a linear ordering, \leq , on modification site names. It must also convert restricted link labels to natural-number link labels, for which we assume an injective function of the form $enc(n, b) = n'$; a definition could e.g. be based on a Gödel numbering. Finally, the translation must disregard modification site types.

Given an LBS modification $\beta_\sigma = \{n_m j \mapsto (\sigma, (i^+, l^+))_j\}$ we then define $kap_m(\beta_\sigma)$ to be the list with the element $(n_m j, i_j^+, enc(l_j^+))$ at index $|S|$ where

$$S \stackrel{\Delta}{\simeq} \{n_m \in dom(\beta_\sigma) \mid n_m \leq n_m j\}$$

We here assume enc extended to LBS links in an evident manner.

The translation of an atomic species to a κ agent simply translates the modification sites and adds an additional site with an internal state representing the enclosing compartments. For the latter we assume a distinguished compartment name, *comp*, and that the set of internal state values contains the set of compartment name lists. We then define a *kappa translation function* of the form $kap(v_{gns}) = \underline{a}$ for translating ground normal form species values to lists of κ agents as follows:

$$kap(\underline{n}_c[n_s, \beta_\sigma]) \stackrel{\Delta}{\simeq} (n_s, kap_m(\beta_\sigma)(comp, \underline{n}_c, \circ))$$

where $kap_m(\beta_\sigma)(comp, \underline{n}_c, \circ)$ following our notational conventions is the postfixing of the triple $(comp, \underline{n}_c, \circ)$ to the list $kap_m(\beta_\sigma)$.

LBS reactions are translated into κ rules by applying the above function to each ground normal form species value and flattening the lists representing reactants and products. We define a *flattening function* of the form $flatten(\underline{x}) = \underline{x}'$ as follows:

$$flatten(\underline{x}) \stackrel{\Delta}{\simeq} (\underline{x}.1) \dots (\underline{x}.|\underline{x}|)$$

and we also define a *duplication function* of the form $n \times \underline{x} = \underline{x}'$ for duplicating lists according to the stoichiometry given in reactions:

$$n \times \underline{x} \stackrel{\Delta}{\simeq} \underbrace{\underline{x} \dots \underline{x}}_{n \text{ times}}$$

The concrete κ semantics of LBS can now be defined.

Definition 12. The concrete semantics for LBS in terms of κ is given by the tuple $(\mathcal{K}, |\mathcal{K}, \mathbf{0}_{\mathcal{K}}, G_{\mathcal{K}}, I_{\mathcal{K}})$ where

- $K_1 |_{\mathcal{K}} K_2 \stackrel{\Delta}{\simeq} K$ where
 - $X_K \stackrel{\Delta}{\simeq} X_{K_1} \cup X_{K_2}$
 - $I_K \stackrel{\Delta}{\simeq_t} I_{K_1} I_{K_2}$
- $\mathbf{0}_{\mathcal{K}} \stackrel{\Delta}{\simeq} (\emptyset, \varepsilon)$
- $G_{\mathcal{K}}(\underline{n} \cdot v_{\text{gns}} \Rightarrow^{\{r\}} \underline{n'} \cdot v'_{\text{gns}}, b) \stackrel{\Delta}{\simeq} K$ where
 - $X_K \stackrel{\Delta}{\simeq_t} \{\underline{a} \rightarrow^r \underline{a}'\}$ where
 - $\underline{a} \stackrel{\Delta}{\simeq} \text{flatten}(\underline{n} \times \text{kap}(v_{\text{gns}}))$
 - $\underline{a}' \stackrel{\Delta}{\simeq} \text{flatten}(\underline{n'} \times \text{kap}(v'_{\text{gns}}))$
 - $I_K \stackrel{\Delta}{\simeq} \varepsilon$
- $I_{\mathcal{K}}(v_{\text{gns}}, n) \stackrel{\Delta}{\simeq} K$ where
 - $X_K \stackrel{\Delta}{\simeq} \emptyset$
 - $I_K \stackrel{\Delta}{\simeq_t} n \times \text{kap}(v_{\text{gns}})$

In the case of parallel composition, the resulting initial condition is only defined if the sets of link labels in the initial conditions of the two components are disjoint, for otherwise the initial condition would not be well-typed. The case of reactions is likewise only defined if the resulting κ rule is well-typed, and the case of initial conditions is only defined if the resulting agent list is well-typed.

Note that well-typedness of species expressions with binding sites is only determined by the semantics when the species expressions are used in reactions. A dedicated typing system would be needed to determine well-typedness earlier, e.g. at time of species definition.

Chapter 7

Concrete Petri Net Flow Semantics of LBS

We have demonstrated how modularity in LBS facilitates a structured approach to modelling, and we have given a formal presentation of the language including a concrete semantics in terms of Petri nets. We now turn to the question of how modularity can be exploited in analysis, specifically in the case of Petri net flows.

Intuitively, a *transition flow* (or T-flow) is a vector representing occurrences of reactions which together have no net effect on species populations. T-flows hence correspond to a notion of cyclic pathways. A *place flow* (or P-flow) is a vector representing species weights for which the weighted sum of species populations is always constant. P-flows hence correspond to chemical conservation relations.

More precisely, T and P-flows are natural-number solutions to the equations $Wx = 0$ and $xW = 0$, respectively, where W is the *flow matrix* of a Petri net. W can be derived from Petri net flow functions and also corresponds to the stoichiometry matrix of a biological reaction network. These equations generally have infinitely many solutions, but one can always find finite sets of *minimal* flows which can be combined to generate all other flows. Algorithms for obtaining minimal flows are described in e.g. [52] and are computationally expensive.

We start by introducing the relevant background in Section 1, including a matrix-based view of Petri nets, formal definitions of flows and existing results from the literature. In Section 2 we recast the definition of Petri net composition, introduced in the previous chapter, in terms of matrices, and also show a duality result relating T-flows and P-flows in a modular setting. We then show how the flows of a composite Petri net can be derived from its components, with T and P-flows treated in Section 3 and 4,

respectively. These results are used to define concrete Petri net flow semantics of LBS in Section 5. Previous efforts have been made towards modular definitions of P-flows in particular, and related work is discussed in Section 6. Detailed proofs of all results can be found in Appendix C.

7.1 Preliminaries

7.1.1 Flow Matrices

Recall from the previous chapter that a Petri net consists of a set of places, a set of transitions, flow in and flow out functions, and an initial marking. In this chapter we are not concerned with initial markings since flows are independent of these. We do however need an alternative, matrix-based representation of flow functions in order to take advantage of standard notation and results from linear algebra.

An *ordered Petri net*, OPN , is a triple (PN, \prec_S, \prec_T) consisting of a Petri net PN together with linear orderings $\prec_S \subseteq S_{PN} \times S_{PN}$ on places and $\prec_T \subseteq T_{PN} \times T_{PN}$ on transitions. Using these orderings we can write $S_{PN} = (s_1, \dots, s_m)$ and $T_{PN} = (t_1, \dots, t_n)$ and view the flow functions of PN as $m \times n$ *flow-in* and *flow-out matrices* thus:

$$(W_{OPN}^{\text{in}})_{i,j} \stackrel{\Delta}{=} F_{PN}^{\text{in}}(t_j, s_i) \quad (W_{OPN}^{\text{out}})_{i,j} \stackrel{\Delta}{=} F_{PN}^{\text{out}}(t_j, s_i)$$

The i th row of W_{OPN}^{in} represents the number of tokens *consumed* from the place s_i by each of the transitions, and the i th row of W_{OPN}^{out} represents the number of tokens *produced* in the place s_i by each of the transitions. We furthermore assume a *net flow matrix* (or *flow matrix*) $W_{OPN} \stackrel{\Delta}{=} W_{OPN}^{\text{out}} - W_{OPN}^{\text{in}}$ associated with any given ordered Petri net OPN . The i th row of W_{OPN} then represents the net effect of each of the transitions on the place s_i . The net flow matrix is also referred to as the *incidence matrix* in the Petri net literature.

We often assume an arbitrary but fixed choice of linear orderings and write PN instead of OPN . Furthermore, when the Petri net PN is given by the context or when it is not significant, we often omit subscripts and write e.g. W instead of W_{PN} , and similarly for other components.

For the sake of illustration, we give an alternative, matrix-based definition of Petri net behaviour that is equivalent to Definition 2 on page 103. A state, or marking, of a Petri net is now given by a vector with natural-number entries representing the

number of tokens in the corresponding places. Hence the set of all markings of a Petri net is defined as follows; here and throughout, we use $(\cdot)^T$ to denote vector/matrix transposition:

$$\mathcal{M}(PN) \stackrel{\Delta}{\simeq} (\mathbb{N}^{|S|})^T$$

The formal definition of behaviour is then given below.

Definition 13 (Behaviour). *Let $PN = (S, T, F^{\text{in}}, F^{\text{out}}, M^0)$ be a Petri net. Then the transition relation $\rightarrow \subseteq \mathcal{M}(PN) \times (\mathbb{N}^{|T|})^T \times \mathcal{M}(PN)$ is defined as follows: $M \xrightarrow{x} M'$ iff*

1. $M \geq W^{\text{in}}x$
2. $M' = M + W^{\text{out}}x - W^{\text{in}}x = M + Wx$

7.1.2 Petri Net Flows

The formal definition of T and P-flows follows below.

Definition 14 (T and P-flows). *Let $PN = (S, T, F^{\text{in}}, F^{\text{out}}, M^0)$ be a Petri net. Define*

$$\begin{aligned} TF(PN) &= TF(W) \stackrel{\Delta}{\simeq} \{x \in (\mathbb{N}^{|T|})^T \mid Wx = 0 \wedge x \neq 0\} \\ PF(PN) &= PF(W) \stackrel{\Delta}{\simeq} \{y \in \mathbb{N}^{|S|} \mid yW = 0 \wedge y \neq 0\} \end{aligned}$$

The elements of $TF(PN)$ and $PF(PN)$ are called transition flows (or T-flows) and place flow (or P-flows), respectively.

Observe that T and P-flows are dual in the following sense:

$$x \in TF(PN) \Leftrightarrow Wx = 0 \Leftrightarrow x^T W^T = 0 \Leftrightarrow x^T \in PF(PN^D)$$

where the Petri net duality operator $(\cdot)^D$ swaps around the places and transitions in a Petri net and reverses arcs [65].

A Petri net generally has infinitely many flows. But it is possible to obtain a finite set of *minimal flows* which can be combined to form all other flows. In the following we consider the structure of flows irrespectively of whether they are T or P flows. We hence use $F(PN)$ and $MF(PN)$ to denote the set of either type of flows and minimal flows of PN , respectively.

Definition 15 (Support). *The support of a vector $x \in \mathbb{N}^*$, denoted by $\text{sup}(x)$, is the set of indices of non-zero entries in x : $\text{sup}(x) \stackrel{\Delta}{\simeq} \{i \mid x_i \neq 0\}$.*

Definition 16 (Minimal flows). A flow $x \in F(PN)$ is minimal if

1. x is canonical, i.e. the greatest common divisor of non-zero entries of x , written $\text{gcd}(x)$, is 1 and
2. x has minimal support, i.e. there is no other flow $x' \in F(PN)$ with $\text{sup}(x') \subsetneq \text{sup}(x)$.

We denote by $MTF(PN)$ (or $MTF(W)$) and $MPF(PN)$ (or $MPF(W)$) the sets of minimal T and P-flows of PN , respectively.

There is a less common definition of minimality, *weak minimality*, which dispenses with the notion of support. We state it because the results in this paper are proved valid for both definitions of minimality, and the proofs are generally simpler for weak minimality.

Definition 17 (Weakly minimal flows). A flow $z \in F(PN)$ is weakly minimal if there are no other flows $x, y \in F(PN)$ s.t. $z = x + y$. We denote by $M_wTF(PN)$ and $M_wPF(PN)$ the set of weakly minimal T and P-flows of PN , respectively.

Given a set of flows we sometimes need to filter out the non-minimal ones as in the following definitions.

Definition 18 (Minimisation). Let X be a set of flows. Define minimisation thus:

$$\min(X) \stackrel{\Delta}{=} \left\{ \frac{x}{\text{gcd}(x)} \mid x \in X \wedge \forall x' \in X. \text{sup}(x') \not\subseteq \text{sup}(x) \right\}$$

Definition 19 (Weak minimisation). Let X be a set of flows. Define weak minimisation thus:

$$\min_w(X) \stackrel{\Delta}{=} \left\{ x \in X \mid \forall x_1, \dots, x_k \in X. \forall a_1, \dots, a_k \in \mathbb{N}. x \neq a_1x_1 + \dots + a_kx_k \right\}$$

7.1.3 A Running Example: Photosynthesis and Respiration

As a running example we consider a simple model of the foundations of life itself, namely photosynthesis and respiration. An LBS program with two modules representing these processes is shown in Listing 7.1.1.

Photosynthesis is the process by which plants produce sugar and oxygen from water, carbon dioxide and sunlight (photons). This is modelled by three reactions. The first reaction introduces photons into the system. The second reaction converts photons and water into chemical energy (CE) and oxygen, and the third reaction converts chemical energy and carbon dioxide into sugar.

Listing 7.1.1: An LBS model of photosynthesis and respiration.

```

1 spec Photons = new{}, CE = new{};
2 spec H2O = new{}, O2 = new{}, CO2 = new{}, Sugar = new{};
3 spec ChE = new{}, Heat = new{};

4
5 module photosynthesis() {
6   -> Photons |
7   Photons + H2O -> CE + O2 |
8   CE + CO2 -> Sugar
9 };

10
11 module respiration() {
12   Sugar -> ChE + CO2 |
13   ChE + O2 -> Heat + H2O |
14   Heat ->
15 };

16
17 photosynthesis() ||
18 respiration() ||
19 ( photosynthesis() | respiration() )

```

Respiration is the converse process by which e.g. humans use oxygen to break down sugar while producing carbon dioxide and water. This is also modelled by three reactions. The first reaction breaks down sugar into carbon dioxide and chemical energy ChE, distinct from the chemical energy used in photosynthesis. The second reaction utilises this chemical energy and oxygen to make e.g. muscles move, and in the process producing water and heat; the heat is finally removed from the system in the third reaction. We note that both models are strongly simplified and not chemically correct.

The last three lines of the LBS program consist of a variation composition of the module invocations in isolation and in parallel, hence giving rise to a set of three semantical objects. In the case of Petri nets, these are shown in Figures 7.1.1a, 7.1.1b and 7.1.1c, respectively.

We give the minimal flows of the photosynthesis and respiration Petri nets informally by listing only the places and transitions which have non-zero entries in the flows rather than writing out the full vectors. There are three minimal P-flows in the photosynthesis Petri net determined by the places (H₂O,O₂), (CO₂,Sugar) and

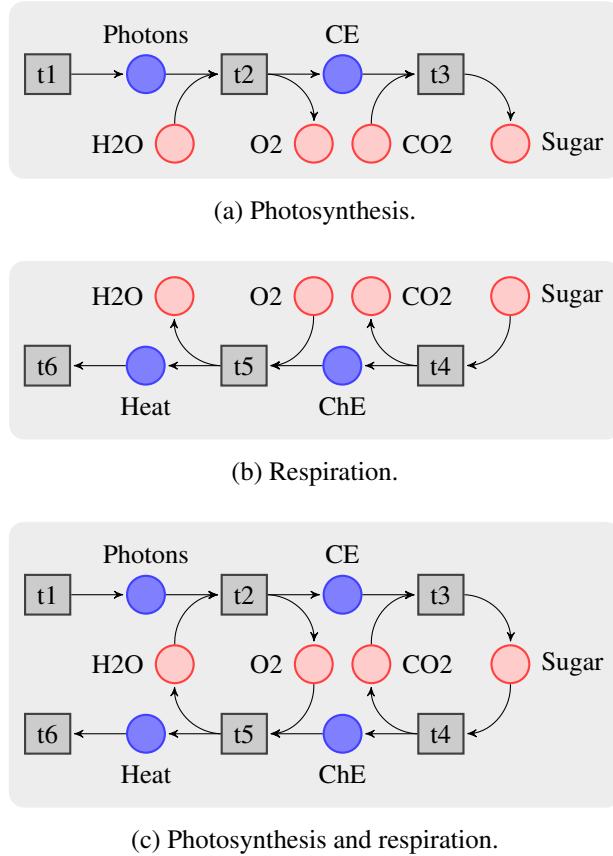


Figure 7.1.1: Petri net models of photosynthesis and respiration in isolation and in parallel; a distinct shading is used for shared places.

(CE, H₂O, Sugar). Symmetrically, there are three minimal P-flows in the respiration Petri net determined by the places (H₂O, O₂), (CO₂, Sugar) and (H₂O, Sugar, ChE). However, neither the photosynthesis nor the respiration Petri net has any T-flows.

In Sections 3 and 4 we investigate the flows of the composite Petri net and their relation to the above mentioned flows of the components.

7.1.4 Existing Results

The following two theorems are adapted from [52]. They state that $MTF(PN)$ and $MPF(PN)$ are well-defined, and that any flow can be generated from minimal flows by natural-number linear combinations followed by a division.

Theorem 1. $MTF(PN)$ and $MPF(PN)$ are finite and unique.

Theorem 2. For any flow $x \in F(PN)$ there are $a, a_1, \dots, a_k \in \mathbb{N}$ and minimal flows $x_1, \dots, x_k \in MF(PN)$ s.t. $x = \frac{1}{a}(a_1x_1 + \dots + a_kx_k)$.

We also need the following theorem, adapted from [62], which states that any two flows with the same minimal support are multiples of each other.

Theorem 3. *Let $x, y \in F(PN)$. If x and y both have the same minimal support, i.e. there is no other flow $z \in F(PN)$ with $\text{sup}(z) \subsetneq \text{sup}(x) = \text{sup}(y)$, then there is $n \in \mathbb{N}$ s.t. either $x = ny$ or $y = nx$.*

Analogously to Theorems 1 and 2 we have the following for weak minimality, again adapted from [52].

Theorem 4. $M_wTF(PN)$ and $M_wPF(PN)$ are finite and unique.

Theorem 5. *For any flow $x \in F(PN)$ there are $a_1, \dots, a_k \in \mathbb{N}$ and weakly minimal flows $x_1, \dots, x_k \in M_wF(PN)$ s.t. $x = a_1x_1 + \dots + a_kx_k$.*

In contrast to standard minimality, weakly minimal flows can generate any flows without division and by natural-number linear combination alone. This simplicity comes at a price because weak minimality is harder to compute than standard minimality [52]. In general, standard minimality implies weak minimality, and the set of standard minimal flows may be strictly contained in the set of weakly minimal flows.

7.2 Flow Matrix Composition and Modular Duality

The concrete Petri net semantics of LBS given in the previous chapter includes a composition operator which merges the places of component Petri nets using set operations, and we have seen an example of how this works in Figure 7.1.1. We now recast the definition of this composition operator in terms of flow matrices. We also see that P and T-flows are *not* dual in a modular sense when considering place sharing alone, but that the duality arises when also considering transition sharing. This allows our results for place sharing to be adapted to transition sharing which may be of general interest outside of the biological domain.

7.2.1 Matrix-Based Composition With Place Sharing

In this section and throughout the chapter, we let $|_s$ denote the place-based composition operator $|_{\mathcal{PN}}$ defined in the previous chapter. We start by considering the structure of the full flow matrix W arising from the composition $PN_1|_s PN_2$ of PN_1 and PN_2 with flow matrices W_1 and W_2 . We say that two ordered Petri nets (or more generally,

two linearly ordered sets of places) are *composable* if the shared places are ordered *after* the non-shared places in PN_1 and *before* the non-shared places in PN_2 , and if the orderings agree on shared places. More precisely, for $\Delta S = S_{PN_1} \cap S_{PN_2}$ and all $s_1 \in S_{PN_1} \setminus \Delta S$, $s, s' \in \Delta S$ and $s_2 \in S_{PN_2} \setminus \Delta S$ it must hold that $s_1 \prec_{S_{PN_1}} s, s \prec_{S_{PN_2}} s_2$ and $s \prec_{S_{PN_1}} s' \Leftrightarrow s \prec_{S_{PN_2}} s'$. In the running example the composability condition can for example be satisfied by ordering the respective places as (Photons, CE, H₂O, O₂, CO₂, Sugar) and (H₂O, O₂, CO₂, Sugar, ChE, Heat). One can of course always chose appropriate orderings that make two Petri nets composable, and we do so in Section 5 in the context of concrete flow semantics of LBS; until then, we assume that any two Petri nets under consideration are composable.

Under this assumption, the matrices W_1 , W_2 and W can be partitioned as follows where, for $i \in \{1, 2\}$, W_i^s consists of the rows from W_i which represent shared places, and W_i^- are the remaining rows for non-shared places.

$$W_1 = \begin{bmatrix} W_1^- \\ W_1^s \end{bmatrix}, W_2 = \begin{bmatrix} W_2^s \\ W_2^- \end{bmatrix}, W = \begin{bmatrix} W_1^- & 0 \\ W_1^s & W_2^s \\ 0 & W_2^- \end{bmatrix}$$

When considering parallel compositions in the following sections, we write W_1^- , W_1^s , W_2^- , W_2^s and W with the above meaning in mind. We furthermore let W_1^+ and W_2^+ denote respectively the left and right partition of W , i.e. the extensions of W_1 and W_2 with 0-entries for non-shared places from the parallel counterpart. W^s denotes the rows $W_1^s W_2^s$ for shared places, and W^- denotes W without these rows.

7.2.2 Modular Duality: Composition With Transition Sharing

As mentioned in Section 1, T-flows and P-flows are duals. A natural question arises of whether this duality holds in the *modular* sense that:

$$PF(PN_1 \mid_s PN_2) = TF(PN_1^D \mid_s PN_2^D)$$

The answer is *no*. To see why, let us assume that $T_{PN_1} \cap T_{PN_2} = \emptyset$ and write out the flow matrices W^T of $(PN_1 \mid_s PN_2)^D$ and W' of $(PN_1^D \mid_s PN_2^D)$:

$$W^T = \begin{bmatrix} W_1^{-T} & W_1^{sT} & 0 \\ 0 & W_2^{sT} & W_2^{-T} \end{bmatrix} \quad W' = \begin{bmatrix} W_1^{-T} & W_1^{sT} & 0 & 0 \\ 0 & 0 & W_2^{sT} & W_2^{-T} \end{bmatrix}$$

The two matrices do not generally have the same dimensions because the dual nets PN_1^D and PN_2^D share transitions rather than places. Hence the modular duality suggested above clearly does not hold in general.

However, there is a *transition-based composition operator*, $|_t$, where transitions, rather than places, of parallel components are merged. It is defined exactly as for the place-based composition $|_s$, except that the sets of places of the two components are required to be disjoint rather than the sets of transitions.

Then the *P-flows* of a composite Petri net under *transition sharing* are the same as the *T-flows* of the composite dual Petri nets under *place sharing*, and symmetrically for T-flows under transition sharing:

Theorem 6. *Let PN_1 and PN_2 be Petri nets. Then*

1. $TF(PN_1 |_t PN_2) = PF(PN_1^D |_s PN_2^D)$.
2. $PF(PN_1 |_t PN_2) = TF(PN_1^D |_s PN_2^D)$.

The proof relies on a partitioning of flow matrices similar to the partitioning in the previous subsection, but for shared transitions rather than places. It follows from this theorem that the results for modular flows under place sharing, to be given in the following sections, can be adapted to (dual) modular flows under transition sharing.

7.3 Modular Minimal T-Flows

7.3.1 The Intuition

We start with an example of how T-flows arise from a parallel composition. As we observed earlier, neither of the photosynthesis and respiration Petri nets in Figure 7.1.1 on page 126 has any T-flows. However, the composite Petri net *does* have a single minimal T-flow determined by the transitions $(t_1, t_2, t_3, t_4, t_5, t_6)$. To see how this flow arises from the parallel composition, we must look at *potential* T-flows of the two nets rather than the *actual* T-flows of which there are none. The potential T-flows are the ones arising from restricting individual components to private places only, i.e. by disregarding the shared places. If we do so, the photosynthesis Petri net has a single minimal T-flow determined by (t_1, t_2, t_3) , and the respiration Petri net has a single minimal T-flow determined by (t_4, t_5, t_6) . The minimal T-flow in the composite Petri net can be composed from these two because the transitions from the two Petri nets operating on shared places cancel each other out.

The general case is slightly more complicated because there may be many potential minimal T-flows of each restricted parallel component. These T-flows can then be

combined by natural-number linear combinations in such a way that the resulting flow has no net effect on shared places. The weights of this natural-number linear combination must be minimal in some sense in order for there to be any hope of minimality of the composite flow in the composite Petri net.

7.3.2 The Definition

A formal definition is given below, where we use the conventions on flow matrix partitioning introduced in Section 7.2.1. We write $[MTF(W_i^-)]$ for the matrix consisting of the column vectors in $MTF(W_i^-)$ in some arbitrary order.

Definition 20. Let PN_1 and PN_2 be Petri nets and let $X_1 = [MTF(W_1^-)]$, $X_2 = [MTF(W_2^-)]$ and W^s be given. Define the following:

1. $X \triangleq \begin{bmatrix} X_1 & 0 \\ 0 & X_2 \end{bmatrix}$
2. $C \triangleq W^s X$
3. $Z \triangleq \{X\alpha \mid \alpha \in MTF(C)\}$

We then define $MTF^{\text{Par}}(X_1, X_2, W^s) \triangleq \min(Z)$.

To elaborate on this definition, let $m_1 = |T_{PN_1}|$, $m_2 = |T_{PN_2}|$, $n_1 = |MTF(W_1^-)|$ and $n_2 = |MTF(W_2^-)|$. Then X_1 and X_2 are $m_1 \times n_1$ and $m_2 \times n_2$ matrices with the minimal T-flows of respectively PN_1 and PN_2 without their shared places, i.e. of W_1^- and W_2^- . Also,

1. X is an $(m_1 + m_2) \times (n_1 + n_2)$ matrix with columns representing minimal T-flows of W^- .
2. C is an $(|S_{PN_1} \cap S_{PN_2}|) \times n_1 + n_2$ matrix with each column c_i representing the effect of the corresponding minimal T-flow x_i on the shared places.
3. Z is a set of linear combinations of the minimal T-flow-columns in X . These linear combinations are chosen in such a way that they have no net effect on the shared species. Note that the set Z is well-defined because $MTF(C)$ is finite and unique by Theorem 1.

Remarks on the use of minimisation follow towards the end of the section.

7.3.3 Results

The following results state that Definition 20 is sound and complete. Soundness is split into two lemmas, the first of which is needed to prove completeness.

Lemma 1 (Soundness part 1). *Let PN_1 , PN_2 and Z be as given in Definition 20. Then*

1. $Z \subsetneq TF(PN_1 |_s PN_2)$
2. $\min(Z) \subsetneq TF(PN_1 |_s PN_2)$

The proof uses the definition of C to show that any $X\alpha \in Z$ is a T-flow of W^s . Since X consists of minimal T-flows of W^- , $X\alpha$ is also a T-flow of W^- . Together these give that $X\alpha$ is a T-flow of W and hence of $PN_1 |_s PN_2$.

Lemma 2 (Completeness). *Let PN_1 , PN_2 , X_1 , X_2 and W^s be as given in Definition 20. Then $MTF(PN_1 |_s PN_2) \subseteq MTF^{\text{Par}}(X_1, X_2, W^s)$.*

The proof starts by showing that any $x \in MTF(PN_1 |_s PN_2)$ can be written $x = \frac{1}{a}X\alpha$ where $\alpha \in TF(C)$ and $a \in \mathbb{N}$ (uses Theorem 2 and the definition of C). Using Euclid's lemma and that x is canonical, we show that a canonical α can be chosen. We then use Theorem 3 and minimality of x to show that any of the minimal-support α which generate x as above is in fact also minimal in C . We arrive at $x \in Z$. To conclude that also $x \in \min(Z)$, we use that any $x' \in Z$ with a support contained in that of x would also be in $TF(PN_1 |_s PN_2)$ (Lemma 1), hence contradicting minimality of x in $PN_1 |_s PN_2$.

Lemma 3 (Soundness part 2). *Let PN_1 , PN_2 , X_1 , X_2 and W^s be as given in Definition 20. Then $MTF^{\text{Par}}(X_1, X_2, W^s) \subseteq MTF(PN_1 |_s PN_2)$.*

The proof carries on from Lemma 1. To show that the elements of $\min(Z)$ are in fact minimal in $PN_1 |_s PN_2$, we use that all minimal-support (although not necessarily canonical) flows are represented in Z by completeness (Lemma 2).

Together the two previous lemmas prove our main T-flow theorem:

Theorem 7 (Soundness and completeness). *Let PN_1 , PN_2 , X_1 , X_2 and W^s be as given in Definition 20. Then $MTF^{\text{Par}}(X_1, X_2, W^s) = MTF(PN_1 |_s PN_2)$.*

The size of the matrix X , and hence of C , may be reduced by removing columns for transitions which have no effect on any of the shared places; these columns are also flows in the composite Petri net and can be included directly.

The flows in Z may not be minimal, which is why the minimisation function must be applied as a last step. This is illustrated by the two Petri nets in Figure 7.4.1: the left, PN_1 , has two places of which one is shared with the right, PN_2 , consisting of just a single place. The restriction of PN_1 to the place p_1 (corresponding to W^-) has four minimal T-flows represented by $x_1 = (t_1, t_2)$, $x_2 = (t_2, t_3)$, $x_3 = (t_3, t_4)$ and $x_4 = (t_1, t_4)$. The “minimal” combinations of these which cancel out the effect on the shared place p_2 (corresponding to the minimal flows of C) are $x_1 + x_2 = (t_1, 2 \cdot t_2, t_3)$, $x_1 + x_3 = (t_1, t_2, t_3, t_4)$ and $x_2 + x_4 = (t_1, t_2, t_3, t_4)$. But the latter two flows are not minimal because their supports strictly contain the support of the first.

Minimisation is however *not* necessary in cases where the minimal flows in X are linearly independent. Then we get unique decomposition in the sense that any flow can be written uniquely as combinations of minimal flows (linear independence fails in the above example, for $x_1 + x_3 = x_2 + x_4$). This can be used in the proof of the following theorem:

Theorem 8. *Let X and Z be as given in Definition 20. If the columns of X are linearly independent, then*

1. **For standard minimality:** *The elements of Z have minimal support (but still may not be canonical).*
2. **For weak minimality:** $\min_w(Z) = Z$.

7.4 Modular Minimal P-Flows

7.4.1 The Intuition

As for T-flows, we start with an example of how the P-flows of a composite Petri net arise from the P-flows of its components. In Section 1 we listed the three minimal P-flows of each of the two Petri nets in Figure 7.1.1 on page 126, including $x = (\text{CE}, \text{H}_2\text{O}, \text{Sugar})$ from the first Petri net and $y = (\text{H}_2\text{O}, \text{Sugar}, \text{ChE})$ from the second Petri net. Neither is a P-flow in the composite Petri net because of interference from the additional transitions. For example, t_4 consumes tokens from Sugar and produces tokens in ChE, and this violates the first P-flow. However, because x and y are “consistent” in the sense that they have identical weights for their shared places (namely $1 \cdot \text{H}_2\text{O}$ and $1 \cdot \text{Sugar}$), we can “join” them to obtain a new minimal P-flow $(\text{CE}, \text{H}_2\text{O}, \text{Sugar}, \text{ChE})$ for the composite Petri net.

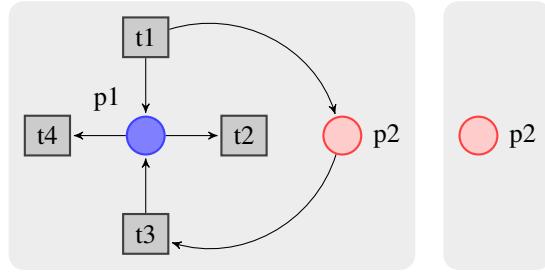


Figure 7.4.1: Two Petri nets illustrating how Definition 20 can give rise to non-minimal flows in Z .

In the general case, not all pairs of minimal P-flows from the two components are consistent and it is not sufficient to only join those that are. Instead we must obtain two linear combinations of minimal P-flows from the respective Petri nets in such a way that they become consistent, and then join them to form a P-flow of the composite Petri net. As for modular T-flows, the weights used in this linear combination must be minimal in some sense in order for there to be any hope of minimality of the resulting join.

7.4.2 The Definition

Here is the formal definition of P-flow joins where again we assume the partitioning of flow matrices given in Section 7.2.1.

Definition 21 (Flow joins). *Let PN_1 and PN_2 be Petri nets and let $x \in PF(PN_1)$, $y \in PF(PN_2)$ and $n = |SP_{N_1} \cap SP_{N_2}|$ be given. Write $x = (x^- x^s)$ and $y = (y^s y^-)$ where x^s consists of the last n components of x and y^s consists of the first n components of y , hence representing the places shared between PN_1 and PN_2 . If $x^s = y^s$ we say that x and y are consistent and define their join $x \stackrel{n}{\smile} y \stackrel{\Delta}{=} (x^- x^s y^-)$.*

When the size n of the set of shared places of the component Petri nets is understood from the context, we write $x \smile y$ instead of $x \stackrel{n}{\smile} y$.

The general modular definition of P-flows is given below. Similarly to the definition for T-flows, $[MPF(PN)]$ is a matrix with rows from $MPF(PN)$ in any order.

Definition 22. *Let PN_1 and PN_2 be Petri nets and let $X = [MPF(PN_1)]$, $Y = [MPF(PN_2)]$ and $n = |SP_{N_1} \cap SP_{N_2}|$ be given. Let X^s and Y^s be the sub-matrices of X and Y containing only columns for shared places as determined by n . Define the following:*

$$1. \ C \stackrel{\Delta}{\simeq} \begin{bmatrix} X^s \\ -Y^s \end{bmatrix}$$

$$2. \ Z \stackrel{\Delta}{\simeq} \{\alpha X \xrightarrow{n} \beta Y \mid (\alpha\beta) \in MPF(C)\}$$

We then define $MPF^{\text{Par}}(X, Y, n) \stackrel{\Delta}{\simeq} \min(Z)$.

To elaborate on this definition, let $m_1 = |MPF(PN_1)|$ and $m_2 = |MPF(PN_2)|$. Then

1. C is an $(m_1 + m_2) \times n$ matrix with the first m_1 rows representing minimal P-flows of PN_1 and the last m_2 rows representing *negated* minimal P-flows from PN_2 , but restricted to the n shared places only.
2. Z contains the joins of consistent linear combinations of flows from the two Petri nets. The weights for this linear combination are chosen exactly so that the resulting flows have the same weights for shared places. Note that the set Z is well-defined because $MTF(C)$ is finite and unique by Theorem 1.

7.4.3 Results

The join of consistent P-flows from two Petri nets is also a P-flow in the composite Petri net:

Lemma 4. *Let PN_1 and PN_2 be Petri nets and let $x \in PF(PN_1)$ and $y \in PF(PN_2)$. If x and y are consistent then $x \frown y \in PF(PN_1 \mid_s PN_2)$.*

Conversely, any P-flow z of a composite Petri net $PN_1 \mid_s PN_2$ is the join of a P-flow from PN_1 (or 0) and a P-flow from PN_2 (or 0):

Lemma 5. *Let PN_1 and PN_2 be Petri nets and let $z \in PF(PN_1 \mid_s PN_2)$. Then there are $x \in PF(PN_1) \cup \{0\}$ and $y \in PF(PN_2) \cup \{0\}$ s.t. $z = x \frown y$.*

As for T-flows we have soundness and completeness results, and soundness is split into two separate lemmas.

Lemma 6 (Soundness part 1). *Let Z be as given in Definition 22. Then*

1. $Z \subsetneq PF(PN_1 \mid_s PN_2)$
2. $\min(Z) \subsetneq PF(PN_1 \mid_s PN_2)$

The proof uses Lemma 4 and the definition of C .

Lemma 7 (Completeness). *Let PN_1, PN_2, X, Y and n be as given in Definition 22. Then $MPF(PN_1 \mid_s PN_2) \subseteq MPF^{\text{Par}}(X, Y, n)$.*

The proof first uses Lemma 5 to write any $z \in MPF(PN_1 \mid_s PN_2)$ as $z = x \frown y$ for some $x \in PF(PN_1)$ and $y \in PF(PN_2)$. Then the main challenge is to show that there is some d and $(\alpha\beta) \in MPF(C)$ s.t. $dx = \alpha X$ and $dy = \beta Y$ (for then we can conclude that $dz \in Z$). First the existence of such $(\alpha\beta) \in PF(C)$ is shown using Theorem 2, the definition of C and the fact that dx and dy are consistent. Minimality of $(\alpha\beta)$ uses an idea similar to the proof of Lemma 2 (completeness for T-flows).

Lemma 8 (Soundness part 2). *Let PN_1, PN_2, X, Y and n be as given in Definition 22. Then $MPF^{\text{Par}}(X, Y, n) \subseteq MPF(PN_1 \mid_s PN_2)$.*

The proof is similar to that of Lemma 3 (soundness for T-flows). Together the last two lemmas prove our main theorem on modular P-flows:

Theorem 9 (Soundness and completeness). *Let PN_1, PN_2, X, Y , and n be as given in Definition 22. Then $MPF^{\text{Par}}(X, Y, n) = MPF(PN_1 \mid_s PN_2)$*

The matrix C in Definition 22 can be reduced in size by removing rows with all 0 entries. Because the corresponding minimal P-flows do not involve shared places, they are also minimal P-flows of the composite Petri net and can be included directly.

As for the modular definition of T-flows, minimisation is not necessary in cases where the minimal P-flows in the rows of C are linearly independent:

Theorem 10. *Let C and Z be as given in Definition 22. If the rows of C are linearly independent, then*

1. **For standard minimality:** The elements of Z have minimal support (but still may not be canonical).
2. **For weak minimality:** $\min_w(Z) = Z$.

7.5 Concrete Semantics of LBS

The definitions of minimal flows given in the previous two sections are not modular in the strongest sense of the word for two reasons. First, the minimal flows of parallel components are given explicitly and not defined inductively; this is because there is no inductive structure on Petri nets and flows per se. Second, in the case of T-flows, the

MTF^{Par} function requires more than just the minimal T-flows of parallel components to be given: it requires the minimal T-flows of the components *without shared places* (which are super-sets of the minimal T-flows of the components) and the flow matrix for shared places.

The first issue is addressed by defining concrete minimal T-flow and P-flow semantics for LBS, thus leveraging the general, modular semantics of LBS. The second issue is addressed by defining semantical minimal T-flow objects which do not just encapsulate minimal T-flows, but rather the flow matrix of the net together with a *function* mapping shared places to minimal T-flows of the restricted Petri net without these places.

The ordering of Petri net places must now be taken into account in order to ensure that two components satisfy the condition for composition. For this we need the notion of an *n-ary permutation* θ_n , where $n \in \mathbb{N}$, which is a bijective function of the form $\theta_n(q) = q'$ where $q, q' \in \{1 \dots n\}$. We extend *n*-ary permutations to *n*-ary vectors in an evident manner. We also extend *m*-ary permutations to $m \times n$ matrices, here with matrix rows being subject to permutation.

We can obtain an $|S|$ -ary permutation from a finite set S and two linear orderings, \prec and \prec' , on S as follows; we write $lst(S, \prec)$ for the list representation of a linearly ordered set (S, \prec) :

$$perm(S, \prec, \prec') = \{q \mapsto q' \mid lst(S, \prec).q = lst(S, \prec').q'\}$$

7.5.1 The Concrete Minimal T-Flow Semantics of LBS

We start with a definition of the concrete semantical objects encapsulating minimal T-flows. In addition to a flow matrix and a function mapping sets of places to actual minimal T-flows, our semantical objects also contain linearly ordered sets of places and transitions used to match entries in flows and flow matrices to the corresponding species and transitions.

Definition 23. An LBS-T-flow structure LTF is a tuple $(S, \prec_S, T, \prec_T, W, h)$ where

- $S \subsetneq_{\text{fin}} \{MS(v_{\text{gns}}) \mid v_{\text{gns}} \in V_{\text{gns}}\}$ is the set of places and \prec_S is a linear ordering on S .
- $T \subsetneq_{\text{fin}} \{0, 1\}^*$ is the set of transitions and \prec_T is a linear ordering on T .
- W is a net flow matrix on (S, \prec_S) and (T, \prec_T) (as described in Subsection 7.1.1).

- h is a function of the form $h(\Delta S) = MTF$ where $\Delta S \subseteq S$ and $MTF \subsetneq_{\text{fin}} \mathbb{N}^{|T|^T}$ is a set of T -flows.

We use the notation S_{LTF} to refer to the places S of a T-flow structure LTF , and similarly for the other components. The set of all T-flow structures is denoted by \mathcal{LTF} .

In the following definition of the T-flow concrete semantics of LBS, we let $PN \setminus \Delta S$ be the Petri net PN without the places ΔS .

Definition 24. *The concrete semantics for LBS in terms of T-flow structures is given by the tuple $(\mathcal{LTF}, |_{\mathcal{LTF}}, \mathbf{0}_{\mathcal{LTF}}, G_{\mathcal{LTF}}, I_{\mathcal{LTF}})$ where*

- $LTF_1 |_{\mathcal{LTF}} LTF_2 \xrightarrow{\Delta} LTF$ where
 - \prec_1 and \prec_2 are arbitrary linear orderings on respectively S_{LTF_1} and S_{LTF_2} for which (S_{LTF_1}, \prec_1) and (S_{LTF_2}, \prec_2) are composable.
 - $\theta_1 \xrightarrow{\Delta} \text{perm}(S_{LTF_1}, \prec_1, \prec_{S_{LTF_1}})$
 - $\theta_2 \xrightarrow{\Delta} \text{perm}(S_{LTF_2}, \prec_2, \prec_{S_{LTF_2}})$
 - $S_{LTF} \xrightarrow{\Delta} S_{LTF_1} \cup S_{LTF_2}$
 - $\prec_{S_{LTF}} \xrightarrow{\Delta} \prec_1 \cup \prec_2 \cup \{(s_1, s_2) \mid s_1 \in S_{LTF_1} \setminus S_{LTF_2} \wedge s_2 \in S_{LTF_2} \setminus S_{LTF_1}\}$
 - $T_{LTF} \xrightarrow{\Delta} T_{LTF_1} \cup T_{LTF_2}$
 - $\prec_{T_{LTF}} \xrightarrow{\Delta} \prec_{T_{LTF_1}} \cup \prec_{T_{LTF_2}} \cup \{(t_1, t_2) \mid t_1 \in T_{LTF_1} \wedge t_2 \in T_{LTF_2}\}$
 - W_{LTF} is composed from $\theta_1(W_{LTF_1})$ and $\theta_2(W_{LTF_2})$ as defined in Subsection 7.2.1.
 - $h_{LTF}(\Delta S) = MTF^{\text{Par}}([MTF_1], [MTF_2], W^{\text{ss}})$
where
 - * $\Delta S'' \xrightarrow{\Delta} S_{LTF_1} \cap S_{LTF_2}$
 - * $\Delta S' \xrightarrow{\Delta} \Delta S \cup \Delta S''$
 - * $MTF_1 \xrightarrow{\Delta} h_{LTF_1}(\Delta S')$
 - * $MTF_2 \xrightarrow{\Delta} h_{LTF_2}(\Delta S')$
 - * W^{ss} is the sub-matrix of W^s containing rows for shared places $\Delta S'' \setminus \Delta S$.

if $T_{PN_1} \cap T_{PN_2} = \emptyset$

- $\mathbf{0}_{\mathcal{LTF}} \xrightarrow{\Delta} (\emptyset, \emptyset, \emptyset, \emptyset, [], h_{LTF})$ where

- $h_{LTF}(\emptyset) = \{()\}$
- $G_{LTF}(\underline{n \cdot v_{gns}} \Rightarrow^{v_{gr}} \underline{n' \cdot v'_{gns}}, t) \stackrel{\Delta}{\simeq} LTF$ where
 - $PN \stackrel{\Delta}{\simeq} G_{\mathcal{PN}}(n \cdot v_{gns} \Rightarrow^{v_{gr}} n' \cdot v'_{gns}, t)$
 - $S_{LTF} \stackrel{\Delta}{\simeq} S_{PN}$
 - $\prec_{S_{LTF}}$ is an arbitrary linear ordering on S_{LTF} .
 - $T_{LTF} \stackrel{\Delta}{\simeq} T_{PN}$
 - $\prec_{T_{LTF}}$ is an arbitrary linear ordering on T_{LTF} .
 - W_{LTF} is the flow matrix of $(PN, \prec_{S_{LTF}}, \prec_{T_{LTF}})$.
 - $h_{LTF}(\Delta S) \stackrel{\Delta}{\simeq} MTF(PN \setminus \Delta S)$
- $I_{LTF}(v_{gns}, n) \stackrel{\Delta}{\simeq} (\{s\}, \{(s, s)\}, \emptyset, \emptyset, [], h_{LTF})$ where
 - $s \stackrel{\Delta}{\simeq} \{MS(v_{gns})\}$
 - $h_{LTF}(\Delta S) = \{()\}$

In the case of parallel composition, linear orderings on the places of the components which satisfy the composability condition, as defined Subsection 7.2.1, are arbitrarily chosen. They are then combined to linear orderings on the places of the composite T-flow structure. The chosen linear orderings are also used to obtain permutations of the flow matrices for the two components before they are composed. Observe that the flow matrix of the entire Petri net, and the flows arising from any restriction of places, are indeed necessary in order to define parallel composition. Hence modularity of T-flows comes at a high price.

In the case of reactions, the corresponding Petri net is obtained using the concrete Petri net semantics given in the previous chapter. This Petri net, together with an arbitrary choice of orderings, is then used to construct a flow matrix. The Petri net is also used for obtaining the actual minimal T-flows after removing the given set of places. A more direct construction would also have been possible because there is only one candidate minimal T-flow to test.

7.5.2 The Concrete Minimal P-Flow Semantics of LBS

The concrete semantical objects encapsulating P-flows are somewhat simpler than for T-flows.

Definition 25. An LBS-P-flow structure LPF is a tuple (S, \prec_S, MPF) where

- $S \subsetneq_{\text{fin}} \{MS(v_{\text{gns}}) \mid v_{\text{gns}} \in V_{\text{gns}}\}$ is the set of places and \prec_S is a linear ordering on S .
- $MPF \subsetneq_{\text{fin}} \mathbb{N}^{|S|}$ is a set of P-flows.

We use the notation S_{LPF} to refer to the places S of a P-flow structure LPF , and similarly for the other components. The set of all P-flow structures is denoted by \mathcal{LPF} .

Definition 26. The concrete semantics for LBS in terms of P-flow structures is given by the tuple $(\mathcal{LPF}, |\mathcal{LPF}|, \mathbf{0}_{\mathcal{LPF}}, G_{\mathcal{LPF}}, I_{\mathcal{LPF}})$ where

- $LPF_1 |_{\mathcal{LPF}} LPF_2 \stackrel{\Delta}{\simeq} LPF$ where
 - \prec_1 and \prec_2 are arbitrary linear orderings on respectively S_{LPF_1} and S_{LPF_2} for which (S_{LPF_1}, \prec_1) and (S_{LPF_2}, \prec_2) are composable.
 - $\theta_1 \stackrel{\Delta}{\simeq} \text{perm}(S_{LPF_1}, \prec_1, \prec_{S_{LPF_1}})$
 - $\theta_2 \stackrel{\Delta}{\simeq} \text{perm}(S_{LPF_2}, \prec_2, \prec_{S_{LPF_2}})$
 - $S_{LPF} \stackrel{\Delta}{\simeq} S_{LPF_1} \cup S_{LPF_2}$
 - $\prec_{S_{LPF}} \stackrel{\Delta}{\simeq} \prec_1 \cup \prec_2 \cup \{(s_1, s_2) \mid s_1 \in S_{LPF_1} \setminus S_{LPF_2} \wedge s_2 \in S_{LPF_2} \setminus S_{LPF_1}\}$
 - $MPF_{LPF} \stackrel{\Delta}{\simeq} MPF^{\text{Par}}([MPF_1], [MPF_2], n)$ where
 - * $MPF_1 \stackrel{\Delta}{\simeq} \theta_1(MPF_{LPF_1})$
 - * $MPF_2 \stackrel{\Delta}{\simeq} \theta_2(MPF_{LPF_2})$
 - * $n \stackrel{\Delta}{\simeq} |S_{LPF_1} \cap S_{LPF_2}|$
- $\mathbf{0}_{\mathcal{LPF}} \stackrel{\Delta}{\simeq} (\emptyset, \emptyset, \{\()\})$
- $G_{\mathcal{LPF}}(\underline{n \cdot v_{\text{gns}}} \Rightarrow^{v_{\text{gr}}} \underline{n' \cdot v'_{\text{gns}}}, t) \stackrel{\Delta}{\simeq} LPF$ where
 - $PN \stackrel{\Delta}{\simeq} G_{\mathcal{PN}}(\underline{n \cdot v_{\text{gns}}} \Rightarrow^{v_{\text{gr}}} \underline{n' \cdot v'_{\text{gns}}}, t)$
 - $S_{LPF} \stackrel{\Delta}{\simeq} S_{PN}$
 - $\prec_{S_{LPF}}$ is an arbitrary linear ordering on S_{LPF} .
 - $MPF_{LPF} \stackrel{\Delta}{\simeq} MPF(PN)$
- $I_{\mathcal{LPF}}(v_{\text{gns}}, n) \stackrel{\Delta}{\simeq} (\{s\}, \{(s, s)\}, \{(1)\})$ where

$$- s \stackrel{\Delta}{\simeq} \{MS(v_{gns})\}$$

In the case of parallel composition, linear orderings on the places of the components which satisfy the compositability conditions, as defined Subsection 7.2.1, are arbitrarily chosen and combined to linear orderings for the composite P-flow structure. The chosen linear orderings are then used to obtain two permutations which are applied to the flows of the respective components before composition. Note how the definition for parallel composition is somewhat simpler than in the case of T-flows. This illustrates how modular T and P-flows are intricately different and non-dual because more information is needed in the modular definition of T-flows.

The case of reactions obtains the Petri net assigned to the reaction by the concrete Petri net semantics given in the previous chapter and uses this to obtain the corresponding minimal P-flows. In contrast to the corresponding case for T-flows, there may be more than one minimal P-flow associated with a reaction.

7.5.3 Results

The following theorem states that the concrete flow semantics given above work as expected. We let $\llbracket \cdot \rrbracket_{\mathcal{PN}}$, $\llbracket \cdot \rrbracket_{\mathcal{LT}\mathcal{F}}$ and $\llbracket \cdot \rrbracket_{\mathcal{LP}\mathcal{F}}$ be the general denotation function instantiated with respectively the concrete Petri net semantics, the concrete T-flow semantics and the concrete P-flow semantics.

Theorem 11. *Let P be an LBS program, let $PN \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\mathcal{PN}}$, $LTF \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\mathcal{LT}\mathcal{F}}$, $LPF \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\mathcal{LP}\mathcal{F}}$, $\Delta S \subseteq S_{PN}$ and let \prec_T be an arbitrary linear ordering on T_{PN} . Then*

1. $h_{LTF}(\Delta S) = MTF(OPN)$ where $OPN \stackrel{\Delta}{\simeq} (PN \setminus \Delta S, \prec_{S_{LTF}}, \prec_{T_{LTF}})$
2. $MPF_{LPF} = MPF(OPN)$ where $OPN \stackrel{\Delta}{\simeq} (PN, \prec_{S_{LPF}}, \prec_T)$

It follows as a special case of 1) that $h_{LTF}(\emptyset)$ gives the minimal T-flows of the full Petri net. The proof is by induction on the structure of LBS programs, using Theorems 7 and 9.

7.6 Related Work

The idea that consistent P-flows from two components can be joined to form a P-flow in the composite Petri net (Lemma 4) is not new. Neither is the converse that any P-flow

in a composite Petri net is a join of P-flows from the parallel components (Lemma 5). These results have been stated previously in some form in [49, 11, 58, 23, 91].

In [58] an algorithm is given for directly computing the minimal P-flows of a *well-formed net* resulting from a place fusion operation (i.e. the merging of two places within a single Petri net) based on the minimal P-flows of the Petri net before fusion. But no proof of correctness is given. In [11] a method similar to Definition 22 is proposed for generative sets of P-flows rather than minimal P-flows. Such a method is also presented for *functional subnets* in [91], which in addition considers how to obtain appropriate components given a flat Petri net. However, in neither case is it clear to us how completeness follows from the proofs given, i.e. that the method does in fact result in generative families of P-flows. In contrast, we give a full proof of *minimality* of the resulting P-flows (which is stronger than generativity).

Modular definitions of T-flows have received somewhat less attention than P-flows in the literature. To our knowledge, the only existing explicit work on modular T-flows is [58] (for well-formed nets), but this only shows an *example* of how new T-flows can arise after a place fusion. No general definition is given. In [23], a characterisation of *P-flows* arising from a composition of modules is given based on *both* place sharing *and* transition sharing. The duality elucidated in Theorem 6 suggests that a dual characterisation can be given for *T-flows* under place and transition sharing. Nevertheless, the characterisation does not result in methods for finding minimal or generative sets of flows and hence is of little practical use. It also considers flows in \mathbb{Z} rather than in \mathbb{N} as is more common.

Chapter 8

GEC by Example

We now turn to synthetic biology and the language for *Genetic Engineering of Cells* (GEC). A GEC program can be translated to a set of *devices*, each representing a possible in-vivo implementation which satisfies the constraints of the GEC program. More precisely, a device consists of a set of sequences of genetic parts such as promoters, ribosome binding sites, protein coding regions and terminators as introduced in Section 2.4. Multiple sequences are necessary in e.g. the case of multi-cellular systems, or more generally when different genes are located on different plasmids (i.e., circular sequences of DNA which typically host genes in bacteria).

A GEC program can also be translated to a set of reactions for each device, represented by simplified forms of LBS programs, allowing the dynamics of gene expression to be simulated. We thereby envision an iterative process of translation, simulation and refinement where each cycle refines a GEC program by e.g. introducing additional constraints that rule out devices with undesired simulation behaviour. Following completion of the necessary iterations, a specific device can be chosen and implemented experimentally in living cells. This process is represented schematically by the flow chart in Figure 8.0.1.

The translation to devices and reactions relies on a database of genetic parts with their relevant properties, and on a database of known reactions. Prototype databases, which form the basis of our discussion, are presented in Section 1. In Section 2 we introduce the basic language constructs of GEC through a number of small examples. Sections 3 and 4 then present two case studies drawn from the existing literature, namely of the repressilator and the predator-prey system.

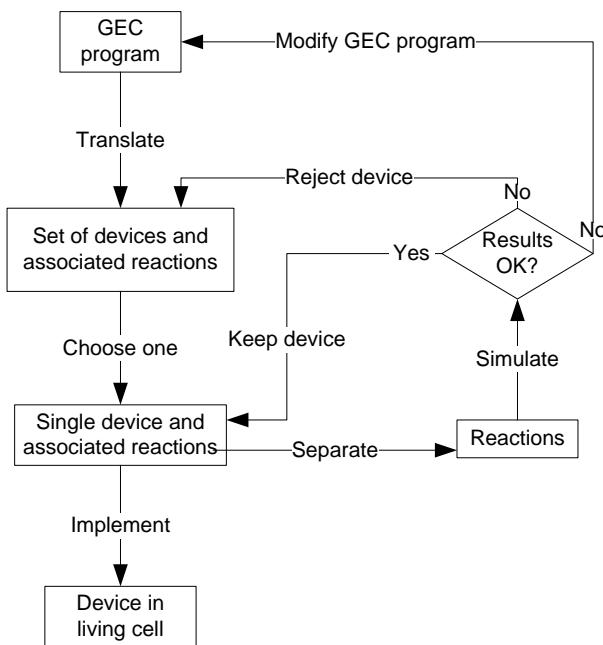


Figure 8.0.1: Flow chart illustrating the process of translation, simulation and refinement of GEC programs, ending with an implementation of a device in a living cell.

Table 8.0.1: A prototype reaction database consisting of basic reactions, enzymatic reactions with enzymes preceding the \sim symbol, and transport reactions with compartments represented by square brackets. Reaction rate constants are enclosed in curly brackets.

$\text{luxR} + \text{m3OC6HSL} \rightarrow_{\{0.5\}} \text{luxR-m3OC6HSL}$ $\text{luxR-m3OC6HSL} \rightarrow_{\{1.0\}} \text{luxR} + \text{m3OC6HSL}$ $\text{lasR} + \text{m3OC12HSL} \rightarrow_{\{0.5\}} \text{lasR-m3OC12HSL}$ $\text{lasR-m3OC12HSL} \rightarrow_{\{1.0\}} \text{lasR} + \text{m3OC12HSL}$ $\text{luxI} \sim \rightarrow_{\{1.0\}} \text{m3OC6HSL}$ $\text{lasI} \sim \rightarrow_{\{1.0\}} \text{m3OC12HSL}$ $\text{ccdB} \sim \text{ccdB} \rightarrow_{\{1.0\}}$ $\text{ccdB2} \sim \text{ccdB} \rightarrow_{\{0.00001\}}$ $\text{m3OC6HSL} \rightarrow_{\{0.5\}} [\text{m3OC6HSL}]$ $\text{m3OC12HSL} \rightarrow_{\{0.5\}} [\text{m3OC12HSL}]$ $[\text{m3OC6HSL}] \rightarrow_{\{0.5\}} \text{m3OC6HSL}$ $[\text{m3OC12HSL}] \rightarrow_{\{0.5\}} \text{m3OC12HSL}$

8.1 The Databases

We have chosen prototype reaction and parts databases with the minimal structure and content necessary to convey the main ideas behind GEC. Technically the databases are implemented in Prolog, but their informal, tabular representations are shown in Tables 8.0.1 and 8.1.1. We stress that the listed rate constants are hypothetical, and that specific rates have been chosen for the sake of example.

8.1.1 The Reaction Database

The reaction table represents reactions in the general form:

$$\text{enzymes} \sim \text{reactants} \rightarrow \{r\} \text{products}$$

where r is a rate constant. Parts of a reaction may optionally be omitted as in e.g. the dimerisation reaction $\text{luxR} + \text{m3OC6HSL} \rightarrow \{0.5\} \text{luxR-m3OC6HSL}$ in which there is no enzyme, or as in e.g. $\text{luxI} \sim \rightarrow \{1.0\} \text{m3OC6HSL}$ in which m3OC6HSL is synthesised by luxI without any reactants specified. The last four lines of the reaction database represent transport reactions, where e.g. $\text{m3OC6HSL} \rightarrow \{0.5\} [\text{m3OC6HSL}]$ is the transport of m3OC6HSL into some compartment.

8.1.2 The Parts Database

The parts table contains three columns: the first represents part *types*, the second represents unique *IDs* (taken from the MIT Registry when possible), and the third represents sets of *properties*. For the purpose of our examples, the available types are restricted to promoters **prom**, ribosome binding sites **rbs**, protein coding regions **per** and terminators **ter**. Table 8.1.2 shows our graphical representations of the four part types, where *id* ranges over part identifiers in the parts database.

Promoters can have the properties **pos(p,rb,rub,rtb)** and **neg(p,rb,rub,rtb)**, where *p* is a transcription factor (a protein or protein complex) resulting in positive or negative regulation, respectively. The remaining entries give a quantitative characterisation of promoter regulation: *rb* and *rub* are the binding and unbinding rates of the transcription factor to the promoter, and *rtb* is the rate of transcription in the bound state. Promoters can also have the property **con(rt)** where *rt* is the constitutive rate of transcription in the absence of transcription factors. Protein coding regions have the single property **codes(p,rd)** indicating the protein *p* they code for, together with a rate *rd* of protein

Table 8.1.1: Table representation of a prototype parts database. The three columns describe the type, identifier and properties associated with each part.

Type	ID	Properties
pcr	c0051	codes (clR, 0.001)
pcr	c0040	codes (tetR, 0.001)
pcr	c0080	codes (araC, 0.001)
pcr	c0012	codes (lacI, 0.001)
pcr	c0061	codes (luxI, 0.001)
pcr	c0062	codes (luxR, 0.001)
pcr	c0079	codes (lasR, 0.001)
pcr	c0078	codes (lasI, 0.001)
pcr	cunknown3	codes (ccdB, 0.005)
pcr	cunknown4	codes (ccdA, 0.1)
pcr	cunknown5	codes (ccdA2, 10.0)
prom	r0051	neg (clR, 1.0, 0.5, 0.00005) con (0.12)
prom	r0040	neg (tetR, 1.0, 0.5, 0.00005) con (0.09)
prom	i0500	neg (araC, 1.0, 0.000001, 0.0001) con (0.1)
prom	r0011	neg (lacI, 1.0, 0.5, 0.00005) con (0.1)
prom	runknown2	pos (lasR-m3OC12HSL, 1.0, 0.8, 0.1) pos (luxR-m3OC6HSL, 1.0, 0.8, 0.1) con (0.000001)
rbs	b0034	rate (0.1)
ter	b0015	

Table 8.1.2: Part types in GEC with their corresponding graphical representation.

Part	Representation
id : prom	
id : rbs	
id : pcr	
id : ter	

degradation. Ribosome binding sites may have the single property **rate(r)**, representing a rate of translation of mRNA.

A part may generally have any number of properties, e.g. indicating regulation of a promoter by different transcription factors. However, we stress that the GEC language is to a large extent independent of any particular choice of part types and properties; the exception is the translation to reactions, which relies on the part types and properties described above.

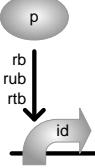
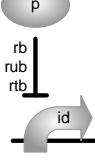
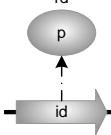
Sometimes we may wish to ignore quantitative information, in which case we assume derived, non-quantitative versions of the properties: for all properties **pos(p,rb,rub,rt)** and **neg(p,rb,rub,rt)** there are derived properties **pos(p)** and **neg(p)**, and for every property **codes(p, rd)** there is a derived property **codes(p)**.

8.1.3 Reactions Associated with Parts

While the reaction database explicitly represents a set of known reactions, the rate information associated with part properties allows further reactions at the level of gene expression to be deduced. Table 8.1.3 shows our graphical representations of part properties together with their resulting reactions. A dotted arrow is used to represent protein production, and arrows for positive and negative regulation are inspired by standard notations.

The **pos** and **neg** properties of promoters each give rise to three reactions: binding and unbinding of the transcription factor and production of mRNA in the bound state. The **con** property of a promoter yields a reaction producing mRNA in the unbound state, while the **rate** property of a ribosome binding site yields a reaction producing

Table 8.1.3: Part properties and their reactions in GEC, with their corresponding graphical representation. The species g represents a gene, p represents a protein and m represents an mRNA. The specific choice of species used in the reactions associated with a part sometimes depends on neighbouring parts.

Part property and reactions	Representation
$\text{id : prom <pos(p,rb,rub,rtb)>}$ $g + p \rightarrow \{rb\} g - p$ $g - p \rightarrow \{rub\} g + p$ $g - p \rightarrow \{rtb\} g - p + m$	
$\text{id : prom <neg(p,rb,rub,rtb)>}$ $g + p \rightarrow \{rb\} g - p$ $g - p \rightarrow \{rub\} g + p$ $g - p \rightarrow \{rtb\} g - p + m$	
$\text{id : prom <con(rt)>}$ $g \rightarrow \{rt\} g + m$ $m \rightarrow \{rdm\}$	
$\text{id : rbs <rate(r)>}$ $m \rightarrow \{r\} m + p$	
$\text{id : pcr <codes(p,rd)>}$ $p \rightarrow \{rd\}$	

protein from mRNA. Finally, the `codes` property of a protein coding region gives rise to a protein degradation reaction. We observe that mRNA degradation rates do not associate naturally with any of the part types since mRNA may be polycistronic (i.e., code for multiple proteins). Therefore, the rate `rdm` used for mRNA degradation is assumed to be defined globally, but may be adjusted manually for individual cases where appropriate. Note also that quantities such as protein degradation rates could in principle be stored in the reaction database as degradation reactions. We choose however to keep as much quantitative information as possible about a given part within the parts database.

8.2 The Basics of GEC

8.2.1 Sequences of Typed Parts

On the most basic level, a program can simply be a sequence of part identifiers together with their types, essentially corresponding to a program in the GenoCad language introduced in Section 2.4. The following program is an example of a transcription unit which expresses the protein `tetR` in a negative feedback loop; a corresponding graphical representation is shown above the program code:



```
r0040 : prom ; b0034 : rbs ; c0040 : pcr ; b0015 : ter
```

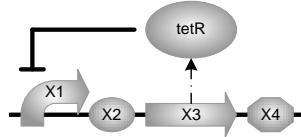
The symbol `:` is used to write the type of a part, and the symbol `;` is the *sequential composition* operator used to put parts together in sequence. Writing this simple program requires the programmer to know that the protein coding region part `c0040` codes for the protein `tetR`, and that the promoter part `r0040` is negatively regulated by this protein. We can confirm these two facts by inspecting Table 8.1.1. The translation simply results in a single list consisting of the given sequence of part identifiers, while ignoring the types:

```
[r0040; b0034; b0040; b0015]
```

8.2.2 Part Variables and Properties

We can increase the level of abstraction of the program by using *variables* and *properties* for expressing that any parts will do, as long as the protein coding region codes

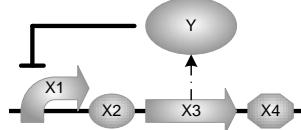
for the protein tetR and the promoter is negatively regulated by tetR:



```
X1 : prom<neg( tetR )>; X2 : rbs ; X3 : pcr<codes( tetR )>; X4 : ter
```

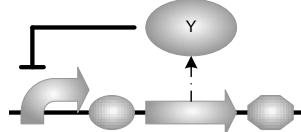
The angled brackets $<>$ delimit one or more properties, and upper-case identifiers such as X1 are variables which represent undetermined part identifiers or species. The translation of this program gives exactly the same result as before, but without the programmer having to find the specific parts required; these are deduced from the parts database.

The translation may in general produce several results. For example, we can replace the fixed species name tetR with a new variable, thus resulting in a program expressing *any* transcription unit behaving as a negative feedback device:



```
X1 : prom<neg( Y )>; X2 : rbs ; X3 : pcr<codes( Y )>; X4 : ter
```

This time the translation produces 4 devices, one of them being the tetR device from above. When variables are only used once, as is the case for X1, X2, X3 and X4 above, their names are of no significance and we will use the *wild card*, $_$, instead. When there is no risk of ambiguity, we may omit the wild card altogether and write the above program more concisely as follows:



```
prom<neg( Y )>; rbs ; pcr<codes( Y )>; ter
```

8.2.3 Parameterised Modules

Parameterised modules allow further abstraction away from the level of individual parts. Modules which act as positive or negative gates, or which constitutively express a protein, can be written as shown in Listing 8.2.1, where i denotes *input* and o denotes *output*.

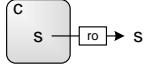
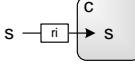
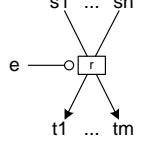
Listing 8.2.1: Gate module definitions.

```

1 module t1(o) { rbs; pcr<codes(o)>; ter };
2 module gatePos(i, o) { prom<pos(i)>; t1(o) };
3 module gateNeg(i, o) { prom<neg(i)>; t1(o) };
4 module gateCon(o) { r0051:prom; t1(o) };
5
6 module t12(o1, o2) { rbs; pcr<codes(o1)>; rbs; pcr<codes(o2)>; ter };
7 module gateCon2(o1, o2) { r0051:prom; t12(o1, o2) };

```

Table 8.2.1: Reactions in GEC together with their corresponding graphical representation.

Reaction	Representation
$c[s] \rightarrow \{ro\} s$	
$s \rightarrow \{ri\} c[s]$	
$e \sim s_1 + \dots + s_n \rightarrow \{r\} t_1 + \dots + t_m$	

The **module** keyword is followed by the name of the module, a list of formal parameters and the body of the module enclosed in curly brackets. For the constitutive expression module, we arbitrarily fix a promoter. Modules can be invoked simply by naming them together with a list of actual parameters, as in the case of the “tail” module, *t1*. The last module, *gateCon2*, is a dual-output version of the constitutive expression module *gateCon*; similar dual-output versions of the positive and negative gates can be defined, but they are not needed for our examples. Note that the dual-output module gives rise to devices which express polycistronic mRNA.

8.2.4 Compartments and Reactions

GEC has compartments which, as in LBS, enable a distinction between different instances of the same parts or proteins in different cells; however, compartments also

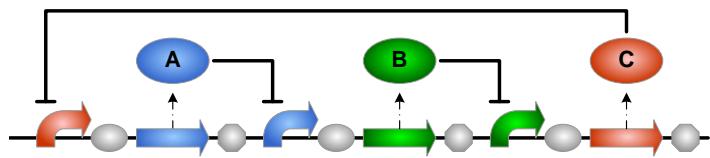


Figure 8.3.1: A diagrammatic representation of the repressilator circuit.

play an important role when resolving constraints as we demonstrate below. GEC furthermore has reactions, including cross-compartment transport, which can be used to impose additional constraints on parts, also demonstrated below. Table 8.2.1 shows the general form of reactions together with their graphical representation.

8.3 Case Study: The Repressilator

8.3.1 The GEC Model

Our first case study considers the repressilator circuit [33] which consists of three genes negatively regulating each other as shown in Figure 8.3.1. The first gene in the circuit expresses some protein A which represses the second gene; the second gene expresses some protein B which represses the third gene; and the third gene expresses some protein C which represses the first gene, thus closing the feedback loop. Using our standard gate modules, the repressor can be written in GEC as follows:

```
gateNeg(C, A); gateNeg(A, B); gateNeg(B, C)
```

8.3.2 Translation and Simulation

The translation of the repressilator program results in 24 possible devices. One of these is the following:

```
[r0051, b0034, c0040, b0015, r0040, b0034,
c0080, b0015, i0500, b0034, c0051, b0015]
```

To see why 24 devices have been generated, an inspection of the databases reveals that there are four promoter/repressor pairs that can be used in the translation of the program: r0011/c0012, r0040/c0040, r0051/c0051 and i0500/c0080. It follows that there are 4 choices for the first promoter in the target device, 3 choices for the second promoter, and 2 remaining choices for the third promoter. There is only one ribosome binding

site and one terminator registered in the parts database, and hence there are indeed $4 \cdot 3 \cdot 2 = 24$ possible target devices.

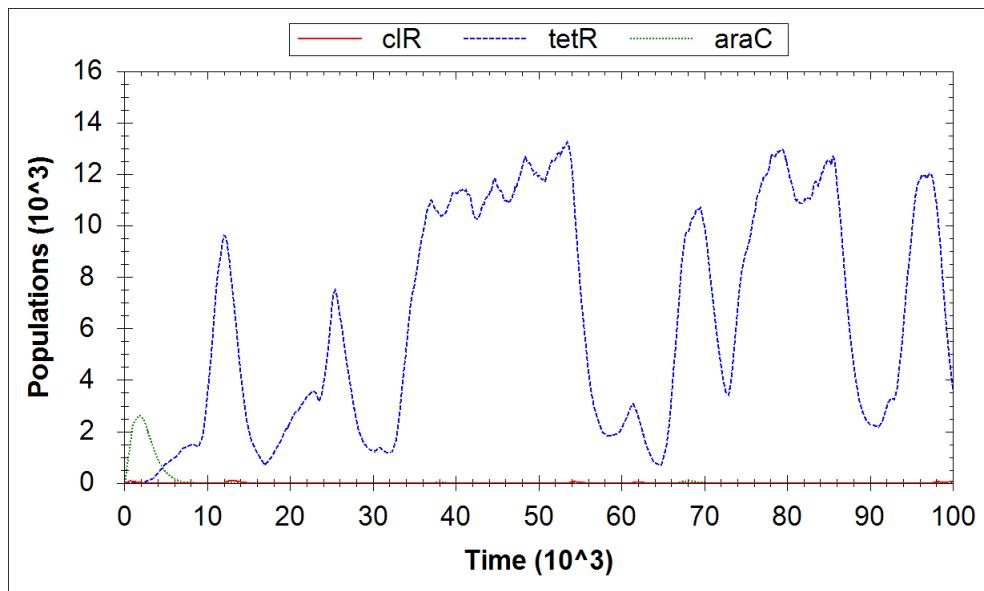
Our above reasoning reflects an important assumption about the semantics of the language: *distinct variables must take distinct values*. If we allowed e.g. A and B to represent the same protein, we would get self-inhibiting gates in some devices which would certainly prevent the desired behaviour. This assumption seems to be justified in most cases, although it is easy to change the semantics of GEC on a per-application basis in order to allow variables to take identical values. We also note that variables range over atomic species rather than complexes, so any promoters which are regulated by dimers, for example, would not be chosen by the translation.

In the face of multiple possible devices, the question of which device to choose naturally arises. This is where simulation and model refinement become relevant. Figure 8.3.2a shows the result of simulating the reactions associated with the device listed above. We observe that the expected oscillations are not obtained. By further inspection, we discover the failure to be caused by a very low rate of transcription factor unbinding for the promoter i0500: once a transcription factor (araC) is bound, the promoter is likely to remain repressed.

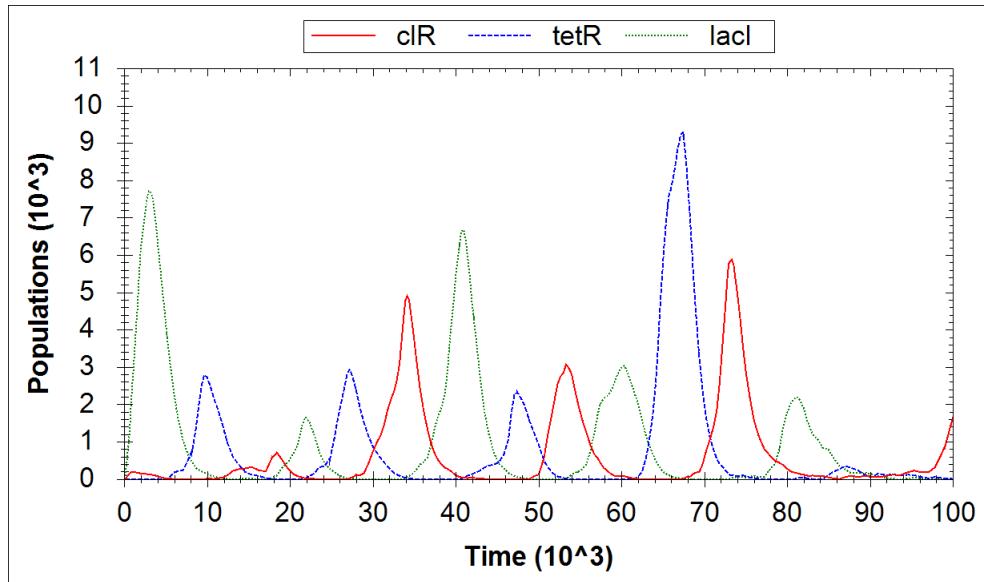
8.3.3 The Revised GEC Model

Appropriate ranges for quantitative parameters in which oscillations do occur can be found through further simulations or parameter scans as in [9]. We can then refine the repressilator program by imposing these ranges as quantitative constraints. This can be done by redefining the negative gate module as shown in Listing 8.3.1, leaving the body of the repressilator program unmodified. This ability to make localised changes to a program is one important benefit of modularity.

The first two lines use the *new operator* to ensure that variables are globally fresh. This means that variables of the same name, but under different scopes of the new operator, are considered semantically distinct. This is important in the repressilator example because the gateNeg module is instantiated three times, and we do not require that e.g. the binding rate RB is the same for all three instances. A sequence of part types with properties then follows, this time with rates given by variables. Finally, constraints on these rate variables are composed using the *constraint composition operator*, $|$. With this module replacing the non-quantitative gate module defined previously, the translation of the repressilator program now results in the 6 devices without



(a) A defective repressor device.



(b) A working repressor device.

Figure 8.3.2: Stochastic simulation plots of two repressor devices. The device using clR, tetR and araC (a) is defective due to the low rate of unbinding between araC and its promoter, while the device using clR, tetR and lacI (b) behaves as expected.

Listing 8.3.1: A revised repressilator program with quantitative constraints

```

1 module gateNeg(i, o) {
2   new RB. new RUB. new RTB.
3   new RT. new R. new RD.
4
5   prom<con(RT), neg(i, RB, RUB, RTB)>;
6   rbs<rate(R)>;
7   pcr<codes(o, RD)>; ter |
8
9   0.9 < RB | RB < 1.1 |
10  0.4 < RUB | RUB < 0.6 |
11  0.05 < RT | RT < 0.15 |
12  RTB < 0.01 |
13  0.05 < R | R < 0.15
14 };

```

the promoter regulated by araC, rather than the 24 devices from before. One of these is the repressilator device contained in the MIT Registry under the identifier I5610:

```
[r0040, b0034, c0051, b0015, r0051, b0034,
c0012, b0015, r0011, b0034, c0040, b0015]
```

Simulation of the associated reactions now yields the expected oscillations as shown in Figure 8.3.2b.

8.4 Case Study: The Predator-Prey System

8.4.1 The GEC Model

Our second case study, an *Escherichia coli* predator-prey system [5] shown in Figure 8.4.1, represents one of the largest synthetic systems implemented to date. It is based on two quorum sensing systems, one enabling predator cells to *induce* expression of a death gene in the prey cells, and another enabling prey cells to *inhibit* expression of a death gene in the predator. In the predator, Q1a is constitutively expressed and synthesises H1, which diffuses to the prey where it dimerises with the constitutively expressed Q1b. This dimer in turn induces expression of the death protein ccdB. Symmetrically, the prey constitutively expresses Q2a for synthesising H2, which diffuses to

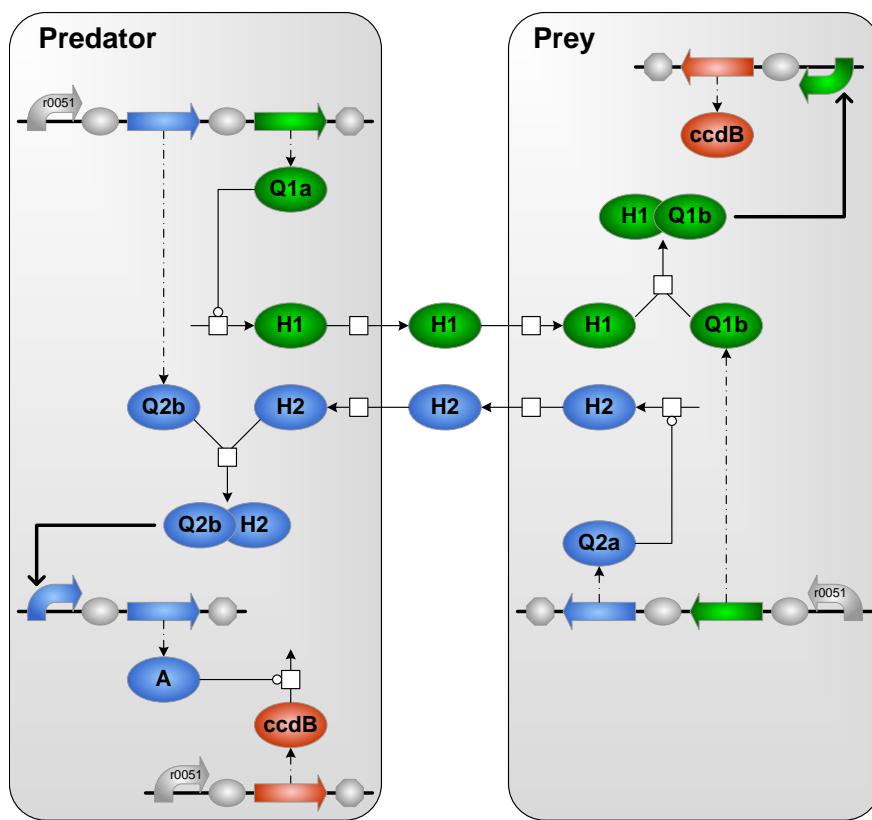


Figure 8.4.1: A diagrammatic representation of the predator-prey system.

the predator where it dimerises with the constitutively expressed Q2b. Instead of inducing cell death, this dimer induces expression of an *antidote* A, which interferes with the constitutively expressed death protein.

A GEC program implementing the logic of Figure 8.4.1 is shown in Listing 8.4.1. Note how the details of the quorum sensing system and antidote have been left unspecified by using variables (upper-case identifiers) for the species involved. Only the death protein is specified explicitly (using a lower-case identifier).

The predator and prey are programmed in two separate modules which reflect our informal description of the system, and a third module links the predator and prey by defining transport reactions. Several additional language constructs are demonstrated in this program. *Reactions* are composed with each other and with the standard gate modules through the constraint composition operator which is also used for quantitative constraints. Reactions have no effect on the “layout” of the resulting devices since they do not add any parts to the system, but they restrict the possible choices of proteins and hence of parts.

The last two lines of the predator and prey modules specify reactions which are

Listing 8.4.1: A GEC model of the predator-prey system.

```

1  module predator() {
2    gateCon2(Q2b, Q1a) |
3    Q1a ~ -> H1;
4
5    Q2b + H2 <-> Q2b-H2 |
6    gatePos(Q2b-H2, A);
7    gateCon(ccdB) |
8    A ~ ccdB -> |
9
10   ccdB ~ Q1a *->{10.0} |
11   H1 *->{10.0} | H2 *->{10.0}
12 };
13
14 module prey() {
15   gatePos(H1-Q1b, ccdB) |
16   H1 + Q1b <-> H1-Q1b ;
17
18   Q2a ~ -> H2 |
19   gateCon2(Q2a, Q1b) |
20
21   ccdB ~ Q2a *->{10.0} |
22   H1 *->{10.0} | H2 *->{10.0}
23 };
24
25 module transport() {
26   c1[H1] -> H1 | H1 -> c2[H1] |
27   c2[H2] -> H2 | H2 -> c1[H2]
28 };
29
30 c1[predator()] || c2[prey()] || transport()

```

used for *simulation only* and do not impose any constraints, indicated by the star preceding the reaction arrows. The second to last line of each module is a simple approach to modelling cell death, and we return to this when discussing simulations shortly. The last line consists of degradation reactions for H1 and H2; since these are not the result of gene expression (they are chemicals, not proteins), the associated degradation reactions are not deduced automatically by the translation.

The transport module defines *transport reactions* in and out of two compartments, c1 and c2, representing respectively the predator and prey cell boundaries. The choice of compartment names is not important for the translation to devices, but it is important for the translation to reactions where a distinction must be made between the populations of the same species in different compartments.

The main body of the program invokes the three modules while putting the predator and prey inside their respective compartments using the *compartment operator*. The modules are composed using the *parallel composition operator*. In contrast to the sequential composition operator which intuitively concatenates the part sequences of its operands, the parallel composition intuitively results in the union of the part sequences of its operands. This is useful when devices are implemented on different plasmids, or even in different cells as in this example.

8.4.2 Translation and Simulation

The translation results in four devices, each consisting of two lists of parts that implement the predator and prey, respectively. One of the devices is shown below.

```
[r0051, b0034, c0062, b0034, c0078, b0015, runknown2, b0034,
cunknow5, b0015, r0051, b0034, cunknow3, b0015]
```

```
[runknown2, b0034, cunknow3, b0015, r0051, b0034, c0061,
b0034, c0079, b0015]
```

By inspection of the database we establish that the translation has selected luxR/lasI/m3OC12HSL and lasR/luxI/m3OC6HSL for implementing the quorum sensing components in the respective cells, and ccdA2 for the antidote. We also see that it has found a unique promoter, runknown2, which is used *both* for regulating expression of ccdA2 in the predator *and* for regulating expression of ccdB in the prey. The fact that the two instances of this one promoter are located in different compartments now plays a crucial role: without the compartment boundaries, undesired crosstalk would

arise between lasR–m3OC12HSL and the promoter runknown2 in the predator, and between luxR–m3OC6HSL and the promoter runknown2 in the prey. Indeed, if we remove the compartments from the program, this crosstalk will be detected during translation, resulting in the empty set of devices.

This illustrates another important assumption about the semantics of GEC: *a part may be used only if its “implicit” properties do not contain species which are present in the same compartment as the part.* By “implicit” properties we mean the properties of a part which are not explicitly specified in the program. In our example, the part runknown2 in the predator has the implicit property that it is positively regulated by lasR–m3OC12HSL. Hence the part may not be used in a compartment where lasR–m3OC12HSL is present. The use of compartments in our example ensures that this condition of crosstalk avoidance is met.

Simulation results, in which the populations of the killer protein ccdB in the predator and prey are plotted, are shown in Figure 8.4.2a. We observe that the killer protein in the predator remains expressed, hence blocking the synthesis of H1 through the simulation-only reaction, and preventing expression of the killer protein in the prey. This constant level of killer protein in the predator is explained by the low rate at which the antidote protein ccdA2 used in this particular device degrades ccdB, and by the high degradation rate of ccdA2. The second of the four devices is identical to the device above, except that the more effective antidote ccdA is expressed using cunknown4 instead of cunknown5:

```
[r0051, b0034, c0062, b0034, c0078, b0015, runknown2, b0034,
```

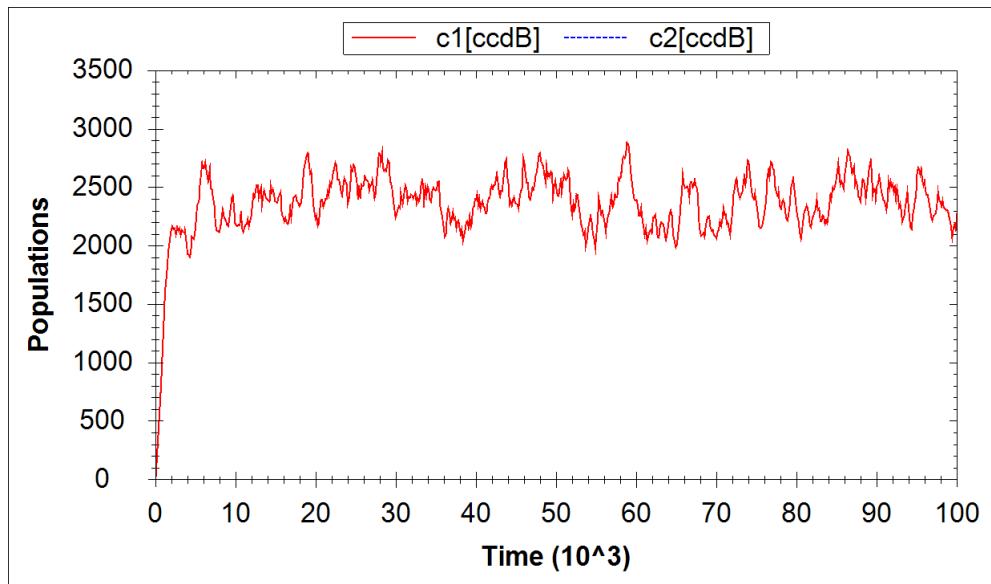
```
cunknown4, b0015, r0051, b0034, cunknown3, b0015]
```

```
[runknown2, b0034, cunknown3, b0015, r0051, b0034, c0061,
```

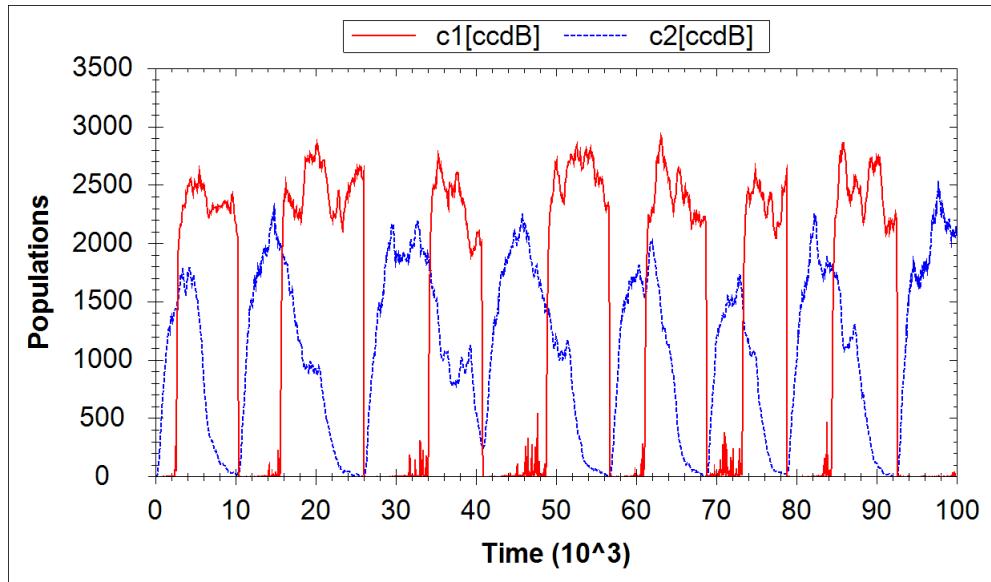
```
b0034, c0079, b0015]
```

The simulation results of the reactions associated with this device are shown in Figure 8.4.2b. The two remaining devices are symmetric to the ones shown above in the sense that the same two quorum sensing systems are used, but they are swapped around such that the predator produces m3OC6HSL rather than m3OC12HSL, and vice-versa for the prey.

We stress that the simulation results obtained for the predator-prey system do not reproduce the published results. One reason is that we are plotting the levels of killer proteins in a single predator and a single prey cell rather than cell populations, in order



(a) A defective predator-prey device.



(b) A working predator-prey device.

Figure 8.4.2: Stochastic simulation plots of two predator-prey devices. The device using ccdA2 (a) is defective due to the low rate of ccdA2-catalysed ccdB degradation, while the device using ccdA (b) behaves as expected.

to simplify the simulations. Another reason is that the published results are based on a reduced ODE model, and the parameters for the full model are not readily available. Our simplified model is nevertheless sufficient to illustrate the approach, and can be further refined to include additional details of the experimental setup.

We note a number of additional simplifying omissions in our model: expression of the quorum-sensing proteins in the prey, and of the killer protein in the predator, are IPTG-induced in the original model; activated luxR (i.e. in complex with m3OC6HSL) dimerises before acting as a transcription factor; and the antidote mechanism is more complicated than mere degradation [1].

Chapter 9

The Abstract Syntax and Semantics of GEC

In the previous chapter we discussed informally how GEC programs can be translated to devices and to the associated reactions which enable simulations to be carried out. Technically, the translation is achieved through three separate denotation functions. The first is concerned with resolving the constraints of GEC programs, and its target semantical objects are sets of *context-sensitive substitutions* which, in addition to other necessary structures as detailed below, contain mappings from variables to part identifiers, species identifiers and real numbers. The second denotation function is concerned with the “layout” of devices, and its target semantical objects are *device templates*, which are just devices that may contain variables. The third denotation function pertains to reactions, and its target semantical objects are *reaction program templates*; these are simplified CBS or LBS programs which may contain variables.

Each of the substitutions resulting from the first denotation function can be applied to the device template and the reaction program template arising from the two other denotation functions in order to obtain each of the possible devices and their associated reactions.

We start in Section 1 with the abstract syntax of GEC and then consider each of the three denotation functions in Sections 2-4. The denotation functions for context-sensitive substitutions and for device templates are independent of any particular choice of part types and properties, but the denotation function for reaction program templates does assume the specific part types and properties introduced in the previous chapter.

9.1 The Abstract Syntax of GEC

We assume a given set ID_s of *species identifiers*, a given set ID_p of *part identifiers*, ranged over by id_p , and a given set X of *variables*, ranged over by x ; we assume for technical reasons that the set X includes the set $\{0,1\}^*$ of binary strings. We let $S \stackrel{\Delta}{=} MS(ID_s \cup X)$ be the set of *complex species with variables*, ranged over by S .

We furthermore let $A \stackrel{\Delta}{=} \mathbb{R} \cup X \cup S$ be the set of *actual parameters*, ranged over by a , and we let $U \stackrel{\Delta}{=} ID_s \cup ID_p \cup X$ be the set of *formal parameters*, ranged over by u . A type system is needed to ensure the proper use of formal parameters and their bindings to actual parameters, but we omit that aspect from the presentation for the sake of simplicity.

A given set T of *part types*, ranged over by t , is also assumed, together with a set Q^t of possible part *properties* for each type $t \in T$. We define $Q \stackrel{\Delta}{=} \bigcup_{t \in T} Q^t$ and let $Q^t \subsetneq_{\text{fin}} Q^t$. In the case studies given in the previous chapter, and in the semantics in terms of reaction template programs given in Section 4, properties are appropriate terms over $\mathbb{R} \cup X \cup S$, but otherwise the specific structure of properties is not important; we just assume given functions $FV : Q \rightarrow X$ and $FS : Q \rightarrow S$ for obtaining the variables and species of properties, respectively, and we assume these functions extended to other structures as appropriate.

We let id_c range over a given set of compartment identifiers, we let id_m range over a given set of module identifiers and we let \otimes range over a given set of arithmetic operators which could e.g. be the usual operators as for algebraic rate expressions in LBS. Finally, we let $n \in \mathbb{N}$, $r \in \mathbb{R}$ and $v_b \in \{\text{tt}, \text{ff}\}$ as before; the boolean value v_b is used to indicate whether reactions and transport reactions should be used for simulation only. The abstract syntax of GEC is then given by the grammar in Table 9.1.1.

Derived forms Some of the language constructs used in the previous chapter are not represented explicitly in the grammar of Table 9.1.1 but can be defined in terms of those which are. Reversible reactions are defined as the constraint composition of two reactions, each representing one of the directions, as in LBS. The underscore wildcard, $_$, can be defined using variables and the **new** operator; for example:

$$_ : t(Q') \stackrel{\Delta}{=} \mathbf{new} \ x. x : t(Q')$$

In the specific case of basic part programs the wild card can be omitted, i.e. $t(Q') \stackrel{\Delta}{=} _ : t(Q')$. We also allow constraints to be composed to the left of programs and define $C \mid P \stackrel{\Delta}{=} P \mid C$.

Table 9.1.1: The abstract syntax of GEC.

$P ::=$	GEC PROGRAM
$u : t(Q^t)$	TYPED PART WITH PROPERTIES
$\mathbf{0}$	NIL PROGRAM
$id_m(\underline{u}) = P_1 ; P_2$	MODULE DEFINITION
$id_m(\underline{a})$	MODULE INVOCATION
$P C$	CONSTRAINT COMPOSITION
$P_1 \parallel P_2$	PARALLEL COMPOSITION
$P_1 ; P_2$	SEQUENTIAL COMPOSITION
$id_c[P]$	LOCATED PROGRAM
$\mathbf{new} \ x. \ P$	NEW VARIABLE
$C ::=$	CONSTRAINT
R^{v_b}	REACTION
T^{v_b}	TRANSPORT REACTION
K	NUMERICAL CONSTRAINT
$R ::= S \sim \sum n_i \cdot S_i \rightarrow^r \sum n'_j \cdot S'_j$	REACTION
$T ::=$	TRANSPORT
$S \rightarrow^r id_c[S]$	TRANSPORT INTO COMPARTMENT
$id_c[S] \rightarrow^r S$	TRANSPORT OUT OF COMPARTMENT
$K ::= E_1 > E_2$	NUMERICAL CONSTRAINT
$E ::=$	ARITHMETIC EXPRESSION
r	REAL NUMBER
x	VARIABLE
$E_1 \otimes E_2$	ARITHMETIC OPERATOR

Abstract vs. Concrete Syntax The example GEC programs given in the previous chapter are written using a concrete syntax that can be understood by the implemented parser. The main difference, compared to the above abstract syntax, is that variables are represented by upper case identifiers whereas species and part names are represented by lower case identifiers. Complex species are composed using the hyphen, `-`, in the concrete syntax, and the fact that complex species are multisets in the abstract syntax reflects that complex formation is commutative. Similar considerations apply to the sum operator in reactions. We also assume some standard precedence rules, in particular that sequential composition (`;`) binds tighter than parallel composition (`||`), and we allow the use of parentheses to override these standard rules if necessary.

9.2 The Substitution Semantics of GEC

We first illustrate the substitution semantics of GEC informally through a small example which exhibits characteristics from the predator-prey case study, and then turn to the formal presentation.

9.2.1 The Intuition

The substitution semantics is given in terms of *context-sensitive substitutions* $(\theta, \rho, \sigma, \tau)$ which represent solutions to the constraints of a program; informally, θ is a mapping from variables to species identifiers, part identifiers or real numbers; ρ is a set of variables for which θ must be injective; σ and τ are, respectively, the species that have been used in the current context and the species that are excluded for use. Context-sensitive substitutions capture the information needed to ensure both piece-wise injectivity over compartment boundaries and cross-talk avoidance, as mentioned in the case studies.

Consider the following example:

$(X1 : \text{prom} < \text{pos} (H1 - Q1b) > ; X2 : \text{rbs}) \parallel (Y1 : \text{prom} < \text{pos} (Q2b - H2) > ; Y2 : \text{rbs})$

The translation first processes the two sequential compositions in isolation, and then the parallel composition.

1. **The first sequential composition.** Observe that the database only lists a single ribosome binding site part and a single promoter part that is positively regulated by a dimer, namely `runknown2`. The first sequential composition gives rise

to two context-sensitive substitutions, one for each possible choice of transcription factors listed in the database for this promoter part. The context-sensitive substitutions are $(\theta_1, \rho_1, \sigma_1, \tau_1)$ and $(\theta'_1, \rho'_1, \sigma'_1, \tau'_1)$ where

- $\theta_1 \stackrel{\Delta}{\simeq} \{(X1 \mapsto \text{runknown2}), (X2 \mapsto b0034), (H1 \mapsto \text{m30C12HSL}), (Q1b \mapsto \text{lasR})\}$
- $\rho_1 \stackrel{\Delta}{\simeq} \{H1, Q1b\}$
- $\sigma_1 \stackrel{\Delta}{\simeq} \{\text{m30C12HSL-lasR}\}$
- $\tau_1 \stackrel{\Delta}{\simeq} \{\text{m30C6HSL-luxR}\}$
- $\theta'_1 \stackrel{\Delta}{\simeq} \{(X1 \mapsto \text{runknown2}), (X2 \mapsto b0034), (H1 \mapsto \text{m30C6HSL}), (Q1b \mapsto \text{luxR})\}$
- $\rho'_1 \stackrel{\Delta}{\simeq} \{H1, Q1b\}$
- $\sigma'_1 \stackrel{\Delta}{\simeq} \{\text{m30C6HSL-luxR}\}$
- $\tau'_1 \stackrel{\Delta}{\simeq} \{\text{m30C12HSL-lasR}\}$

Note that $\tau_1 \stackrel{\Delta}{\simeq} \{\text{m30C6HSL-luxR}\}$, since the complex m30C6HSL-luxR is in the properties of runknown2 but has not been mentioned explicitly in the program under the corresponding substitution θ_1 . Therefore, this complex should not be used anywhere in the same compartment as runknown2 , in order to prevent unwanted interference between parts. Similar ideas apply to τ'_1 .

2. The second sequential composition. The second sequential composition produces similar results, namely two context-sensitive substitutions, one for each possible choice of transcription factors in the database for the promoter part. The context-sensitive substitutions are $(\theta_2, \rho_2, \sigma_2, \tau_2)$ and $(\theta'_2, \rho'_2, \sigma'_2, \tau'_2)$ where

- $\theta_2 \stackrel{\Delta}{\simeq} \{(Y1 \mapsto \text{runknown2}), (Y2 \mapsto b0034), (H2 \mapsto \text{m30C12HSL}), (Q2b \mapsto \text{lasR})\}$
- $\rho_2 \stackrel{\Delta}{\simeq} \{H2, Q2b\}$
- $\sigma_2 \stackrel{\Delta}{\simeq} \{\text{m30C12HSL-lasR}\}$
- $\tau_2 \stackrel{\Delta}{\simeq} \{\text{m30C6HSL-luxR}\}$
- $\theta'_2 \stackrel{\Delta}{\simeq} \{(Y1 \mapsto \text{runknown2}), (Y2 \mapsto b0034), (H2 \mapsto \text{m30C6HSL}), (Q2b \mapsto \text{luxR})\}$
- $\rho'_2 \stackrel{\Delta}{\simeq} \{H2, Q2b\}$
- $\sigma'_2 \stackrel{\Delta}{\simeq} \{\text{m30C6HSL-luxR}\}$
- $\tau'_2 \stackrel{\Delta}{\simeq} \{\text{m30C12HSL-lasR}\}$

3. **The parallel composition.** For the parallel composition we observe that none of the context-sensitive substitutions are “compatible”. We cannot combine θ_1 and θ_2 , nor θ'_1 and θ'_2 , because neither pair-wise union is injective on the corresponding domains determined by $\rho_1 \cup \rho_2$ and $\rho'_1 \cup \rho'_2$, respectively. And we cannot combine θ_1 and θ'_2 , nor θ'_1 and θ_2 , because the corresponding used species and excluded species overlap. Hence we are left with the empty set of context-sensitive substitutions.

This example demonstrates how solutions which give rise to cross-talk are filtered out. Suppose that we place the parallel components in separate compartments as in the predator-prey case study, e.g.:

```
c [X1 : prom<pos(H1-Q1b)> ; X2 : rbs] || d [Y1 : prom<pos(Q2b-H2)> ; Y2 : rbs]
```

The evaluation of this program then proceeds as follows:

1. **The first located sequential composition.** The first sequential composition gives rise to the same two context-sensitive substitutions as before, but after applying the compartment, the context-sensitive information is discarded. Hence we get the substitutions $(\theta_1, \rho_1, \sigma_1, \tau_1)$ and $(\theta'_1, \rho'_1, \sigma'_1, \tau'_1)$ where

- $\theta_1 \stackrel{\Delta}{\simeq} \{(X1 \mapsto \text{runknown2}), (X2 \mapsto b0034), (H1 \mapsto \text{m30C12HSL}), (Q1b \mapsto \text{lasR})\}$
- $\theta'_1 \stackrel{\Delta}{\simeq} \{(X1 \mapsto \text{runknown2}), (X2 \mapsto b0034), (H1 \mapsto \text{m30C6HSL}), (Q1b \mapsto \text{luxR})\}$

and the remaining sets are empty.

2. **The second located sequential composition.** Similarly, the second component of the parallel composition results in the two context-sensitive substitutions $(\theta_2, \rho_2, \sigma_2, \tau_2)$ and $(\theta'_2, \rho'_2, \sigma'_2, \tau'_2)$ where

- $\theta_2 \stackrel{\Delta}{\simeq} \{(Y1 \mapsto \text{runknown2}), (Y2 \mapsto b0034), (H2 \mapsto \text{m30C12HSL}), (Q2b \mapsto \text{lasR})\}$
- $\theta'_2 \stackrel{\Delta}{\simeq} \{(Y1 \mapsto \text{runknown2}), (Y2 \mapsto b0034), (H2 \mapsto \text{m30C6HSL}), (Q2b \mapsto \text{luxR})\}$

and the remaining sets are empty.

3. **The parallel composition.** All four combinations of context-sensitive substitutions from the two components are now compatible, and the parallel composition results in the set of unions of each combination. Hence we obtain the four context-sensitive substitutions $(\theta_1 \cup \theta_2, \emptyset, \emptyset, \emptyset)$, $(\theta_1 \cup \theta'_2, \emptyset, \emptyset, \emptyset)$, $(\theta'_1 \cup \theta_2, \emptyset, \emptyset, \emptyset)$ and $(\theta'_1 \cup \theta'_2, \emptyset, \emptyset, \emptyset)$.

9.2.2 The Definition

Transport reactions and databases Transport reactions contain explicit compartment identifiers which are important for simulation, but only the logical property that *transport is possible* is captured in the reaction database. We therefore define the operator $(\cdot)^\downarrow$ on transport reactions to disregard compartment identifiers, i.e.:

$$(S \rightarrow id_c[S])^\downarrow \stackrel{\Delta}{\simeq} S \rightarrow [S] \quad \text{and} \quad (id_c[S] \rightarrow S)^\downarrow \stackrel{\Delta}{\simeq} [S] \rightarrow S$$

The meaning of a program is then given relative to global databases \mathcal{K}_b and \mathcal{K}_r of parts and reactions, respectively. We assume these to be given by two finite sets of ground terms:

$$\begin{aligned} \mathcal{K}_b &\subsetneq \{id_p : t(Q^t) \mid FV(Q^t) = \emptyset\} \text{ and} \\ \mathcal{K}_r &\subsetneq \{R \mid FV(R) = \emptyset\} \cup \{T^\downarrow \mid FV(T) = \emptyset\} \end{aligned}$$

Context-sensitive substitutions We define CS to be the set of *context-sensitive substitutions* $(\theta, \rho, \sigma, \tau)$ where

1. $\theta : X \hookrightarrow_{\text{fin}} ID_s \cup ID_p \cup \mathbb{R}$ is a substitution.
2. $\rho \subsetneq_{\text{fin}} X$ is a set of variables over which θ is injective, i.e. $\forall x_1, x_2 \in \rho. (x_1 \neq x_2) \Rightarrow (\theta(x_1) \neq \theta(x_2))$.
3. $\sigma, \tau \subsetneq_{\text{fin}} S$ are, respectively, the species identifiers that have been used in the current context and the species identifiers that are excluded for use, and $\sigma \cap \tau = \emptyset$.

The target semantical objects of the denotation function are sets $\Theta \subsetneq CS$ of context-sensitive substitutions which represent solutions to constraints. They also capture the information necessary to ensure piece-wise injectivity over compartment boundaries, together with cross-talk avoidance, as mentioned in the case studies and in the above example.

We define the *composition* $\Theta_1 \otimes \Theta_2$ of two sets Θ_1 and Θ_2 of context-sensitive substitutions as follows:

$$\{(\theta_i, \rho_i, \sigma_i, \tau_i)\}_I \otimes \{(\theta'_j, \rho'_j, \sigma'_j, \tau'_j)\}_J \stackrel{\Delta}{\simeq} \{(\theta_i \cup \theta'_j, \rho_i \cup \rho'_j, \sigma_i \cup \sigma'_j, \tau_i \cup \tau'_j)\}_{I \times J} \cap CS$$

Informally, the composition consists of all possible pairwise unions of the operands which satisfy the conditions of context-sensitive substitutions. This means in particular

that the resulting substitutions are indeed functions, that they are injective over the relevant domain, and that the resulting sets of used and excluded species are disjoint. The latter implies that any combinations in which the used species in one component are not disjoint from the excluded species in another are filtered out by the composition.

If two sets of context-sensitive substitutions represent the solutions to the constraints of two programs, their composition represents the solutions to the constraints of the composite program (e.g. the parallel or sequential compositions). From now on we omit the indices I and J from indexed sets when they are understood from the context.

Module definitions Module definitions give rise to partial functions f of the form $f(\underline{a}, b) = \Theta$ mapping actual parameters \underline{a} and a binary string b to the target semantical object Θ of the module; b is used for generating fresh names as in the semantics of LBS. Module definitions are recorded in environments which are partial finite functions Γ of the form $\Gamma(id_m) = f$.

The substitution of actual parameters \underline{a} for formal parameters \underline{u} in a program P is written $P\{\underline{u}.i \mapsto \underline{a}.i\}$ and is defined inductively on programs along standard lines, with formal parameters and new variables as binders, except that a multiset interpretation is assumed in which nested multisets are flattened. For example, the substitution $\{s2 \mapsto s2a + s2b\}$ applied to the complex species $s1 + s2 + s2$ results in $s1 + 2 \cdot s2a + 2 \cdot s2b$. In the corresponding concrete syntax, the substitution $\{s2 \mapsto s2a - s2b\}$ applied to the complex species $s1 - s2 - s2$ is written $s1 - s2a - s2b - s2a - s2b$.

The denotation function We write $dom_s(\theta)$ for the subset of the domain of θ mapping to species identifiers, i.e.:

$$dom_s(\theta) \stackrel{\Delta}{=} \{x \in dom(\theta) \mid \theta(x) \in ID_s\}$$

A denotation function of the form $\llbracket K \rrbracket_{gs}\theta = v_b$ assigning a boolean value $v_b \in \{\text{tt}, \text{ff}\}$ to a numerical constraint K , given a function $\theta : X \hookrightarrow_{fin} \mathbb{R}$, can be defined along standard lines and we refrain from doing so here. The context-sensitive substitution denotational semantics of GEC is then given by a partial function of the form:

$$\llbracket P \rrbracket_{gs}\Gamma, b = \Theta$$

which, given an environment Γ and a binary string $b \in \{0, 1\}^*$, assigns a set Θ of context-sensitive substitutions to a GEC program; the binary string is used for assign-

ing fresh names to new variables following the same idea as for LBS. The definition is given below.

- $\llbracket u : t(Q^t) \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} \Theta$ where
 - $\Theta \stackrel{\Delta}{\simeq} \{(\theta_i, \rho_i, \sigma_i, FS(Q_i) \setminus \sigma_i) \mid u\theta_i : t(Q_i) \in \mathcal{K}_b \wedge Q^t\theta_i \subseteq Q_i \wedge \text{dom}(\theta_i) = FV(u : t(Q^t)) \wedge \rho_i = \text{dom}_s(\theta_i) \wedge \sigma_i = FS(Q^t\theta_i)\}$
- $\llbracket \mathbf{0} \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} \{(\emptyset, \emptyset, \emptyset, \emptyset)\}$
- $\llbracket id_m(\underline{u}) \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} P_1; P_2 \llbracket \Gamma, b \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{\text{gs}} \Gamma \langle id_m \mapsto f \rangle, b$ where
 - $f(\underline{a}, b') \stackrel{\Delta}{\simeq} \llbracket P_1 \{ \underline{u}.i \mapsto \underline{a}.i \} \rrbracket_{\text{gs}} \Gamma, b'$
- $\llbracket id_m(\underline{a}) \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} f(\underline{a}, b)$ where
 - $f \stackrel{\Delta}{\simeq} \Gamma(id_m)$
- $\llbracket P \mid C \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} \Theta_1 \otimes \Theta_2$ where
 - $\Theta_1 \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\text{gs}} \Gamma, b$
 - $\Theta_2 \stackrel{\Delta}{\simeq} \llbracket C \rrbracket_{\text{gs}}$
- $\llbracket P_1 \parallel P_2 \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} \Theta_1 \otimes \Theta_2$ where
 - $\Theta_1 \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{\text{gs}} \Gamma, 0b$
 - $\Theta_2 \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{\text{gs}} \Gamma, 1b$
- $\llbracket P_1 ; P_2 \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} \Theta_1 \otimes \Theta_2$ where
 - $\Theta_1 \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{\text{gs}} \Gamma, 0b$
 - $\Theta_2 \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{\text{gs}} \Gamma, 1b$
- $\llbracket id_c[P] \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} \{(\theta, \emptyset, \emptyset, \emptyset) \mid (\theta, \rho, \sigma, \tau) \in \Theta\}$ where
 - $\Theta \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\text{gs}} \Gamma, b$
- $\llbracket \mathbf{new} x. P \rrbracket_{\text{gs}} \Gamma, b \stackrel{\Delta}{\simeq} \llbracket P \{ x \mapsto b'0b \} \rrbracket_{\text{gs}} \Gamma, 1b$ where
 - b' is the shortest string in $\{b' \in \{0\}^* \mid b'0b \notin FV(P)\}$
- $\llbracket R^{\mathbf{tt}} \rrbracket_{\text{gs}} \stackrel{\Delta}{\simeq} \{(\emptyset, \emptyset, \emptyset, \emptyset)\}$

- $\llbracket R^{\text{ff}} \rrbracket_{\text{gs}} \stackrel{\Delta}{\simeq} \{(\theta_i, \text{dom}_s(\theta_i), \text{FS}(R\theta_i), \emptyset) \mid R\theta_i \in \mathcal{K}_r \wedge \text{dom}(\theta_i) = \text{FV}(R)\}$
- $\llbracket T \rrbracket_{\text{gs}} \stackrel{\Delta}{\simeq} \{(\theta_i, \text{dom}_s(\theta_i), \text{FS}(T\theta_i), \emptyset) \mid T^\downarrow \theta_i \in \mathcal{K}_r \wedge \text{dom}(\theta_i) = \text{FV}(T)\}$
- $\llbracket K \rrbracket_{\text{gs}} \stackrel{\Delta}{\simeq} \{(\theta_i, \emptyset, \emptyset, \emptyset) \mid \llbracket K \rrbracket_{\text{gs}} \theta_i = \mathbf{tt} \wedge \text{dom}(\theta_i) = \text{FV}(K)\}$

We furthermore define $\llbracket P \rrbracket_{\text{gs}} \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\text{gs}} \emptyset, \varepsilon$.

Explanation of the denotation function In the first case, the denotation function gives rise to substitutions satisfying the constraint that the part with the given properties, after the substitution has been applied, is in the parts database. The substitutions are furthermore required to be defined exactly on the variables mentioned in the program. The excluded species are those which are associated with the part in the database but not stated explicitly in the program.

The cases of module definition and invocation reflect the intuition given earlier. The cases of constraint composition, parallel composition and sequential composition all compose the substitutions arising from the components.

The case of compartments simply “forgets” about the injective domain, used species and excluded species. Hence subsequent compositions of the compartment program with other programs will not be restricted in the use of species, reflecting the intuition that cross-talk is not a concern across compartment boundaries, as illustrated in the predator-prey case study and in the above example.

In the case of the new variable operator, the specified variable is simply replaced by an appropriate fresh variable in the following program P . The fresh variable is constructed from the binary string parameter b of the denotation function by prefixing another string b' , chosen to ensure that the result is indeed fresh in P ; this is necessary because there may be free variables in P that are not generated through the semantics. Note that the construction must be based on b in order to ensure that the same new variable in different instances of a module get replaced by different fresh variables.

The cases of reactions and of transport follow a similar idea as the case of parts, except that the reaction database is used instead of the parts database. Finally, the case of numerical constraints gives rise to the set of all substitutions which satisfy the given constraints.

9.2.3 Results

Recall the two requirements for the translation mentioned informally in the case studies in the previous chapter. The first requirement is piece-wise injectivity of substitutions: *distinct species variables within the same compartment must take distinct values*. The second requirement is non-interference: *a part may be used only if its “implicit” properties do not contain species which are present in the same compartment as the part*. These requirements are formalised in the following two propositions. We use *contexts* $C(\cdot)$ to denote any program with zero or more holes, and $C(P)$ to denote the context with the (capture-free) substitution of P for the holes. The *free module identifiers* of P , defined in a standard manner with module definitions as binding constructs, are denoted by $FM(P)$. Finally, we say that a program P is *compartment-free* if it does not contain located programs, and we say that it is *well-formed* if all its module identifiers are defined exactly once and used at least once.

Proposition 9.2.1 (Piece-wise injectivity). *Let $C(\cdot)$ be any context with at least one hole and let P be any compartment-free, well-formed program with $FM(P) = \emptyset$. Let $\{(\theta_i, \rho_i, \sigma_i, \tau_i)\} \stackrel{\Delta}{=} \llbracket C(P) \rrbracket_{\text{gs}}$. Then θ_i is injective on the domain $FV(P) \cap \text{dom}_s(\theta_i)$.*

Proposition 9.2.2 (Non-interference). *Let $P = u : t(Q^t)$ be any basic program and let $C(\cdot)$ be any compartment-free, well-formed context with at least one hole. Let $\{(\theta_i, \rho_i, \sigma_i, \tau_i)\} \stackrel{\Delta}{=} \llbracket C(P) \rrbracket_{\text{gs}}$. Then $u\theta_i : t(Q) \in \mathcal{K}_b$ for some Q and $\sigma_i \cap (FS(Q) \setminus FS(Q'\theta_i)) = \emptyset$.*

9.3 The Device Semantics of GEC

9.3.1 The Intuition

A *device template* is a set of lists over part identifiers and variables, and this captures the relevant genetic structure of a program. A context-sensitive substitution can be applied to a device template in order to obtain a final concrete device, i.e. a set of lists over part identifiers. As an example, let us revisit the small program from the previous section:

```
c [X1:prom<pos(H1–Q1b)> ; X2:rbs] || d [Y1:prom<pos(Q2b–H2)> ; Y2:rbs]
```

The meaning of this program in terms of device templates is deduced informally as follows:

1. **The first sequential composition.** This gives rise to the device template $\{[X_1, X_2]\}$ obtained by concatenating the singleton lists in the two singleton sets $\{[X_1]\}$ and $\{[X_2]\}$; these in turn are obtained from the atomic programs in the sequential composition by preserving the part identifier while forgetting about the part type and properties.
2. **The second sequential composition.** This gives rise to the device template $\{[Y_1, Y_2]\}$ in a similar fashion.
3. **The parallel composition.** This results in the union $\{[X_1, X_2], [Y_1, Y_2]\}$ of the device templates from the two components.

Any of the four context-sensitive substitutions resulting from the substitution semantics can be applied to the resulting device template. In this particular example, all substitutions give rise to the same device, namely $\{[\text{runknown2}, \text{b0034}], [\text{runknown2}, \text{b0034}]\}$.

9.3.2 The Definition

We let Δ range over the set 2^{U^*} of device templates, i.e. sets of lists over variables and part identifiers, and we let δ range over the set U^* of single lists of variables and part identifiers. The denotational function assigns device templates to GEC programs and is of the form:

$$\llbracket P \rrbracket_{\text{gd}} \Gamma, b = \Delta$$

It is defined in the following.

- $\llbracket u : t(Q^t) \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \{[u]\}$
- $\llbracket \mathbf{0} \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \{\epsilon\}$
- $\llbracket id_m(\underline{u}) \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \llbracket P_1; P_2 \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{\text{gd}} \Gamma \langle id_m \mapsto f \rangle, b$ where
 - $f(\underline{a}, b') \stackrel{\Delta}{\simeq} \llbracket P_1 \{ \underline{u}.i \mapsto \underline{a}.i \} \rrbracket_{\text{gd}} \Gamma, b'$
- $\llbracket id_m(\underline{a}) \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} f(\underline{a}, b)$ where
 - $f \stackrel{\Delta}{\simeq} \Gamma(id_m)$
- $\llbracket P \mid C \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\text{gd}} \Gamma, b$
- $\llbracket P_1 \parallel P_2 \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \Delta_1 \cup \Delta_2$ where

- $\Delta_1 \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{\text{gd}} \Gamma, 0b$
- $\Delta_2 \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{\text{gd}} \Gamma, 1b$
- $\llbracket P_1 ; P_2 \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \{\delta_{1_i} \delta_{2_j}\}_{I \times J}$ where
 - $\{\delta_{1_i}\}_I \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{\text{gd}} \Gamma, 0b$
 - $\{\delta_{2_j}\}_J \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{\text{gd}} \Gamma, 1b$
- $\llbracket id_c[P] \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\text{gd}} \Gamma, b$
- $\llbracket \mathbf{new} x. P \rrbracket_{\text{gd}} \Gamma, b \stackrel{\Delta}{\simeq} \llbracket P \{x \mapsto b'0b\} \rrbracket_{\text{gd}} \Gamma, 1b$ where
 - b' is the shortest string in $\{b' \in \{0\}^* \mid b'0b \notin FV(P)\}$

We furthermore define $\llbracket P \rrbracket_{\text{gd}} \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\text{gd}} \emptyset, \varepsilon$.

Explanation of the denotation function The cases of module definition and invocation, and of new variables, are the same as the corresponding cases in the substitution semantics. In the case of a basic part program, the denotation function assigns a singleton set with a sequence consisting of the relevant part identifier or variable. The case of a nil program results in a singleton set with the empty list. The cases of constraint composition and of located programs simply evaluate the nested programs inductively, i.e. constraints and compartments have no effect on the structure of devices because they do not give rise to any parts.

The cases of parallel and sequential composition reflect the intuition given in our example. The parallel composition produces all the part sequences resulting from the first component together with all the part sequences resulting from the second component, i.e. the union of two sets, and the case of sequential composition gives rise to a Cartesian product.

9.4 The Reaction Semantics of GEC

9.4.1 The Intuition

In the previous chapter we outlined informally how reactions can be deduced from the properties of parts. While this process is simple in principle, it is complicated by modularity. We demonstrate the formal process with the following small example:

Table 9.4.1: An example of the translation from a part sequence to the associated reactions. A globally defined mRNA degradation rate, rdm , is assumed.

#	Part Sequence	Input	Output	Reactions
1	prom < con(rt) >		m	$g \rightarrow\{rt\} g + m$ $m \rightarrow\{rdm\}$
2	rbs < rate(r) >	m' , p'		$m' \rightarrow\{r\} m' + p'$
3	prom < con(rt) >; rbs < rate(r) >	p'		$g \rightarrow\{rt\} g + m$ $m \rightarrow\{rdm\}$ $m \rightarrow\{r\} m + p'$
4	pcr < codes(p, rd) >		p	$p \rightarrow\{rd\}$
5	prom < con(rt) >; rbs < rate(r) >; pcr < codes(p, rd) >			$g \rightarrow\{rt\} g + m$ $m \rightarrow\{rdm\}$ $m \rightarrow\{r\} m + p$ $p \rightarrow\{rd\}$

```
prom<con(rt)>; rbs<rate(r)>; pcr<codes(p, rd)>
```

Here we have included the **con** and **rate** properties explicitly since in the following examples and definitions, reactions are only generated from properties which are listed explicitly in programs. In practice however, we assume that **con** and **rate** properties with fresh variables are implicitly added to promoters and ribosome binding site programs if not given explicitly.

Each of the parts is translated to a corresponding set of reactions, as shown in Table 9.4.1. The actual species used in the reactions sometimes depend on the context in which the parts are placed. Thus, the reactions can take some species as “inputs” from neighbouring parts, and produce other species as “outputs” for other parts, as demonstrated with the input and output columns of the table.

We explain the translation to reactions by considering the first two parts in isolation, then the first sequential composition, the third part, and finally the third sequential composition, following the ordering shown in Table 9.4.1.

1. The translation of the promoter outputs a transcription reaction of the form $g \rightarrow\{rt\} g + m$ and a degradation reaction of the form $m \rightarrow\{rdm\}$ where g and m

are fresh identifiers representing respectively a gene and an mRNA. The evaluation also outputs the name m of the mRNA since this is necessary for subsequent sequential compositions.

2. The translation of the ribosome binding site outputs a translation reaction of the form $m' \rightarrow\{r\} m' + p'$ where m' is the mRNA and p' is the protein being produced. But neither m' nor p' is known until the ribosome binding site is placed in a context of other parts. Hence the translation of the ribosome binding site gives rise to a *function* which is parameterised on m' and p' : $f(m', p') \stackrel{\Delta}{=} m' \rightarrow\{r\} m' + p'$.
3. The left-most sequential composition can now be translated by applying the function f obtained from the ribosome binding site to the mRNA m obtained from the promoter. But since the second parameter of f is not yet known, this results in a new function $g(p') \stackrel{\Delta}{=} f(m, p') = m \rightarrow\{r\} m + p'$ with just a single parameter. The translation hence outputs g together with the reactions already obtained from the promoter.
4. The translation of the protein coding region immediately gives rise to a degradation reaction, and also to the name of the protein coded for which is needed for the subsequent composition. Hence the translation outputs both the degradation reaction $p \rightarrow\{rd\}$ and the identifier p .
5. The right-most sequential composition can now be translated by applying the function g obtained previously to the protein p obtained from the protein coding region, resulting in the reaction $g(p) = m \rightarrow\{r\} m + p$. The translation outputs this new reaction together with the reactions already obtained.

The translation is complicated further in the presence of compartments and transport. Compartments are handled by representing reactions as programs in a small language resembling LBS and CBS; we call these *reaction template programs* since they may generally contain variables. Transport reactions complicate matters because protein degradation reactions may need to be placed in compartments in which the protein is not expressed. To address this, the translation function returns both a program representing reactions and a separate set of protein degradation reactions. After the translation, the protein degradation reactions can be placed in the relevant compartments and then composed with the other reactions.

9.4.2 The Definition

Here is the formal definition of our reaction template programs, where R and T are GEC reactions and transport reactions, respectively, as generated by the grammar in Table 9.1.1:

$$L ::= R \quad | \quad T \quad | \quad \mathbf{0} \quad | \quad L_1 | L_2 \quad | \quad id_c[L]$$

A program L can easily be translated to a CBS or LBS program, allowing e.g. the LBS tools to be used for simulation. Given a set $\{L_i\}$ of reaction template programs, we let $par\{L_i\}$ denote their parallel composition; the ordering is insignificant since parallel composition is commutative.

With our motivating example in mind, the denotation function takes the form:

$$\llbracket P \rrbracket_{gr}\Gamma, b = (L, D, M, Pr, F, G, H)$$

where

- L is a reaction template program.
- D is a finite set $\{R_i\}$ of protein degradation reactions.
- $M \subsetneq_{fin} U$ is a set of mRNA species.
- $Pr \subsetneq_{fin} U$ is a set of protein species.
- F is a set of functions of the form $f(m, p) = R$ mapping pairs $(m, p) \in U \times U$ of mRNA and protein species to a reaction.
- G is a set of functions of the form $g(m) = R$ mapping an mRNA species $m \in U$ to a reaction.
- H is a set of functions of the form $h(p) = R$ mapping a protein species $p \in U$ to a reaction.

The denotation function is defined in the following; again we assume a global mRNA degradation rate rdm , and in order to clarify the presentation, we write reaction template programs L in a notation that resembles the concrete syntax of LBS.

- $\llbracket u : \mathbf{prom}(Q) \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} (L, \emptyset, \{m\}, \emptyset, \emptyset, \emptyset, \emptyset)$ where
 - $g \stackrel{\Delta}{\simeq} 0b$

- $m \stackrel{\Delta}{\simeq} 1b$
- $reacs(\mathbf{con(rt)}) \stackrel{\Delta}{\simeq} g ->\{rt\} g + m$
- $reacs(\mathbf{pos}(s, rb, rub, rtb)) \stackrel{\Delta}{\simeq}$
 $g + s ->\{rb\} g - s | g - s ->\{rub\} g + s | g - s ->\{rtb\} g - s + m$
- $reacs(\mathbf{neg}(s, rb, rub, rtb)) \stackrel{\Delta}{\simeq}$
 $g + s ->\{rb\} g - s | g - s ->\{rub\} g + s | g - s ->\{rtb\} g - s + m$
- $L \stackrel{\Delta}{\simeq} par\{reacs(q) \mid q \in Q\} \mid m ->\{rdm\}$
- $\llbracket u : \mathbf{rbs}(\{\mathbf{rate}(r)\}) \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} (\mathbf{0}, \emptyset, \emptyset, \emptyset, \{f\}, \emptyset, \emptyset)$ where
 - $f(m, p) \stackrel{\Delta}{\simeq} m ->\{r\} p$
- $\llbracket u : \mathbf{pcr}(\{\mathbf{codes}(p, r)\}) \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} (\mathbf{0}, \{p ->\{r\}\}, \emptyset, \{p\}, \emptyset, \emptyset, \emptyset)$
- $\llbracket u : \mathbf{ter} \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} (\mathbf{0}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
- $\llbracket \mathbf{0} \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} (\mathbf{0}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
- $\llbracket id_m(u) \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} P_1; P_2 \llbracket id_m \mapsto f \rrbracket_{gr}\Gamma, b$ where
 - $f(\underline{a}, b') \stackrel{\Delta}{\simeq} \llbracket P_1 \{ \underline{a}.i \mapsto \underline{a}.i \} \rrbracket_{gr}\Gamma, b'$
- $\llbracket id_m(\underline{a}) \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} f(\underline{a}, b)$ where
 - $f \stackrel{\Delta}{\simeq} \Gamma(id_m)$
- $\llbracket P \mid C \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} (L_1 \mid L_2, D, M, Pr, F, G, H)$ where
 - $(L_1, D, M, Pr, F, G, H) \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{gr}\Gamma, b$
 - $L_2 \stackrel{\Delta}{\simeq} \llbracket C \rrbracket_{gr}$
- $\llbracket P_1 \parallel P_2 \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} (L_1 \mid L_2, D_1 \cup D_2, M_1 \cup M_2, Pr_1 \cup Pr_2, F_1 \cup F_2, G_1 \cup G_2, H_1 \cup H_2)$ where
 - $(L_1, D_1, M_1, Pr_1, F_1, G_1, H_1) \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{gr}\Gamma, 0b$
 - $(L_2, D_2, M_2, Pr_2, F_2, G_2, H_2) \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{gr}\Gamma, 1b$
- $\llbracket P_1 ; P_2 \rrbracket_{gr}\Gamma, b \stackrel{\Delta}{\simeq} (L_1 \mid L_2 \mid L, D_1 \cup D_2, M, Pr, F, G, H)$ where
 - $(L_1, D_1, M_1, Pr_1, F_1, G_1, H_1) \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{gr}\Gamma, 0b$
 - $(L_2, D_2, M_2, Pr_2, F_2, G_2, H_2) \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{gr}\Gamma, 1b$

- $M \stackrel{\Delta}{\simeq} \begin{cases} M_1 & \text{if } M_2 = \emptyset \\ M_2 & \text{otherwise} \end{cases}$
- $Pr \stackrel{\Delta}{\simeq} \begin{cases} Pr_2 & \text{if } Pr_1 = \emptyset \\ Pr_1 & \text{otherwise} \end{cases}$
- $(F'_1, H'_1) \stackrel{\Delta}{\simeq} \begin{cases} (F_1, H_1) & \text{if } Pr_2 = \emptyset \\ (\emptyset, \emptyset) & \text{otherwise} \end{cases}$
- $(F'_2, G'_2) \stackrel{\Delta}{\simeq} \begin{cases} (F_2, G_2) & \text{if } M_1 = \emptyset \\ (\emptyset, \emptyset) & \text{otherwise} \end{cases}$
- $F \stackrel{\Delta}{\simeq} F'_1 \cup F'_2$
- $G \stackrel{\Delta}{\simeq} \{g \mid g(m) \stackrel{\Delta}{\simeq} f(m, p) \wedge f \in F_1 \wedge p \in Pr_2\} \cup G'_1 \cup G'_2$
- $H \stackrel{\Delta}{\simeq} \{h \mid h(p) \stackrel{\Delta}{\simeq} f(m, p) \wedge f \in F_2 \wedge m \in M_1\} \cup H'_1 \cup H'_2$
- $L \stackrel{\Delta}{\simeq} \text{par}\{g(m) \mid g \in G_2, m \in M_1\} \cup \text{par}\{h(p) \mid h \in H_1, p \in Pr_2\}$
- $\llbracket id_c[P] \rrbracket_{\text{gr}} \Gamma, b \stackrel{\Delta}{\simeq} (id_c[L], D, M, Pr, F, G, H)$ where
 - $(L, D, M, Pr, F, G, H) \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{\text{gr}} \Gamma, b$
- $\llbracket \text{new } x. P \rrbracket_{\text{gr}} \Gamma, b \stackrel{\Delta}{\simeq} \llbracket P \{x \mapsto b'0b\} \rrbracket_{\text{gr}} \Gamma, 1b$ where
 - b' is the shortest string in $\{b' \in \{0\}^* \mid b'0b \notin FV(P)\}$
- $\llbracket R^{v_b} \rrbracket_{\text{gr}} \stackrel{\Delta}{\simeq} R$
- $\llbracket T^{v_b} \rrbracket_{\text{gr}} \stackrel{\Delta}{\simeq} T$
- $\llbracket K \rrbracket_{\text{gr}} \stackrel{\Delta}{\simeq} \mathbf{0}$

We furthermore define $\llbracket P \rrbracket_{\text{gr}} \stackrel{\Delta}{\simeq} \llbracket P \rrbracket_{\text{gd}} \emptyset, \epsilon$.

Explanation of the denotation function The cases of module definition and invocation, and of new variables, are the same as the corresponding cases in the substitution and device template semantics. In the case of basic part programs, properties give rise to reactions and functions as outlined in the examples; fresh gene and mRNA species are constructed from the binary string parameter of the denotation function. Constraint composition and parallel composition simply give rise to pairwise unions.

In the case of sequential composition, the functions in the sets G_2 and H_1 are applied to respectively mRNA and proteins from M_2 and Pr_2 , giving rise to concrete reactions which are then composed in parallel. New sets G and H of functions parameterised on respectively mRNA and proteins are obtained in a similar fashion from appropriate instantiations of the functions in F_1 and F_2 .

Chapter 10

Conclusions

We evaluate our results with respect to the general aim of the thesis in Section 1. We discuss limitations and future work in Section 2.

10.1 Evaluation

Recall from the introduction that the general aim of the thesis is the development of formal languages for biology which:

1. allow one to write modular models of large cellular systems, and
2. allow one to write intuitively and concisely.

We have introduced two languages in pursuit of this aim, namely LBS for systems biology and GEC for synthetic biology. Both have been defined formally in terms of an abstract syntax and semantics, and compilers for the languages have been implemented.

We have demonstrated LBS and GEC through examples and case studies. LBS inherits some support for writing modular and intuitive models from CBS. We have contrasted our example LBS models with corresponding CBS models, clearly demonstrating how the LBS models achieve a higher degree of modularity through parameterisation and more concise models through species expressions and nondeterminism. We have shown that LBS can be used for large-scale modelling through case studies of the yeast pheromone and ErbB signalling pathways. We have also shown how modularity can be exploited in analysis in the case of Petri net flows.

In the case of GEC, our examples have shown how models can be composed from genetic parts in a concise manner. These examples should be contrasted with models

using other languages such as LBS, in which reactions for representing the dynamics of gene expression have to be written explicitly, and in which the geometric structure of genetic circuits is not readily apparent. Logical properties support the writing of intuitive models by allowing abstraction away from specific parts, and parameterised modules allow further abstraction away from the level of parts to e.g. genetic gates. We have shown through case studies how GEC can be used to model the repressilator and predator-prey systems, the latter of which represents one of the largest synthetic devices to date.

10.2 Future Work

10.2.1 LBS

Nondeterminism Nondeterminism, based on the `or` operator, provides a means to handling moderately combinatorial systems in a compact manner. Combinations within the same nondeterministic species can be restricted by using the restriction operator, `not`, and combinations between multiple nondeterministic species in reactions can be restricted by using conditionals. Conditionals can however only distinguish species based on their internal state. Although the identity of species can be encoded as internal states, doing so is cumbersome and artificial. One would therefore like language-level support for a more refined mechanism.

A type system The LBS denotation functions impose certain constraints on their arguments. The resulting notion of well-typedness is a dynamical one: the denotation of a module is a function, and whether or not this function is defined for a given set of actual parameters is determined by applying the function to these parameters. This approach falls short in two respects. First, the function may not be defined for any parameters at all. In this case it is the module definition that should be reported as ill-typed, rather than the module invocation. Second, well-typedness of a module invocation should be determined based on the actual parameters and an appropriate interface of the module, rather than by attempting to translate the body of the module under a given set of actual parameters. Hence one would like a dedicated type system which addresses these problems.

Concrete semantics We have given concrete semantics of LBS in terms of Petri nets, coloured Petri nets, ODEs, CTMCs, κ and Petri net flows. Additional concrete semantics, for example in terms of BioNetGen and BioPEPA, would be of interest. A BioNetGen concrete semantics can easily be defined based on the κ concrete semantics and would make LBS more accessible to modellers already using BioNetGen and its supporting tools. A BioPEPA concrete semantics would enable the large range of analysis tools supporting BioPEPA to be used in the context of LBS. Finally, having demonstrated how modularity can be exploited in the concrete Petri net flow semantics, it would be interesting to investigate if modularity can also be exploited elsewhere, e.g. in the κ analysis methods.

Visualisation Although LBS has been designed with ease of use in mind, it remains a textual language that may not be easily accessible to some biologists. Graphical representations of models can ameliorate this problem. Specifically, tools for visualising LBS programs and, conversely, for generating LBS programs from visual diagrams, would be useful. These tools might follow the *Systems Biology Graphical Notation* (SBGN) [54]. One challenge, not currently addressed in SBGN, is to devise a suitable graphical representation of parameterised modules.

Modelling We have demonstrated how LBS can be used in practical modelling applications by reproducing published models of the yeast pheromone and ErbB signalling pathways. More can be done for the latter, in particular by resolving the use of the COPY species which have been introduced in order to make the LBS model consistent with the published model. The LBS model also sheds light onto some potential inconsistencies which should be investigated further.

Going beyond the reproduction of existing models, one would like to apply LBS to the development of novel models in collaboration with biologists. This is likely to reveal practical problems to be addressed through further iterations of language design.

10.2.2 GEC

Genetic parts databases The translation of GEC models to devices relies on a database of genetic parts and their relevant biological properties. We have used a proof-of-concept database in this thesis, chosen with the aim of demonstrating the key features of GEC. However, in order for GEC to be useful in practical applications, a fully developed database of known parts is needed. One barrier to developing such a

database is the lack of characterisation data available for parts, although the situation is rapidly improving.

The translation of GEC programs also relies on a database of known reactions. Building a comprehensive database of this type is likely to be difficult in practice. There is also a question of representation to be addressed: instead of the multiset representation currently used, species in reactions can be represented at the lower level of binding domains as in κ . Choosing this level of representation could make profitable use of additional part types, such as *protein domains*, which are already present to some extent in the MIT Registry. This in turn would allow for more flexible models which are not limited by a fixed set of known proteins, and it could reduce the number of reactions recorded in the database.

We stress however that the reaction database is not essential for our approach; reactions can be “starred”, indicating that they are only used for simulation and not as constraints to be satisfied.

Constraint satisfaction engine The implementation of the GEC compiler includes a constraint satisfaction engine for selecting appropriate parts from the given database following the substitution denotational semantics of GEC. Currently this engine is implemented in Prolog. It uses the standard resolution algorithm of Prolog, and does not scale well. Its performance depends heavily on the ordering of constraints within the GEC model, and even for our proof-of-concept case studies, compilation can take in the order of minutes. This time will increase rapidly with the size of the model and the size of the database, which significantly limits the practical applicability of GEC. One would therefore like a more efficient implementation which takes advantage of dedicated constraint logic programming techniques.

A type system GEC does not have a notion of well-typedness, so it is possible to e.g. use a species identifier in places where a part identifier is expected without any error being flagged. A dedicated type system would hence be useful to prevent such situations. A type system could furthermore incorporate the idea of GenoCad of ensuring that part sequences are biologically meaningful.

Language development We have presented an idealised view that any device resulting from a GEC model can be readily implemented in a living cell. In reality, there are of course many factors which could prevent such an implementation from work-

ing. One must for example consider the impact of devices on the host cell physiology where the metabolic burden caused by circuit activation may overload the cell [66]. One would like such effects to be automatically deduced from GEC models, e.g. in terms of reactions modelling the relevant interactions between a device and a host cell. Extensions of the databases, and of the GEC language itself, may in turn be needed to facilitate such deductions.

Appendix A

The Yeast Pheromone Pathway in LBS

```
1 // Rate constant definitions:  
2 rate k1 = 0.03;  
3 rate k2 = 0.0012;  
4 rate k3 = 0.6;  
5 rate k4 = 0.24;  
6 rate k5 = 0.024;  
7 rate k6 = 0.0036;  
8 rate k7 = 0.24;  
9 rate k8 = 0.33;  
10 rate k9 = 2000;  
11 rate k10 = 0.1;  
12 rate k11 = 5;  
13 rate k12 = 1;  
14 rate k13 = 3;  
15 rate k14 = 1;  
16 rate k15 = 3;  
17 rate k16 = 3;  
18 rate k17 = 100;  
19 rate k18 = 5;  
20 rate k19 = 1;  
21 rate k20 = 10;  
22 rate k21 = 5;  
23 rate k22 = 47;  
24 rate k23 = 5;  
25 rate k24 = 345;  
26 rate k25 = 5;  
27 rate k26 = 50;  
28 rate k27 = 5;  
29 rate k28 = 140;
```

```

30 | rate k29 = 10;
31 | rate k30 = 1;
32 | rate k31 = 250;
33 | rate k32 = 5;
34 | rate k33 = 50;
35 | rate k34 = 18;
36 | rate k35 = 10;
37 | rate k36 = 0.1;
38 | rate k37 = 0.1;
39 | rate k38 = 0.01;
40 | rate k39 = 18;
41 | rate k40 = 1;
42 | rate k41 = 0.002;
43 | rate k42 = 0.1;
44 | rate k43 = 0.01;
45 | rate k44 = 0.01;
46 | rate k45 = 0.1;
47 | rate k46 = 200;
48 | rate k47 = 1;

49 |
50 // Fus3 is shared between most modules and is therefore global:
51 spec Fus3 = new{p:bool};

52 |
53 module ReceptorAct(comp cyto; spec degrador; specout ar) {
54   spec Alpha = new{};
55   init Alpha 1000.0 |

56 |
57   spec Ste2 = new{p:bool};
58   init cyto[Ste2] 1666.67 |

59 |
60 // pheromone and receptor degradation
61 cyto[degrador] ~ Alpha ->{k1} |
62 cyto[Ste2] ->{k5} |

63 |
64 // receptor activation:
65 Alpha + cyto[Ste2] ->{k2} Alpha + cyto[Ste2{p} as ar];
66 cyto[ar] ->{k3} cyto[Ste2];
67 // receptor-ligand degradation:
68 cyto[ar] ->{k4}
69 };
70 |
71 module GProtCycle(spec act; specout Gbg) {

```

```

72 spec Ga = new{}, Gbg = new{};
73 spec Sst2 = new{p:bool}, GDP = new{}, GTP = new{};
74 spec Gbga = Gbg-Ga-GDP;
75 init Gbga 1666.67 |

76
77 // disassociation of G-protein complex:
78 act ~ Gbga ->{k6} Gbg + Ga-GTP |
79 // ... and cycle:
80 Ga-GTP ->{k7} Ga-GDP |
81 Sst2{p} ~ Ga-GTP ->{k8} Ga-GDP |
82 // next reaction does not follow mass-action kinetics:
83 rate v46 =
84     k46 * ( Fus3{p}^2 / (4^2 + Fus3{p}^2) );
85 Fus3{p} ~ Sst2 <->[v46]{k47} Sst2{p} |
86 Ga-GDP + Gbg ->{k9} Gbga
87 };

88
89 // Species common to the remaining modules:
90 spec Ste11 = new{p:bool};
91 spec Ste7 = new{p:bool};
92 spec Ste5 = new{p:bool};

93
94 module ScaffoldForm(spec gbg; specout e : Fus3-Ste5-Ste7-Ste11) {
95     spec Ste20 = new{}; init Ste20 1000.0 |

96
97 // a sub-module for scaffold formation; hides gbg and Ste20:
98 module formation(specout e:Ste5-Ste7-Ste11-Fus3) {
99     Ste11 + Ste5 <->{k12}{k13} Ste11-Ste5 as a; init a 105.94 |
100    Ste7 + Fus3 <->{k14}{k15} Ste7-Fus3 as b; init b 77.87 |
101    a + b ->{k16} a-b as c; init c 235.72 |
102    c + gbg <->{k10}{k11} c-gbg as d;
103    d + Ste20 <->{k18}{k19} d-Ste20 as e;
104    c ->{k17} Ste11 + Ste5 + Ste7 + Fus3
105};

106
107 // ... and a submodule for degradation:
108 module degradation(spec complex; rate r) {
109     complex ->{r} Ste5 + Ste7 + Ste11 + Ste20 + Fus3 + gbg
110};

111
112 // invoke the formation module:
113 formation(spec e);

```

```

114
115 // invoke degradation for each complex modification:
116 degradation(e, k21);
117 spec f = e<Ste11{p}>;
118 degradation(f, k23);
119 spec g = f<Ste7{p}>;
120 degradation(g, k25);
121 spec h = g<Fus3{p}>;
122 degradation(h, k27);
123 spec l = (h\>Fus3)<Ste5{p}>;
124
125 // cannot use module to degrade l, as this does not have Fus3:
126 l ->[k32 * 1] Ste5 + Ste7 + Ste11 + Ste20 + gbg
127 };
128
129 module MAPKCascade(spec e : mk1{p}-mk2{p}-mk3{p}; specout h : e) {
130   e ->{k20} e<mk3{p}> as f;
131   f ->{k22} f<mk2{p}> as g;
132   g ->{k24} g<mk1{p}> as h
133 };
134
135 module RepeatedFus3Phos(spec h : Fus3{p}-Ste5{p}) {
136   h ->{k26} h<Ste5{p}> as i;
137   i ->{k28} i\>Fus3 as l + i.Fus3;
138   l + Fus3 <->[k29]{k30} l-Fus3 as k;
139   k ->{k31} i |
140   Fus3{p} ->[k33] Fus3
141 };
142
143 module PrepMating(comp nucleus; spec gbg)
144 {
145   spec Far1 = new{p:bool,u:bool};
146   init nucleus[Far1] 500.0 |
147
148   spec Cdc28 = new{};
149   init nucleus[Cdc28] 300.0 |
150
151 // next reaction does not use mass-action kinetics:
152 rate v39 = k39 * nucleus[Far1] *
153           Fus3{p}^2 / (100^2 + Fus3{p}^2);
154 Fus3{p} ~ nucleus[Far1] <->[v39]{k40} nucleus[Far1{p}];
155

```

```

156 nucleus [
157   Cdc28 ~ Far1 ->{k41} Far1{u} |
158   Far1{p} + Cdc28 <->{k45}{k44} Far1{p}-Cdc28
159 ] |
160 nucleus [ Far1{p} ] + gbg <->{k42}{k43} nucleus [ Far1{p}-gbg ]
161 } ;
162
163 module GeneExpAlt(spec Bar1:{ act:bool}; comp cytosol , nucleus) {
164   spec Ste12 = new{ act:bool} ;
165   init cytosol[ nucleus[ Ste12 ] ] 200.0 |
166
167   cytosol [
168     spec act = Fus3{p}-Ste12{ act } ;
169     Fus3{p} + nucleus[ Ste12 ] <->{k34}{k35} nucleus[ act ];
170     nucleus[ act ~ Bar1 <->{k36}{k37} Bar1{ act } ]
171   ] |
172
173   Bar1{ act } ->{k38}
174 };
175
176 // main body of the module. first declare our compartments: *)
177 comp cytosol = new comp;
178 comp nucleus = new comp inside cytosol;
179
180 spec Bar1 = new{ act:bool} ;
181 init cytosol[ nucleus[ Bar1 ] ] 200.0 |
182
183 ReceptorAct(cytosol , nucleus[ Bar1{ act } ] , spec ar );
184
185 cytosol [
186   init Fus3 686.40 |
187   init Ste11 158.33 |
188   init Ste7 36.40 |
189   init Ste5 158.33 |
190
191   GProtCycle(ar , spec gbg);
192   ScaffoldForm(gbg , spec e);
193   MAPKCascade(e : Fus3{p}-Ste7{p}-Ste11{p} , spec h);
194   RepeatedFus3Phos(h : Fus3{p}-Ste5{p});
195   PrepMating(nucleus , gbg)
196 ] |
197

```

198 | GeneExpAlt(Bar1:{ act }, cytosol , nucleus)

Appendix B

The ErbB Pathway in LBS

```
1 // compartment definitions:  
2 comp plas = new comp; // plasma membrane  
3 comp endo = new comp; // endosomal membrane  
4 comp cyto = new comp; // cytosol  
5 comp medium = new comp; // extra-cellular space  
6 comp world = new comp; // top-level compartment  
7 comp endosomes = new comp; // endosomes  
8 comp lysosomes = new comp; // lysosomes  
9  
10 // rate constant definitions:  
11 rate kd1 = 0.0033;  
12 rate k1c = 800;  
13 rate kd1c = 1;  
14 rate kd1d = 0.1;  
15 rate k1d = 518;  
16 rate k2 = 7.44622E-06;  
17 rate kd2 = 0.16;  
18 rate k2b = 3.73632E-08;  
19 rate kd2b = 0.016;  
20 rate k3 = 1;  
21 rate kd3 = 0.001;  
22 rate k4 = 6.73E-06;  
23 rate kd4 = 0.000166;  
24 rate k4b = 0;  
25 rate kd4b = 0.000166;  
26 rate k5 = 0;  
27 rate kd5 = 0.80833;  
28 rate k5b = 0;  
29 rate kd5b = 0.0080833;
```

```
30 | rate kd5c = 0.162;  
31 | rate k6 = 0.013;  
32 | rate kd6 = 5E-05;  
33 | rate k8 = 5.91474E-07;  
34 | rate kd8 = 0.2;  
35 | rate kd8b = 0.02;  
36 | rate k8b = 9.34641E-06;  
37 | rate k10 = 140000;  
38 | rate k10b = 0.05426;  
39 | rate kd10 = 0.011;  
40 | rate k13 = 0;  
41 | rate kd13 = 0;  
42 | rate k14 = 0;  
43 | rate kd14 = 0;  
44 | rate k15 = 1.667E-08;  
45 | rate kd15 = 0;  
46 | rate k16 = 1.67E-05;  
47 | rate k16b = 1.667E-07;  
48 | rate k17 = 1.67E-05;  
49 | rate kd17 = 0.06;  
50 | rate k18 = 2.5E-05;  
51 | rate kd18 = 1.3;  
52 | rate k19 = 1.667E-07;  
53 | rate kd19 = 0.5;  
54 | rate k20 = 1.1068E-05;  
55 | rate kd20 = 0.4;  
56 | rate k21 = 3.67E-07;  
57 | rate kd21 = 0.23;  
58 | rate k22 = 1.39338E-07;  
59 | rate kd22 = 0.1;  
60 | rate k23 = 6;  
61 | rate kd23 = 0.06;  
62 | rate kd24 = 0.55;  
63 | rate k25 = 1.67E-05;  
64 | rate kd25 = 0.0214;  
65 | rate k28 = 5E-06;  
66 | rate kd28 = 0.0053;  
67 | rate k28b = 5E-06;  
68 | rate kd28b = 0.0053;  
69 | rate k29 = 1.17E-06;  
70 | rate kd29 = 3.1;  
71 | rate kd32 = 0.1;
```

```
72 | rate k32 = 4E-07;
73 | rate kd33 = 0.2;
74 | rate k33 = 3.5E-05;
75 | rate kd34 = 0.03;
76 | rate k34 = 7.5E-06;
77 | rate kd35 = 0.0015;
78 | rate k35 = 7.5E-06;
79 | rate k36 = 0.005;
80 | rate kd36 = 0;
81 | rate kd37 = 0.3;
82 | rate k37 = 1.5E-06;
83 | rate k40 = 5E-05;
84 | rate kd40 = 0.064;
85 | rate k41 = 5E-05;
86 | rate kd41 = 0.0429;
87 | rate k42 = 6E-05;
88 | rate kd42 = 0.0141589;
89 | rate kd43 = 31.6228;
90 | rate k43 = 0;
91 | rate kd44 = 0.01833;
92 | rate kd45 = 1.9;
93 | rate k45 = 0;
94 | rate kd47 = 0.8;
95 | rate k47 = 0;
96 | rate k48 = 2.37E-05;
97 | rate kd48 = 0.79;
98 | rate kd49 = 0.112387;
99 | rate k49 = 0;
100 | rate k50 = 4.74801E-08;
101 | rate kd50 = 0.252982;
102 | rate kd52 = 0.033;
103 | rate kd53 = 0.28;
104 | rate k53 = 0;
105 | rate kd55 = 70.1662;
106 | rate k55 = 0;
107 | rate kd56 = 5;
108 | rate k56 = 0.000397392;
109 | rate kd57 = 0.0076;
110 | rate k57 = 0;
111 | rate k58 = 8.33E-07;
112 | rate kd58 = 56.7862;
113 | rate k52 = 8.85125E-06;
```

```
114 rate k44 = 1.07E-05;  
115 rate k60 = 0.00266742;  
116 rate kd60 = 0;  
117 rate k61 = 0.00057;  
118 rate kd61 = 0;  
119 rate kd63 = 0.275;  
120 rate k64 = 1.67E-05;  
121 rate kd64 = 0.3;  
122 rate kd65 = 0.2;  
123 rate k65 = 0;  
124 rate k66 = 1.5E-05;  
125 rate kd66 = 0.2;  
126 rate k67 = 5E-05;  
127 rate kd67 = 0.02;  
128 rate kd68 = 0.2;  
129 rate k68 = 0;  
130 rate kd68b = 20.5;  
131 rate k69 = 3.33E-05;  
132 rate kd69 = 0.1;  
133 rate k70 = 6.67E-07;  
134 rate kd70 = 0.1;  
135 rate k71 = 0;  
136 rate kd71 = 25.2;  
137 rate k72 = 0;  
138 rate kd72 = 5.01187;  
139 rate k73 = 0.00374845;  
140 rate kd73 = 0.5;  
141 rate k74 = 6.36184E-07;  
142 rate kd74 = 0.355656;  
143 rate kd75 = 0.00633957;  
144 rate k75 = 0;  
145 rate k76 = 0;  
146 rate kd76 = 142.262;  
147 rate kd60d = 0;  
148 rate k22b = 3.5E-05;  
149 rate kd22b = 0.1;  
150 rate kd34b = 0.1;  
151 rate k34b = 7.5E-05;  
152 rate k94b = 5E-05;  
153 rate k94 = 5E-05;  
154 rate kd94 = 0.01;  
155 rate k95 = 0;
```

```
156 | rate kd95 = 33;  
157 | rate k96 = 1.67E-06;  
158 | rate kd96 = 0.1;  
159 | rate kd6b = 0;  
160 | rate k7 = 5E-05;  
161 | rate kd7 = 0.000138;  
162 | rate k62b = 0.000416;  
163 | rate kd60b = 0;  
164 | rate k60c = 0.00052;  
165 | rate k60b = 0.0471248;  
166 | rate k97 = 1000000;  
167 | rate kd97 = 0.015;  
168 | rate k97c = 1000000;  
169 | rate kd97c = 0.001;  
170 | rate kd98 = 0.001;  
171 | rate k98 = 33300;  
172 | rate Kinh4 = 0.113;  
173 | rate kd99 = 0.5;  
174 | rate k99 = 4.42;  
175 | rate Kinh3 = 0.001;  
176 | rate kd100 = 0.001;  
177 | rate k100 = 1;  
178 | rate k101 = 8.33E-07;  
179 | rate kd101 = 0.03;  
180 | rate k102 = 5E-07;  
181 | rate kd102 = 5.61009;  
182 | rate k103 = 8.36983E-09;  
183 | rate kd103 = 0.016;  
184 | rate k104 = 0;  
185 | rate kd104 = 0.2;  
186 | rate k105 = 6.67E-05;  
187 | rate kd105 = 0.1;  
188 | rate k106 = 1.33E-05;  
189 | rate kd106 = 0.1;  
190 | rate k106b = 2.63418E-08;  
191 | rate kd106b = 0.1;  
192 | rate k107 = 3.33E-05;  
193 | rate kd107 = 0.1;  
194 | rate k108 = 0;  
195 | rate kd108 = 5;  
196 | rate k109 = 5E-06;  
197 | rate kd109 = 0.1;
```

```

198 | rate k110 = 0.000333;
199 | rate kd110 = 0.1;
200 | rate kd111 = 6.57;
201 | rate k111 = 0;
202 | rate k112 = 0.0047067;
203 | rate kd112 = 0.1;
204 | rate k113 = 0;
205 | rate kd113 = 177.828;
206 | rate k114 = 4.98816E-06;
207 | rate kd114 = 0.1;
208 | rate k115 = 0;
209 | rate kd115 = 1;
210 | rate k116 = 0.0150356;
211 | rate kd116 = 0;
212 | rate k117 = 8.33E-08;
213 | rate kd117 = 0.1;
214 | rate k118 = 0;
215 | rate kd118 = 0.03;
216 | rate kd119 = 0.0103115;
217 | rate k119 = 10000000;
218 | rate k120 = 1.48131E-08;
219 | rate kd120 = 0.1;
220 | rate k120b = 5.92538E-11;
221 | rate kd120b = 0.1;
222 | rate Ks = 0.001;
223 | rate k121 = 0.001;
224 | rate kd121 = 1;
225 | rate kd122 = 1;
226 | rate k123 = 0;
227 | rate kd123 = 0.177828;
228 | rate k6b = 0;
229 | rate k1 = 10000000;
230 | rate k122 = 1.8704E-08;
231 | rate k123h = 0;
232 | rate kd123h = 0.1;
233 |
234 // atomic species definitions:
235 | spec EGF = new{};
236 | spec ErbB2 = new{};
237 | spec ErbB3 = new{};
238 | spec ErbB4 = new{};
239 | spec ATP = new{};

```

```

240 | spec cPP = new {};
241 | spec GAP = new {};
242 | spec HRG = new {};
243 | spec Grb2 = new {};
244 | spec Shc = new{p1:bool};
245 | spec Sos = new {};
246 | spec Raf = new {};
247 | spec Raf = new{p1:bool};
248 | spec Pase1 = new {};
249 | spec MEK = new {};
250 | spec MEK = new{p1:bool};
251 | spec MEK = new{p1:bool, p2:bool};
252 | spec Pase2 = new {};
253 | spec ERK = new {};
254 | spec ERK = new{p1:bool};
255 | spec ERK = new{p1:bool, p2:bool};
256 | spec Pase3 = new {};
257 | spec Sos = new{p1:bool};
258 | spec PI3K = new {};
259 | spec PIP3 = new {};
260 | spec AKT = new {};
261 | spec AKT = new{p1:bool};
262 | spec PDK1 = new {};
263 | spec AKT = new{p1:bool, p2:bool};
264 | spec Pase4 = new {};
265 | spec ErbB2 = new{p1:bool};
266 | spec ErbB1 = new {};
267 | spec Inh = new{e1:bool, e2:bool, e3:bool, e4:bool};
268 | spec ErbB3 = new{p1:bool};
269 | spec ErbB4 = new{p1:bool};
270 | spec Shp = new {};
271 | spec PIP2 = new {};
272 | spec PTEN = new {};
273 | spec Gab1 = new {};
274 | spec Shp2 = new {};
275 | spec Pase9t = new {};
276 | spec Ras = new {};
277 | spec GDP = new {};
278 | spec GTP = new {};
279 | spec i = new {};
280 | spec h = new {};
281 | spec Pase = new {};

```

```

282 spec Pase4 = new{};
283 spec RTK = new{};
284 spec R = new{};
285 spec degraded = new{};
286 spec activated = new{};
287 spec ErbB1 = new{p1:bool};
288 spec Gab1 = new{p1:bool, p2:bool};
289 spec PIP22 = new{};
290 spec PIP23 = new{};
291 spec PIP24 = new{};
292 spec PIP25 = new{};
293 spec PIP26 = new{};
294 spec AKT = new{p1:bool, p2:bool};
295 spec ErbB34 = new{p1:bool};
296 spec ErbB22 = new{p1:bool};
297 spec MKP = new{};
298 spec FullActive = new{};
299 spec HalfActive = new{};
300 spec Ser = new{};
301 spec deg = new{};
302 spec COPY = new{};

303
304 // some nondeterministic species used throughout
305 spec ErbB234 = (ErbB2:{p1} or ErbB3:{p1} or ErbB4:{p1})::ErbB234{p1};
306 spec iSNil = SNil or i;

307
308 // initial populations:
309 init world[ATP] 1200000000 |
310 init medium[EGF] 5E-09 |
311 plas[
312   init ErbB2 462000 |
313   init ErbB3 6230 |
314   init ErbB4 794 |
315   init cPP 4498.73 |
316   init ErbB1 1080000
317 ] |
318 cyto[
319   init GAP 534751 |
320   init Grb2 1264.91 |
321   init GDP-Ras 58095.2 |
322   init Shc 1100000 |
323   init Raf 71131.2 |

```

```

324  init Grb2–Sos 88914000 |
325  init Pase1 50000 |
326  init MEK 3020000 |
327  init Pase2 124480 |
328  init ERK 695000 |
329  init Pase3 16870.2 |
330  init PI3K 35565600 |
331  init AKT 905000 |
332  init PDK1 300416000 |
333  init Pase4 450000 |
334  init Pase–RTK 70000 |
335  init Shp 2213.59 |
336  init PIP2 393639 |
337  init PTEN 56100.9 |
338  init Gab1 94868.3 |
339  init Shp2 1000000
340 ] |
341
342 // module definitions
343 module receptorLigandBinding() {
344   spec recept1 = ErbB1–(ATP–(SNil or h) or Inh{e1}–(SNil or h));
345   medium[EGF] + plas[recept1]
346     <–>{k1 , kd1} medium[EGF] + plas[recept1–EGF] |
347
348   medium[EGF] + plas[ErbB2–ErbB3]
349     <=⇒{k1c , kd1c} medium[EGF] + plas[ErbB2{p1}–ErbB3{p1}] |
350   medium[EGF] + plas[ErbB2–ErbB4]
351     <=⇒{k1d , kd1d} medium[EGF] + plas[ErbB2{p1}–ErbB4{p1}] |
352
353   spec recept2 = ErbB3 or ErbB4;
354   medium[HRG] + plas[recept2]
355     <–>{k119 , kd119} medium[HRG] + plas[recept2–HRG] |
356
357   endosomes[EGF] + endosomes[ATP–ErbB1–h]
358     <=⇒{k10b , kd10} endo[ATP–EGF–ErbB1] |
359   endosomes[EGF] + endo[ATP–ErbB1]
360     <=⇒{k10b , kd10} endo[ATP–EGF–ErbB1] |
361   endosomes[HRG] + endo[ErbB3]
362     <=⇒{k10b , kd10} endo[ErbB3–HRG]
363 };
364
365 module receptorInhibition() {

```

```

366 plas [ErbB1] + medium[Inh]
367   <=>{k97 ,kd97} medium[Inh] + plas [ErbB1–Inh{e1}] |
368 plas [ErbB2] + medium[Inh]
369   <=>{k98 ,kd98} medium[Inh] + plas [ErbB2–Inh{e2}] |
370 plas [ErbB4] + medium[Inh]
371   <=>{k99 ,kd99} medium[Inh] + plas [ErbB4–Inh{e4}] |
372 plas [ErbB3] + medium[Inh]
373   <=>{k100 ,kd100} medium[Inh] + plas [ErbB3–Inh{e3}] |
374 plas [ErbB1–h] + medium[Inh]
375   <=>{k97c ,kd97c} medium[Inh] + plas [ErbB1–Inh{e1}–h]
376 };
377
378 module receptorDimerisation() {
379   plas [ErbB2{p1} + ErbB2{p1} <=>{k96 ,kd96} 2.ErbB2{p1}] |
380
381   plas [
382     spec complex =
383       ATP–EGF–ErbB1–(SNil or h) or EGF–ErbB1–Inh{e1}–(SNil or h);
384       ATP–EGF–ErbB1 as c + complex <->{k2 ,kd2} c–complex |
385
386       // the following does not generalise because of the COPY species .
387       EGF–ErbB1–Inh{e1} as c + c <=>{k2 ,kd2} 2.c |
388       EGF–ErbB1–Inh{e1}–h as c + c <=>{k2 ,kd2} 2.c |
389       (EGF–ErbB1–Inh{e1} as c)–h + c <=>{k2 ,kd2} 2.c–h |
390       (EGF–ErbB1 as c)–Inh{e1} + ATP–c–h
391       <=>{k2 ,kd2} ATP–(2.c)–Inh{e1}–h–COPY |
392       ATP–EGF–ErbB1–h as c + c
393       <=>{k2 ,kd2} 2.c–FullActive |
394       ATP–(EGF–ErbB1 as c)–h + c–Inh{e1}–h
395       <=>{k2 ,kd2} ATP–(2.c)–Inh{e1}–h–COPY–COPY
396   ] |
397
398   plas [EGF–ErbB1–Inh{e1} as c + ErbB234 <->{k2b ,kd2b} c–ErbB234] |
399   endo [ATP–EGF–ErbB1 as c + c <=>{k2 ,kd2} 2.c] |
400
401   plas [
402     // should ErbB1/2 not be phosphorylated in the product?
403     ATP–EGF–ErbB1 + ErbB2–Inh{e2}
404     <=>{k2b ,kd2b} EGF–ErbB1–ErbB2–Inh{e2} |
405     // should the product have ErbB4 instead of ErbB3?
406     ATP–EGF–ErbB1 + ErbB4–Inh{e4}
407     <=>{k2b ,kd2b} EGF–ErbB1{p1}–ErbB3{p1}–Inh{e4} |

```

```

408    ATP–EGF–ErbB1 + ErbB3–Inh{e3}
409    <=>{k2b ,kd2b} EGF–ErbB1{p1}–ErbB3{p1}–Inh{e3}
410  ] |
411
412  endo [ 2.(EGF–ATP–ErbB1) as c ] + world[ATP]
413  <=>{k122 ,kd122} cyto[c] |
414  plas [ 2.(EGF–ATP–ErbB1) as c ] + world[ATP]
415  <=>{k122 ,kd122} cyto[c–FullActive] |
416  plas [EGF–ErbB1–ErbB234 as c ] + world[ATP]
417  <->{k122 ,kd122} cyto[ATP–c–COPY] |
418
419  plas [ErbB2{p1}] + plas [ErbB2]
420  <=>{k103 ,kd103} endo[ErbB2–ErbB2{p1}] |
421  plas [
422    spec complex = ErbB234{p1};
423    EGF–ErbB1{p1} + complex <->{k102 ,kd102} ErbB1{p1}–complex |
424    EGF–ErbB1{p1} + EGF–ErbB1{p1} <=>{k102 ,kd102} 2.(EGF–ErbB1{p1}) |
425
426    // NOTE: example of use of variables .
427    spec e34 = ErbB3{p1= $x} or ErbB4{p1= $x};
428    spec e2 = ErbB2{p1=$x};
429    e2 + e34 <->{k103 ,kd103} e2–e34 |
430
431    spec e34 = ErbB3 or ErbB4;
432    ErbB2–Inh{e2} + e34 <->{k103 ,kd103} ErbB2–Inh{e2}–e34 |
433    ErbB2{p1} + ErbB2–Inh{e2} <=>{k103 ,kd103} ErbB2–ErbB2–Inh{e2} |
434    ErbB2 + ErbB4–Inh{e4} <=>{k103 ,kd103} ErbB2–ErbB4–Inh{e4}
435  ] |
436
437  // the following reactions take place in both plas and endo .
438  // asbtract into module and invoke with these :
439  module reacs(comp d) {
440    d[ATP–(EGF–ErbB1 as c) + ErbB234 <->{k2b ,kd2b} c–ErbB234] |
441
442    spec complex = 2.(EGF–ErbB1{p1})–GAP–Grb2–(SNil or Shc{p1});
443    cyto[Sos{p1}] + d[complex] <->{k101 ,kd101} d[complex–Sos{p1}] |
444
445    spec complex = (ErbB3 or ErbB4)–HRG;
446    d[complex + ErbB2 <->{k120 ,kd120} complex–ErbB2] |
447
448    spec complex = (ErbB3 or ErbB4)–HRG;
449    d[complex + ATP–ErbB1 <->{k120b ,kd120} complex–ErbB1]
```

```

450    };
451    reacs(plas) | reacs(endo)
452  };
453
454 module receptorActivation() {
455   spec complex = ErbB2-ErbB2{p1} or EGF-ErbB1-ErbB234;
456   endo[complex] + world[ATP] <->{k122,kd122} cyto[ATP-complex] |
457
458   spec receptors =
459     (ErbB1{p1} or ErbB2{p1})-ErbB234{p1} or 2.(EGF-ErbB1{p1});
460   spec complex =
461     receptors-Gab1-GAP-Grb2 or ErbB1-(ErbB3 or ErbB4)-HRG;
462   plas[complex] + world[ATP] <->{k122,kd122} cyto[ATP-complex] |
463
464   // difficult to generalise due to the use of the
465   // the use of the COPY species and dif compartments.
466   plas[ErbB2-ErbB4-HRG as c] + world[ATP]
467     <=>{k122,kd122} cyto[ATP-c] |
468   plas[ErbB2-ErbB3-HRG as c] + world[ATP]
469     <=>{k122,kd122} plas[ATP-c] |
470   endo[ErbB2-ErbB3-HRG as c] + world[ATP]
471     <=>{k122,kd122} plas[ATP-c-COPY] |
472   endo[ErbB2-ErbB4-HRG as c] + world[ATP]
473     <=>{k122,kd122} endo[ATP-c] |
474
475   // the following seems inconsistent:
476   plas[ATP-EGF-EGF-ErbB1-ErbB1-Inh{e1} as c] + world[ATP]
477     <=>{k122,kd122} plas[c-HalfActive] |
478   plas[(ATP-(2.(EGF-ErbB1))-Inh{e1}-h as c)-COPY] + world[ATP]
479     <=>{k122,kd122} plas[c-HalfActive] |
480   plas[(ATP-(2.(EGF-ErbB1))-Inh{e1}-h-COPY as c)-COPY] + world[ATP]
481     <=>{k122,kd122} plas[c-HalfActive] |
482   plas[ErbB1] + world[ATP]
483     <=>{k122,kd122} plas[ATP-ErbB1] |
484   plas[ErbB1-h] + world[ATP]
485     <=>{k122,kd122} plas[ATP-ErbB1-h] |
486   plas[2.(ATP-EGF-ErbB1)-h as c] + world[ATP]
487     <=>{k122,kd122} plas[c-FullActive] |
488   plas[2.(EGF-ATP-ErbB1-h)-FullActive as c] + world[ATP]
489     <=>{k122,kd122} plas[c-COPY] |
490   plas[ATP-(2.(EGF-ErbB1))-Inh{e1}-h as c] + world[ATP]
491     <=>{k122,kd122} cyto[c-HalfActive] |

```

```

492
493 spec complex =
494   (ErbB1{p1}:{p1})-(ErbB234{p1}:{p1}) :: ErbB1{p1}-ErbB234{p1} ;
495   plas [ complex ] + world [ATP] <->{k123 ,kd123}
496   cyto [ATP-EGF-complex<ErbB1{p1=ff}><ErbB234{p1=ff}>-COPY] |
497   endo [ complex ] + world [ATP] <->{k123 ,kd123}
498   cyto [ATP-EGF-complex<ErbB1{p1=ff}><ErbB234{p1=ff}>] |
499
500 spec receptors =
501   (ErbB1{p1} or ErbB2{p1})-ErbB234{p1} or 2.(EGF-ErbB1{p1});
502 spec complex =
503   receptors-Gab1{p1}-GAP-Grb2 ;
504   plas [ complex ] + world [ATP]
505   <->{k123 ,kd123} cyto [ATP-complex<Gab1{p1=ff}>] |
506
507 // strange that HRG magically appears on RHS. does not easily
508 // generalise due to COPY and the use of dif compartments.
509   plas [ErbB1{p1}-ErbB3{p1}] + world [ATP]
510   <=>{k123 ,kd123} cyto [ATP-ErbB1-ErbB3-HRG] |
511   plas [ErbB1{p1}-ErbB4{p1}] + world [ATP]
512   <=>{k123 ,kd123} cyto [ATP-ErbB1-ErbB4-HRG] |
513   plas [ErbB2{p1}-ErbB4{p1}] + world [ATP]
514   <=>{k123 ,kd123} cyto [ATP-ErbB2-ErbB4-HRG] |
515   endo [ErbB2{p1}-ErbB4{p1}] + world [ATP]
516   <=>{k123 ,kd123} endo [ATP-ErbB2-ErbB4-HRG] |
517   plas [ErbB2{p1}-ErbB3{p1}] + world [ATP]
518   <=>{k123 ,kd123} plas [ATP-ErbB2-ErbB3-HRG] |
519   endo [ErbB2{p1}-ErbB3{p1}] + world [ATP]
520   <=>{k123 ,kd123} plas [ATP-ErbB2-ErbB3-HRG-COPY] |
521   endo [2.(EGF-ErbB1{p1})] + world [ATP]
522   <=>{k123 ,kd123} cyto [2.(EGF-ErbB1-ATP)] |
523 // strange that only one ErbB2 gets phosphorylated:
524   plas [2.ErbB2{p1}] + world [ATP]
525   <=>{k123 ,kd123} cyto [ATP-ErbB2-ErbB2{p1}] |
526
527 // the following appears inconsistent and does not easily
528 // generalise because of COPY, HalfActive and FullActive:
529 spec rhs =
530   2.(EGF-ErbB1)-ATP-(ATP-FullActive-h or
531   ATP-FullActive-COPY-h-h or HalfActive-Inh{e1});
532   plas [2.(EGF-ErbB1{p1})] + world [ATP] <->{k123 ,kd123} plas [rhs] |
533 spec rhs =

```

```

534    2.(EGF-ErbB1)-ATP-(ATP-FullActive or h-HalfActive-Inh{e1});  

535    plas[2.(EGF-ErbB1{p1})] + world[ATP] <->{k123,kd123} cyto[rhs] |  

536    spec rhs =  

537    2.(EGF-ErbB1)-ATP-h-HalfActive-Inh{e1}-(SNil or COPY);  

538    plas[2.(EGF-ErbB1{p1})] + world[ATP] <->{k123h,kd123h} plas[rhs]  

539  };  

540  

541 module cPPTagging() {  

542   spec scaffoldNoReceptor =  

543     SNil or Sos or Sos-(  

544       Ras-(GDP or GTP)  

545     ) or Shc{p1}-(  

546       SNil or Sos-(SNil or Ras-(GDP or GTP))  

547     );  

548   spec complex = 2.(EGF-ErbB1{p1})-GAP-Grb2-scaffoldNoReceptor;  

549   plas[complex + cPP <->{k4,kd4} cPP-complex] |  

550   endo[complex] + endo[cPP] <->{k5,kd5} plas[cPP-complex] |  

551  

552   // note: dif rate from the above, so cannot combine.  

553   spec complexAll =  

554     (ErbB1{p1} or ErbB2{p1})-(ErbB234{p1})-  

555       GAP-Grb2-scaffoldNoReceptor;  

556   spec complexDifRate = ErbB1{p1}-ErbB234{p1}-GAP-Grb2-GTP-Ras-Sos;  

557   spec complex = complexAll not complexDifRate;  

558   endo[complexAll] + endo[cPP] <->{k5b,kd5b} plas[cPP-complexAll] |  

559   plas[complexDifRate + cPP <->{k4,kd4} cPP-complexDifRate] |  

560   plas[complex + cPP <->{k4b,kd4} cPP-complex] |  

561  

562   endo[cPP] <=>{k15,kd15} plas[cPP]  

563 };  

564  

565 module endocytosis() {  

566   spec scaffoldNoReceptor =  

567     SNil or GAP or GAP-  

568     (Shc or Shc{p1} or Grb2-  

569       (SNil or Sos-(SNil or GTP-Ras) or  

570         Shc{p1}-(  

571           SNil or Sos-(  

572             SNil or Ras-(GDP or GTP)  

573             )  

574           )  

575         ))
```

```

576 );
577 spec complex = 2.(EGF–ErbB1{p1})–scaffoldNoReceptor;
578 plas [ complex ] <->{k6 , kd6} endo [ complex ] |
579
580 plas [ ErbB234 ] <->{k6b , kd6b} endo [ ErbB234 ] |
581
582 spec complex =
583     ATP–ErbB1 or ErbB2{p1}–ErbB4{p1}–GAP–Shc or
584     2.(EGF–ErbB1{p1})–GAP–GDP–Grb2–Ras–Sos ;
585 plas [ complex ] <->{k6 , kd6} endo [ complex ] |
586
587
588 spec complex =
589     ErbB2–Inh{e2} or ErbB4–Inh{e4} or
590     ErbB2{p1}–GAP–(
591         ErbB3{p1}–(Shc or Shc{p1})) or ErbB4{p1}–Shc{p1}
592     );
593 plas [ complex ] <->{k6b , kd6b} endo [ complex ] |
594
595 plas [ ATP–ErbB1–h ] <=>{k6 , kd6} endosomes [ ATP–ErbB1–h ] |
596
597 spec complex =
598     (ErbB1{p1} or ErbB2{p1})–ErbB234{p1} or
599     2.ErbB2{p1}–GAP–(SNil or Shc or Shc{p1});
600 plas [ complex ] <->{k7 , kd7} endo [ complex ]
601 };
602
603 module scaffoldFormation() {
604     spec receptors =
605         ErbB2{p1}–ErbB234{p1} or ErbB1{p1}–ErbB2{p1} or
606         2.(EGF–ErbB1{p1});
607         cyto [GAP] + plas [ receptors ] <->{k8 , kd8} plas [ receptors –GAP] |
608
609     spec receptors = ErbB1{p1}–(ErbB3{p1} or ErbB4{p1});
610     cyto [GAP] + plas [ receptors ] <->{k8b , kd8b} plas [ receptors –GAP] |
611
612     spec complex = 2.(EGF–ErbB1{p1})–GAP;
613     cyto [ Grb2 ] + plas [ complex ] <=>{k16 , kd63} plas [ complex–Grb2 ] |
614     cyto [ Grb2 + Shc{p1} <=>{k16 , kd24} Grb2–Shc{p1}] |
615
616     spec receptor = (ErbB1{p1} or ErbB2{p1})–ErbB234{p1};
617     spec difRateComplex = ErbB2{p1}–ErbB3{p1}–GAP or 2.ErbB2{p1}–GAP;

```

```

618 spec complex = receptor-(GAP-(SNil or Shc{p1})) not difRateComplex ;
619 cyto[Grb2] + plas[complex] <->{k16,kd24} plas[complex-Grb2] |
620 cyto[Grb2] + plas[difRateComplex]
621 <->{k16,kd63} plas[difRateComplex-Grb2] |

622
623
624 spec complex = 2.(EGF-ErbB1{p1})-GAP-Shc{p1};
625 cyto[Grb2] + plas[complex] <=>{k16,kd24} plas[complex-Grb2] |
626 cyto[Grb2] + endo[complex] <=>{k16,kd24} endo[complex-Grb2] |

627
628 cyto[GTP-Ras-iSNil as c + Raf <->{k28,kd28} c-Raf] |
629 cyto[activated-(GTP-Ras as c)-iSNil + Raf{p1}-iSNil
630 <->{k29,kd29} c-Raf-iSNil] |

631
632 // the following reactions take place both in plas and endo, so
633 // abstract into module and invoke twice:
634 module reacs(comp d; spec i) {
635   spec receptors =
636     (ErbB1{p1} or ErbB2{p1})-ErbB234{p1} or 2.(EGF-ErbB1{p1});
637   spec complex = receptors-GAP;
638   // the following seems strange (species appear suddenly on RHS):
639   d[complex] + cyto[Grb2-Shc{p1}-Sos]
640     <->{k32,kd32} d[complex-Grb2-Shc{p1}-Sos] |
641   d[complex] + cyto[Grb2-Sos] <->{k34,kd34} d[complex-Grb2-Sos] |

642
643   d[complex] + cyto[Shc] <->{k22,kd22} d[complex-Shc] |
644   d[complex-Shc] <->{k23,kd23} complex-Shc{p1}] |

645
646   spec complex = receptors-GAP-Grb2-Shc{p1};
647   cyto[Sos] + d[complex] <->{k25,kd25} d[complex-Sos] |

648
649   spec complex = receptors-GAP-Grb2-(Sos-(<b>SNil</b> or Shc{p1}));
650   cyto[GDP-Ras] + d[complex] <->{k21,kd21} d[complex-GTP-Ras] |
651   cyto[GDP-Ras] + d[complex] <->{k18,kd18} d[complex-GDP-Ras] |
652   cyto[activated-GTP-Ras-i] + d[complex]
653     <->{k20,kd20} d[complex-GTP-Ras] |
654   cyto[GTP-Ras-i] + d[complex] <->{k19,kd19} d[complex-GDP-Ras] |

655
656   spec complex = receptors-GAP-Grb2;
657   cyto[Sos] + d[complex] <->{k17,kd17} d[complex-Sos] |

658
659   spec complex = receptors-GAP-Shc{p1};
```

```

660    cyto [ Grb2–Sos ] + d[ complex ] <->{k41 ,kd41} d[ complex–Grb2–Sos ] |
661
662    spec complex = receptors–GAP;
663    spec complex2 = Shc{p1}–(SNil or Grb2);
664    d[ complex ] + cyto [ complex2 ] <->{k37 ,kd37} d[ complex–complex2 ]
665
666    };
667    reacs( plas , SNil) | reacs( endo , i ) |
668
669    cyto [
670      Shc{p1} + Grb2–Sos <=>{k33 ,kd33} Grb2–Shc{p1}–Sos |
671      Sos + Grb2 <=>{k35 ,kd35} Grb2–Sos |
672      Shc{p1} <=>{k36 ,kd36} Shc |
673      Sos + Grb2–Shc{p1} <=>{k40 ,kd40} Grb2–Shc{p1}–Sos
674    ] |
675
676    spec receptor =
677      ErbB1{p1}–(ErbB2{p1} or ErbB4{p1}) or
678      ErbB2{p1}–ErbB4{p1} or 2.(EGF–ErbB1{p1});
679    spec complex = receptor–Gab1{p1}–GAP–Grb2;
680    cyto [ PI3K ] + plas [ complex ] <->{k66 ,kd66} plas [ complex–PI3K ] |
681
682    spec receptor =
683      ErbB2{p1}–(ErbB2{p1} or ErbB3{p1}) or ErbB1{p1}–ErbB3{p1};
684    spec complex = receptor–Gab1{p1}–GAP–Grb2;
685    cyto [ PI3K ] + plas [ complex ] <->{k67 ,kd67} plas [ complex–PI3K ] |
686
687    spec receptor =
688      (ErbB1{p1} or ErbB2{p1})–ErbB234{p1} or 2.(EGF–ErbB1{p1});
689    spec complex = receptor–GAP–Grb2;
690    cyto [ Gab1 ] + plas [ complex ] <->{k105 ,kd105} plas [ complex–Gab1 ] |
691
692    spec complex = receptor–Gab1{p1}–GAP–Grb2;
693    cyto [ Shp2 ] + plas [ complex ] <->{k107 ,kd107} plas [ complex–Shp2 ] |
694    cyto [ Shp2 ] + plas [ complex<Gab1{p1=ff}>]
695      <->{k108 ,kd108} plas [ complex<Gab1{p1}>–Shp2 ] |
696
697    // ErbB4 is left out of the following nondeterministic species
698    // because one reaction has the world compartment in its reactant.
699    // should this really be the case?
700    spec receptor =
701      (ErbB1{p1} or ErbB2{p1})–(ErbB2{p1} or

```

```

702     ErbB3{p1}) or 2.(EGF–ErbB1{p1});  

703     spec scaffold = Gab1{p1}–GAP–Grb2–PI3K;  

704     spec complex = receptor–scaffold;  

705     cyto[GDP–Ras] + plas[complex]  

706         <–>{k112 ,kd112} plas[complex–GDP–Ras] |  

707     cyto[GTP–Ras] + plas[complex]  

708         <–>{k113 ,kd113} plas[complex–GDP–Ras] |  

709     cyto[GDP–Ras] + plas[ErbB1{p1}–ErbB4{p1}–scaffold as c]  

710         <=>{k112 ,kd112} plas[c–GDP–Ras] |  

711     cyto[GDP–Ras] + plas[ErbB2{p1}–ErbB4{p1}–scaffold as c]  

712         <=>{k112 ,kd112} world[c–GDP–Ras] |  

713  

714 // inconsistency in use of Shp2 instead of PI3K, and in the use  

715 // of compartments:  

716     spec scaffold = Gab1{p1}–GAP–Grb2;  

717     cyto[GTP–Ras] + plas[ErbB1{p1}–ErbB4{p1}–scaffold–PI3K as c]  

718         <=>{k113 ,kd113} plas[c–GDP–Ras] |  

719     cyto[GTP–Ras] + plas[(ErbB2{p1}–ErbB4{p1}–scaffold as c)–Shp2]  

720         <=>{k113 ,kd113} world[c–PI3K–GDP–Ras] |  

721  

722     spec receptors = ErbB1{p1}–ErbB234{p1};  

723     endo[receptors] + cyto[GAP] <–>{k8b ,kd8b} endo[receptors–GAP] |  

724  

725     spec receptorsDifRate = ErbB2{p1}–(ErbB2{p1} or ErbB4{p1});  

726     spec receptors =  

727         (ErbB1{p1} or ErbB2{p1})–ErbB234{p1} not receptorsDifRate;  

728     endo[receptors–GAP] + cyto[Grb2]  

729         <–>{k16 ,kd24} endo[receptors–GAP–Grb2] |  

730     endo[receptorsDifRate–GAP] + cyto[Grb2]  

731         <–>{k16 ,kd63} endo[receptorsDifRate–GAP–Grb2] |  

732  

733     endo[2.(EGF–ErbB1{p1})–GAP] + cyto[Grb2]  

734         <=>{k16 ,kd63} endo[2.(EGF–ErbB1{p1})–GAP–Grb2] |  

735  

736     spec receptors = (ErbB1{p1} or ErbB2{p1})–ErbB234{p1};  

737     spec complex = receptors–GAP–Shc{p1};  

738     cyto[Grb2] + endo[complex] <–>{k16 ,kd24} endo[complex–Grb2] |  

739  

740     spec receptors = ErbB2{p1}–ErbB234{p1} or 2.(EGF–ErbB1{p1});  

741     cyto[GAP] + endo[receptors] <–>{k8 ,kd8} endo[receptors–GAP]  

742 };  

743

```

```

744 module MAPKCascade() {
745   // general phosphorylation module.
746   // NOTE: cannot use a common module for single phosphorylation
747   // because of the "i" (which seems inconsistent).
748   module ph(
749     spec k, s:{m1,m2}, i;
750     rate k1, kd1, k2, kd2, k3, kd3, k4, kd4
751   ) {
752     k-i + s <=>{k1,kd1} k-s-i | // why is there no "i" on s here?
753     k-i + s{m1}-i <=>{k2,kd2} k-s{m1}-i |
754     k-i + s{m1}-i <=>{k3,kd3} k-s-i |
755     k-i + s{m1,m2}-i <=>{k4,kd4} k-s{m1}-i
756   };
757
758   // general dephosphorylation module.
759   module dph(
760     spec pt, s:{m1,m2}, i;
761     rate k1, kd1, k2, kd2, k3, kd3, k4, kd4
762   ) {
763     pt + s{m1}-i <=>{k1,kd1} pt-s{m1}-i |
764     // why is there no "i" on s here, and why not
765     // on pt as in the ph module?
766     pt + s <=>{k2,kd2} pt-s{m1}-i |
767     pt + s{m1}-i <=>{k3,kd3} pt-s{m1,m2}-i |
768     pt + s{m1,m2}-i <=>{k4,kd4} pt-s{m1,m2}-i
769   };
770
771   // general phosphorylation/dephosphorylation cycle module.
772   module cycle(spec k, pt, s:{m1,m2}, i;
773     rate k1, kd1, k2, kd2, k3, kd3, k4, kd4,
774     k5, kd5, k6, kd6, k7, kd7, k8, kd8) {
775
776     ph(k, s:{m1,m2}, i, k1, kd1, k2, kd2, k3, kd3, k4, kd4) |
777     dph(pt, s:{m1,m2}, i, k5, kd5, k6, kd6, k7, kd7, k8, kd8)
778   };
779
780   // MEK and ERK cycles.
781   module cycles() {
782     // cycles with and without "i".
783     // NOTE: this is an example of forcing at module invocation time.
784     spec iSNil = force SNil or i;
785     cycle(Raf{p1}, Pase2, MEK:{p1,p2}, iSNil,

```

```

786           k44 , kd52 ,  k44 , kd52 ,  k45 , kd45 ,  k47 , kd47 ,
787           k50 , kd50 ,  k49 , kd49 ,  k49 , kd49 ,  k48 , kd48 )  |
788
789   cycle (MEK{p1 , p2} ,  Pase3 ,  ERK:{p1 ,  p2} ,  iSNil ,
790           k52 , kd44 ,  k52 , kd44 ,  k53 , kd53 ,  k55 , kd55 ,
791           k58 , kd58 ,  k57 , kd57 ,  k57 , kd57 ,  k56 , kd56)
792 };
793
794 module RafDephosphorylation () {
795   Pase1 + iSNil–Raf{p1} <–>{k42 , kd42} iSNil–Pase1–Raf{p1} |
796   Raf + Pase1 <–>{k43 , kd43} iSNil–Pase1–Raf{p1}
797 };
798
799 cyto [ RafDephosphorylation ()  |  cycles ()]
800 };
801
802 module degradation () {
803   spec receptor = 2.(EGF–ErbB1{p1});
804   spec scaffoldWithoutReceptor =
805     GAP–(
806       Shc or Shc{p1} or Grb2–(
807         SNil or Sos–(
808           SNil or Ras–(GDP or GTP)
809           ) or
810           Shc{p1}–(
811             SNil or Sos–(
812               SNil or Ras–(GDP or GTP)
813               )
814             )
815           )
816 );
817   spec complex =
818     receptor–scaffoldWithoutReceptor or
819     ATP–ErbB1 or 2.(EGF–ATP–ErbB1) or
820     2.(EGF–ErbB1{p1})–GAP;
821   endo [ complex ] <–>{k60 , kd60} lysosomes [ degraded–R ]  |
822
823   endo [ ErbB234 ] <–>{k60b , kd60b} lysosomes [ degraded–R ]  |
824
825   spec receptor = ErbB1{p1}–ErbB234{p1};
826   spec scaffoldWithoutReceptor =
827     GAP–Grb2–(

```

```

828     SNil or Shc{p1} or Sos-
829     SNil or Shc{p1} or Ras-
830         GDP-(SNil or Shc{p1}) or GTP-(SNil or Shc{p1})
831         )
832     )
833 );
834 spec complex = receptor-scaffoldWithoutReceptor;
835 endo[complex] <->{k60b ,kd60} lysosomes[degraded-R] |
836
837 spec receptor = 2.(ErbB2{p1});
838 spec scaffoldWithoutReceptor =
839     GAP or
840     GAP-
841         Shc or
842         Shc{p1}-(
843             SNil or
844             Grb2-(SNil or
845                 GDP-Ras-Sos or
846                 GTP-Ras-Sos)
847             ) or
848             Grb2-
849                 SNil or
850                 Sos-(SNil or GDP-Ras or GTP-Ras)
851             )
852 );
853 spec complex = receptor-scaffoldWithoutReceptor or 2.ErbB2;
854 endo[complex] <->{k60b ,kd60} lysosomes[degraded-R] |
855
856 spec receptor = ErbB2{p1}-(ErbB3{p1} or ErbB4{p1});
857 spec scaffoldWithoutReceptor =
858     GAP-
859         Shc or Shc{p1} or Grb2-
860             SNil or
861             Sos-(SNil or Ras-GDP or Ras-GTP) or
862             Shc{p1}-(
863                 SNil or Sos-(SNil or Ras-GDP or Ras-GTP)
864             )
865         )
866 );
867 spec complex = receptor-scaffoldWithoutReceptor;
868 endo[complex] <->{k60c ,kd60} lysosomes[degraded-R] |
869

```

```

870 endosomes[EGF] <=>{k61 ,kd61} lysosomes[ degraded-EGF] |
871
872 spec complex =
873 EGF-ErbB1-ErbB234 or HRG-(ErbB1 or ErbB2)-(ErbB3 or ErbB4) or
874 ErbB2-(ErbB3 or ErbB4);
875 endo[complex] <->{k62b ,kd60b} lysosomes[ degraded-R]
876 };
877
878 module ERKSscaffoldInteraction() {
879   spec complex = 2.(EGF-ErbB1{p1})-GAP-Grb2-Sos-(SNil or Shc{p1});
880   cyto[ERK{p1, p2}] + plas[complex]
881   <->{k64 ,kd64} plas[complex-ERK{p1, p2}] |
882   cyto[ERK{p1, p2}-i] + endo[complex]
883   <->{k64 ,kd64} endo[complex-ERK{p1, p2}] |
884   cyto[ERK{p1, p2}] + plas[complex<Sos{p1}>]
885   <->{k65 ,kd65} plas[complex-ERK{p1, p2}] |
886   cyto[ERK{p1, p2}-i] + endo[complex<Sos{p1}>]
887   <->{k65 ,kd65} endo[complex-ERK{p1, p2}] |
888
889   cyto[ERK{p1, p2}-iSNil + Sos
890   <->{k64 ,kd64} ERK{p1, p2}-iSNil-Sos] |
891   cyto[ERK{p1, p2}-iSNil + Sos{p1}]
892   <->{k65 ,kd65} ERK{p1, p2}-iSNil-Sos] |
893
894   spec receptor =
895   ErbB2{p1}-
896   (ErbB1{p1} or ErbB2{p1}) or (ErbB3{p1} or ErbB4{p1})
897   ) or 2.(EGF-ErbB1{p1});
898   spec complex = receptor-Gab1{p1}-GAP-Grb2;
899   cyto[ERK{p1, p2}-iSNil] + plas[complex]
900   <->{k110 ,kd110} plas[ERK{p1, p2}-complex-iSNil] |
901
902 // NOTE: inconsistent use of species ErbB34 and ErbB22
903 // breaks symmetry. the reactions are strange: some
904 // species suddenly appear on the RHS.
905 spec magic = GAP-Grb2;
906 cyto[ERK{p1, p2}-iSNil] + plas[ErbB2-ErbB34-Gab1{p1}]
907   <->{k111 ,kd111}
908   plas[ErbB2{p1}-ErbB3{p1}-ERK{p1, p2}-Gab1{p1}-iSNil-magic] |
909   cyto[ERK{p1, p2}-iSNil] + plas[2.ErbB22-Gab1{p1}]
910   <->{k111 ,kd111}
911   plas[2.ErbB2{p1}-ERK{p1, p2}-Gab1{p1}-iSNil-magic] |

```

```

912    cyto[ERK{p1, p2}-iSNi1] + cyto[ErbB1-ErbB2-Gab1{p1}]
913      <->{k111, kd111}
914      plas[ErbB1{p1}-ErbB2{p1}-ERK{p1, p2}-Gab1{p1}-iSNi1-magic] |
915    cyto[ERK{p1, p2}-iSNi1] + plas[2.(EGF-ErbB1)-Gab1{p1}]
916      <->{k111, kd111}
917      plas[2.(EGF-ErbB1{p1})-ERK{p1, p2}-Gab1{p1}-iSNi1-magic] |

918
919 spec complex = Gab1{p1}-GAP-Grb2;
920 spec c = ErbB2{p1}-ErbB4{p1}-complex<Gab1{p2}>;
921 cyto[ERK{p1, p2}-iSNi1] + plas[c]
922   <->{k111, kd111}
923   plas[c<Gab1{p2=ff}>-ERK{p1, p2}-iSNi1] |
924 cyto[ERK{p1, p2}-iSNi1] + world[ErbB1{p1}-ErbB4{p1}-complex as c]
925   <->{k111, kd111}
926   endosomes[c-ERK{p1, p2}-iSNi1] |
927 cyto[ERK{p1, p2}] + world[ErbB1{p1}-ErbB3{p1}-complex as c]
928   <=>{k111, kd111} plas[c-ERK{p1, p2}] |
929 cyto[ERK{p1, p2}-i] + world[ErbB1{p1}-ErbB3{p1}-complex as c]
930   <=>{k111, kd111} endosomes[c-ERK{p1, p2}-i] |
931 cyto[ERK{p1, p2}] + plas[ErbB1{p1}-ErbB3{p1}-complex as c]
932   <=>{k110, kd110} plas[c-ERK{p1, p2}] |
933 cyto[ERK{p1, p2}-i] + plas[ErbB1{p1}-ErbB3{p1}-complex as c]
934   <=>{k110, kd110} endosomes[c-ERK{p1, p2}-i] |
935 cyto[ERK{p1, p2}-iSNi1] + plas[ErbB1{p1}-ErbB4{p1}-complex as c]
936   <->{k110, kd110} endosomes[c-ERK{p1, p2}-iSNi1]
937 };
938
939 module PIP() {
940   // NOTE: strange that PIP2 is always used on product side,
941   // even when the actual parameter PIP is bound to PIP3.
942   module PIP2BuildUp(spec PIP; rate k1, kd1, k2, kd2, k3, kd3) {
943     spec receptor = (ErbB1{p1}-ErbB234{p1}) or 2.(EGF-ErbB1{p1});
944     spec complex = receptor-Gab1{p1}-GAP-Grb2-PI3K;
945     cyto[PIP] + plas[complex] <->{k1, kd1} plas[complex-PIP2] |

946
947   // the following has a different rate from the above:
948   spec complex = 2.ErbB2{p1}-Gab1{p1}-GAP-Grb2-PI3K;
949   cyto[PIP] + plas[complex] <=>{k2, kd2} plas[complex-PIP2] |

950
951   // now to the build-up:
952   spec complex = ErbB2{p1}-ErbB3{p1}-Gab1{p1}-GAP-Grb2-PI3K;
953   cyto[PIP] + plas[complex] <=>{k3, kd3} plas[complex-PIP2] |

```

```

954     cyto[PIP] + plas[complex-PIP2] <=>{k3,kd3} plas[complex-PIP22] |
955     cyto[PIP] + plas[complex-PIP22] <=>{k3,kd3} plas[complex-PIP23] |
956     cyto[PIP] + plas[complex-PIP23] <=>{k3,kd3} plas[complex-PIP24] |
957     cyto[PIP] + plas[complex-PIP24] <=>{k3,kd3} plas[complex-PIP25] |
958     cyto[PIP] + plas[complex-PIP25] <=>{k3,kd3} plas[complex-PIP26]
959   };
960
961 PIP2BuildUp(PIP3, k68, kd68, k68, kd68, k68, kd68b) |
962 PIP2BuildUp(PIP2, k106b, kd106b, k106, kd106, k106, kd106) |
963 // NOTE: the next reaction does not seem to fit in anywhere.
964 // it goes to cyto rather than plas, is this correct?
965 cyto[PIP2] + plas[ErbB2{p1}-ErbB4{p1}-Gab1{p1}-GAP-Grb2-PI3K as c]
966   <=>{k106,kd106} cyto[c-PIP2]
967 };
968
969 module PIPAktCascade() {
970   // module for single-site phosphorylation and PIP binding:
971   module ph1(spec s:{m}; rate k1, kd1, k2, kd2, k3, kd3) {
972     PIP3 + s <=>{k1,kd1} PIP3-s |
973     PDK1 + PIP3-s <=>{k2,kd2} PDK1-PIP3-s |
974     PDK1-PIP3 + s{m} <=>{k3,kd3} PDK1-PIP3-s
975   };
976
977   // module for single-site dephosphorylation:
978   module dph1(spec s:{m}; rate k1, kd1, k2, kd2) {
979     Pase4 + s{m} <=>{k1,kd1} Pase4-s{m} |
980     Pase4 + s <=>{k2,kd2} Pase4-s{m}
981   };
982
983   // module for a single phosphorylation/dephosphorylation cycle:
984   module cycle(
985     spec s:{m};
986     rate k1, kd1, k2, kd2, k3, kd3, k4, kd4, k5, kd5
987   ) {
988     ph1(s:{m}, k1, kd1, k2, kd2, k3, kd3) |
989     dph1(s:{m}, k4, kd4, k5, kd5)
990   };
991
992   // module for invoking cycles:
993   module PIPCycle() {
994     // phosphorylation:
995     Shp + PIP2 <=>{k104,kd104} PIP3-Shp |

```

```

996    PIP3 + Shp <=>{k109 ,kd109} PIP3-Shp | 
997
998 // dephosphorylation:
999 PIP3 + PTEN <=>{k109 ,kd109} PIP3-PTEN |
1000 PTEN + PIP2 <=>{k104 ,kd104} PIP3-PTEN
1001 };
1002
1003 cyto [
1004   PIPCycle() |
1005   cycle (
1006     AKT:{p1},
1007     k69 ,kd69 , k70 ,kd70 , k71 ,kd71 , k73 ,kd73 , k75 ,kd75
1008   ) |
1009   cycle (
1010     AKT{p1}:{p2},
1011     k69 ,kd69 , k70 ,kd70 , k72 ,kd72 , k74 ,kd74 , k75 ,kd75
1012   ) |
1013   PDK1 + PIP3 <=>{k76 ,kd76} PDK1-PIP3
1014 ]
1015 };
1016
1017 module RafAktInteraction() {
1018   cyto [AKT{p1,p2} + iSN1-Raf{p1}
1019   <->{k114 ,kd114} AKT{p1 ,p2}-iSN1-Raf{p1}-Ser] |
1020   plas [Raf{p1}-Ser] + cyto [AKT{p1 ,p2}]
1021   <->{k115 ,kd115} cyto [AKT{p1 ,p2}-Raf{p1}-Ser-iSN1]
1022 };
1023
1024 module phosphataseBinding() {
1025   spec complex =
1026     (ErbB1{p1} or ErbB2{p1})-ErbB234{p1} or 2.(EGF-ErbB1{p1});
1027   cyto [Pase-RTK] + endo [complex]
1028   <->{k94b ,kd94} endo [complex-Pase-RTK] |
1029
1030   spec e1a = ErbB1:{p1} :: a{p1};
1031   spec e2a = ErbB2:{p1} :: a{p1};
1032   spec e234b = ErbB234:{p1} :: b{p1};
1033   spec complex1 = e1a-e234b;
1034   spec complex2 = e2a-e234b;
1035   cyto [Pase-RTK] + endo [EGF-complex1]
1036   <->{k95 ,kd95} endo [complex1<a{p1}><b{p1}>-Pase-RTK] |
1037   cyto [Pase-RTK] + endo [complex2]

```

```

1038    <->{k95 ,kd95}  endo [ complex2<a{p1}><b{p1}>–Pase–RTK]  |
1039    cyto [ Pase–RTK] + endo [2.(EGF–ATP–ErbB1)] |
1040    <=>{k95 ,kd95}  endo [2.(EGF–ErbB1{p1})–Pase–RTK]  |
1041
1042    cyto [ Pase3 <=>{k116 ,kd116}  deg–MKP]  |
1043
1044    spec complex =
1045        Gab1{p1} –(2.(EGF–ErbB1) or 2.ErbB22 or ErbB2–ErbB34);
1046        cyto [ Pase9t] + plas [ complex] <->{k117 ,kd117}  plas [ complex–Pase9t]  |
1047
1048    // NOTE: cannot generalise because of differences in compartments.
1049    cyto [ Pase9t] + world [ErbB1{p1}–ErbB3{p1}–Gab1{p1}–GAP–Grb2 as c]
1050    <=>{k117 ,kd117}  endo [c–Pase9t]  |
1051    cyto [ Pase9t] + world [ErbB1{p1}–ErbB4{p1}–Gab1{p1}–GAP–Grb2 as c]
1052    <=>{k117 ,kd117}  plas [c–Pase9t]  |
1053    cyto [ Pase9t] + cyto [ErbB1–ErbB2–Gab1{p1}] as c
1054    <=>{k117 ,kd117}  plas [c–Pase9t]  |
1055    cyto [ Pase9t] + plas [ErbB2{p1}–ErbB4{p1}–Gab1{p1, p2}–GAP–Grb2 as c]
1056    <=>{k117 ,kd117}  world [c–Pase9t]  |
1057
1058    // the following has inconsistent use of e.g. ErbB22 and ErbB34,
1059    // and product species and compartments vary inconsistently.
1060    cyto [ Pase9t] + plas [2.ErbB2{p1}–Gab1{p1}–GAP–Grb2]
1061    <=>{k118 ,kd118}  plas [2.ErbB22–Gab1{p1}–Pase9t]  |
1062    cyto [ Pase9t] + plas [2.(EGF–ErbB1{p1})–Gab1{p1}–GAP–Grb2]
1063    <=>{k118 ,kd118}  plas [2.(EGF–ErbB1)–Gab1{p1}–Pase9t]  |
1064    cyto [ Pase9t] + plas [ErbB2{p1}–ErbB3{p1}–Gab1{p1}–GAP–Grb2]
1065    <=>{k118 ,kd118}  plas [ErbB2–ErbB34–Gab1{p1}–Pase9t]  |
1066    cyto [ Pase9t] + plas [ErbB1{p1}–ErbB2{p1}–Gab1{p1}–GAP–Grb2]
1067    <=>{k118 ,kd118}  plas [ErbB1–ErbB2–Gab1{p1}–Pase9t]  |
1068    cyto [ Pase9t] + plas [ErbB1{p1}–ErbB3{p1}–Gab1{p1}–GAP–Grb2 as c]
1069    <=>{k118 ,kd118}  endo [c–Pase9t]  |
1070    cyto [ Pase9t] + plas [ErbB2{p1}–ErbB4{p1}–Gab1{p1}–GAP–Grb2 as c]
1071    <=>{k118 ,kd118}  world [c<Gab1{p2}>–Pase9t]  |
1072    cyto [ Pase9t] + plas [ErbB1{p1}–ErbB4{p1}–Gab1{p1}–GAP–Grb2 as c]
1073    <=>{k118 ,kd118}  plas [c–Pase9t]
1074  };
1075
1076    // module invocations:
1077    receptorActivation()      |
1078    receptorLigandBinding()  |
1079    receptorDimerisation()  |

```

```
1080 receptorInhibition()      |
1081 scaffoldFormation()       |
1082 phosphataseBinding()     |
1083 cPPTagging()             |
1084 endocytosis()            |
1085 MAPKCascade()            |
1086 ERKScaffoldInteraction() |
1087 RafAktInteraction()      |
1088 PIP()                    |
1089 PIPAktCascade()          |
1090 degradation()
```


Appendix C

Proofs

C.1 Proofs for Compartment Value Lists

Proposition 5.1.1. By induction in $|\{\underline{v}_{c_i}\}|$. In the following we additionally use a and b to range over compartment values.

- **Basis** ($\{\underline{v}_{c_i}\} = \emptyset$). Holds vacuously.

- **Step** ($\{\underline{v}_{c_i}\} \cup \{\underline{v}'_c\}$).

Acyclic: by the induction hypothesis, $G\{\underline{v}_{c_i}\}$ is acyclic. Also $G\{\underline{v}'_c\}$ is acyclic, for otherwise \underline{v}'_c would take the form $\underline{v}'_{c_1} a \underline{v}'_{c_2} a \underline{v}'_{c_3}$, and it follows from well-typedness that the compartment value a must include itself as an ancestor; this is impossible since compartment values are finite. Suppose towards a contradiction that there is a cycle in $G(\{\underline{v}_{c_i}\} \cup \{\underline{v}'_c\})$. This can then only arise from a branch in $G\{\underline{v}_{c_i}\}$ of the form $\underline{v}_{c_1} a \underline{v}_{c_2} b \underline{v}_{c_3}$ and \underline{v}'_c of the form $\underline{v}'_{c_1} b \underline{v}'_{c_2} a \underline{v}'_{c_3}$, both of which are well-typed. This means that the compartment value a must include b as an ancestor, and b in turn must include a as an ancestor. Hence a must include itself as an ancestor. But this is impossible since compartment values are finite.

Max one parent: by the induction hypothesis, each node in $G\{\underline{v}_{c_i}\}$ has at most one parent. Also each node in $G\{\underline{v}'_c\}$ has at most one parent, for otherwise the graph would contain a cycle. Suppose towards a contradiction that there is some node a in $G(\{\underline{v}_{c_i}\} \cup \{\underline{v}'_c\})$ with two parents. This can only arise from a branch in $G\{\underline{v}_{c_i}\}$ of the form $\underline{v}_{c_1} b a \underline{v}_{c_2}$ and \underline{v}'_c of the form $\underline{v}'_{c_1} c a \underline{v}'_{c_2}$ with $b \neq c$. But this is impossible since both lists are well-typed and a can contain only a single parent.

□

C.2 Proofs for Petri Net Flows

C.2.1 Duality

Theorem 6. As for place sharing in Section 7.2, let us consider the structure of the flow matrix W arising from the transition-based composition $PN_1 \mid_t PN_2$ of nets with flow matrices W_1 and W_2 . We make similar assumptions about the ordering of transitions as for places under place-based composition. Then W_1 , W_2 and W can be partitioned as follows where, for $i \in \{1, 2\}$, W_i^t consists of the *columns* from W_i which represent shared *transitions* $T_1 \cap T_2$, and W_i^- are the remaining columns for non-shared transitions:

$$W_1 = [W_1^- \ W_1^t], \quad W_2 = [W_2^t \ W_2^-], \quad W = \begin{bmatrix} W_1^- & W_1^t & 0 \\ 0 & W_2^t & W_2^- \end{bmatrix}$$

We then reason as follows:

$$\begin{aligned} PF(PN_1 \mid_t PN_2) &= PF \begin{bmatrix} W_1^- & W_1^t & 0 \\ 0 & W_2^t & W_2^- \end{bmatrix} \\ &= TF \begin{bmatrix} W_1^- & W_1^t & 0 \\ 0 & W_2^t & W_2^- \end{bmatrix}^T \\ &= TF \begin{bmatrix} W_1^{-T} & 0 \\ W_1^{tT} & W_2^{tT} \\ 0 & W_2^{-T} \end{bmatrix} = TF(PN_1^D \mid_s PN_2^D) \end{aligned}$$

Symmetric reasoning can be used for T-flows under transition sharing. \square

C.2.2 Modular T-Flows

Lemma 1 (soundness part 1). Take any $x \in Z$. Per definition of Z , $x = X\alpha$ for some $\alpha \in MTF(C)$. We now reason as follows, relying on the fact that matrix multiplication is associative:

$$0 = C\alpha = (W^s X)\alpha = W^s(X\alpha) = W^s x$$

So $x \in TF(W^s)$. Also $x \in TF(W^-)$ because x is a linear combination of columns of X which are minimal flows in W^- ; any such combination is itself a flow. Together these give that $Wx = 0$, i.e. $x \in TF(W) = TF(PN_1 \mid_s PN_2)$, which completes the proof of 1). 2) follows immediately from 1) and the fact that division of a flow by gcd is also a flow. \square

Lemma 2 (completeness) for standard minimality. Take any $x \in MTF(PN_1 \mid_s PN_2)$. Then $Wx = 0$, so also $W^s x = 0$ and $W^- x = 0$. Hence $x \in TF(W^s)$ and $x \in TF(W^-)$. Observe that X consists exactly of the minimal T-flows of W^- . Therefore, by Theorem 2 there are $\alpha \in \mathbb{N}^{|col(X)|^T}$ and $a \in \mathbb{N}$ s.t. $x = \frac{1}{a}X\alpha$, i.e. $xa = X\alpha$. There may generally be multiple such α , so pick one which is canonical and has *minimal decomposition-support* in the sense that its support does not contain the support of any other choices. Such a canonical choice *is* indeed possible because it is always the case that $gcd(\alpha)$ divides a . To see this, let $c = gcd(\alpha)$; then there is a canonical α' s.t. $ax = Xc\alpha' = cX\alpha'$. Also $\frac{a}{d}x = \frac{c}{d}X\alpha'$ where $d = gcd(a, c)$. Since x has natural number entries, $\frac{a}{d}$ divides all entries in $\frac{c}{d}X\alpha'$. It follows from Euclid's lemma and $gcd(\frac{a}{d}, \frac{c}{d}) = 1$ that $\frac{a}{d}$ divides all entries in $X\alpha'$. Canonicity of x then forces $c = d$, and hence $d = gcd(\alpha)$ divides a as claimed.

We now show that α is a T-flow of C , i.e. that $C\alpha = 0$. The following steps rely on the fact that matrix multiplication is associative:

$$C\alpha = (W^s X)\alpha = W^s(X\alpha) = W^s(xa) = (W^s x)a = 0a = 0$$

Next we show that α is a *minimal* T-flow of C . It is canonical per assumption. To get that α has minimal support, we show that any T-flow α' of C with $sup(\alpha') \subsetneq sup(\alpha)$ also generates x , contrary to α being a choice with a minimal decomposition-support for which this holds. Note here the subtle distinction between minimality of α wrt. decomposition of x and wrt. flows of C ; the former holds per assumption, and we will now prove the latter.

So, we have $sup(\alpha') \subsetneq sup(\alpha)$ and $C\alpha' = 0$. Then $0 = C\alpha' = (W^s X)\alpha' = W^s(X\alpha')$, so $x' = X\alpha'$ is a T-flow of W^s . Any linear combination of T-flows is also a T-flow, so x' is also a T-flow of W^- . Together these give $x' \in TF(W^-)$. Now since $sup(\alpha') \subsetneq sup(\alpha)$ it must also hold that $sup(x') = sup(X\alpha') \subseteq sup(X\alpha) = sup(x)$. Since x has minimal-support, it must be the case that $sup(x') = sup(x)$. By Theorem 3, either $x = nx'$ or $x' = nx$ for some $n \in \mathbb{N}$. But x is canonical, so $x' = nx$ i.e. $x = \frac{1}{n}x' = \frac{1}{n}X\alpha'$. This contradicts our original choice of α to be a minimal-support decomposition of x .

We conclude that $\alpha \in MTF(C)$ and hence $xa = X\alpha \in Z$. Per assumption x is minimal, so there is no other minimal flow $x'' \in MTF(PN_1 \mid_s PN_2) \supset Z$ (the inclusion is By Lemma 1) with $sup(x'') \subsetneq sup(x)$. Hence $x = \frac{xa}{a} \in min(Z) = MTF^{\text{Par}}(X_1, X_2, W^s)$. \square

Lemma 2 (completeness) for weak minimality. Take any $x \in M_w TF(PN_1 \mid_s PN_2)$. Then $Wx = 0$, so also $W^s x = 0$ and $W^- x = 0$. Hence $x \in TF(W^s)$ and $x \in TF(W^-)$. Observe

that X consists exactly of the minimal T-flows of W^- . Therefore, by Theorem 5 there is an $\alpha \in \mathbb{N}^{|col(X)|^T}$ s.t. $x = X\alpha$.

We first show that α is a T-flow of C , i.e. that $C\alpha = 0$. The following steps rely on the fact that matrix multiplication is associative:

$$C\alpha = (W^s X)\alpha = W^s(X\alpha) = W^s x = 0$$

Next we show that α is a *minimal* T-flow of C , so suppose towards a contradiction that there are $\alpha', \alpha'' \in TF(C)$ s.t. $\alpha = \alpha' + \alpha''$. Then $X\alpha' \in TF(W)$, for $W^s(X\alpha') = (W^s X)\alpha' = C\alpha' = 0$ and also $W^-(X\alpha') = 0$ (any linear combination of flows is again a flow). Similar reasoning shows that $X\alpha'' \in TF(W)$. But $x = X\alpha = X(\alpha' + \alpha'') = X\alpha' + X\alpha''$, contradicting minimality of x .

We conclude that $\alpha \in M_w TF(C)$ and hence $x = X\alpha \in Z$. Per assumption x is minimal, so there are no other flows $x_1, \dots, x_k \in TF(PN_1 \mid_s PN_2) \supset Z$ (the inclusion is By Lemma 1) and $\gamma_1, \dots, \gamma_k \in \mathbb{N}$ s.t. $x = \gamma_1 x_1 + \dots + \gamma_k x_k$. Hence also $x \in min(Z) = MTF^{\text{Par}}(X_1, X_2, W^s)$. \square

Lemma 3 (soundness part 2) for standard minimality.

Take any $x \in MTF^{\text{Par}}(X_1, X_2, W^-) = min(Z)$. By Lemma 1, $x \in TF(PN_1 \mid_s PN_2)$. x is canonical per definition of the minimisation function. Suppose towards a contradiction that there is some $x' \in MTF(PN_1 \mid_s PN_2)$ with $sup(x') \subsetneq sup(x)$. Then by Lemma 2 also $x' \in min(Z)$, so $nx' \in Z$ for some $n \in \mathbb{N}$. But this contradicts the definition of minimisation since $sup(nx') \subsetneq sup(x)$. \square

Lemma 3 (soundness part 2) for weak minimality.

Take any $x \in MTF^{\text{Par}}(X_1, X_2, W^-) = min(Z)$. By Lemma 1, $x \in TF(PN_1 \mid_s PN_2)$. Suppose towards a contradiction that there are some $x_1, \dots, x_k \in MTF(PN_1 \mid_s PN_2)$ and $a_1, \dots, a_k \in \mathbb{N}$ with $x = a_1 x_1 + \dots + a_k x_k$. Then by Lemma 2 also $x_1, \dots, x_k \in min_w(Z) \subsetneq Z$, contradicting the definition of the minimisation function. \square

Theorem 8. The proofs for both cases relies on the fact that linear independence implies unique decomposition.

1. **For standard minimality:** Suppose towards a contradiction that $z \in Z$ does not have minimal support. By Lemma 1, $z \in TF(PN_1 \mid_s PN_2)$. By Theorem 7, $MTF(PN_1 \mid_s PN_2) = min(Z)$, so by Theorem 2, $z = \frac{1}{a}(a_1 z_1 + \dots + a_k z_k)$, $k > 1$ for some distinct $z_i \in min(Z)$ and $a, a_i \in \mathbb{N}$. Per definition of minimisation there

are $b_i \in \mathbb{N}$ s.t. $z_i b_i \in Z$. Per definition of Z there are $\beta_i \in MTF(C)$ s.t. $z_i b_i = X\beta_i$, i.e. $z_i = \frac{1}{b_i} X\beta_i$. Hence

$$\begin{aligned} z &= \frac{1}{a}(a_1 z_1 + \cdots + a_k z_k) \\ &= \frac{1}{a}\left(\frac{a_1}{b_1} X\beta_1 + \cdots + \frac{a_k}{b_k} X\beta_k\right) \\ &= X \frac{1}{a}\left(\frac{a_1}{b_k} \beta_1 + \cdots + \frac{a_k}{b_k} \beta_k\right) \end{aligned}$$

But since also $z = X\alpha$ for some $\alpha \in MTF(C)$ it follows by unique decomposition that

$$\alpha = \frac{1}{a}\left(\frac{a_1}{b_1} \beta_1 + \cdots + \frac{a_k}{b_k} \beta_k\right)$$

Clearly $sup(\beta_i) \subseteq sup(\alpha)$ for all β_i . All β_i have distinct supports, so $sup(\beta_i) \subsetneq sup(\alpha)$ for at least one of the β_i . This contradicts the minimality of α .

2. **For weak minimality:** Support towards a contradiction that there are $x, x', x'' \in Z$ s.t. $x = x' + x''$. Per definition of Z there are $\alpha, \alpha', \alpha'' \in MTF(C)$ s.t. $x = X\alpha$, $x' = X\alpha'$ and $x'' = X\alpha''$. Hence

$$X\alpha = X\alpha' + X\alpha'' = X(\alpha' + \alpha'')$$

By unique decomposition, $\alpha = \alpha' + \alpha''$. But this contradicts that α is minimal in C . □

C.2.3 Modular P-Flows

Lemma 4. Per definition of P-flows, $xW_1 = 0$ and $yW_2 = 0$. Per definition of flow join, also $(x \frown y)W_1^+ = 0$ and $(x \frown y)W_2^+ = 0$ which per definition of matrix left-multiplication gives that $(x \frown y)W = 0$. Hence $(x \frown y) \in PF(PN_1 \mid_s PN_2)$. □

Lemma 5. x and y are the restrictions of z to S_{PN_1} and S_{PN_2} respectively. Since $zW = 0$ also $zW_1^+ = 0$ and $zW_2^+ = 0$. It follows immediately that $xW_1 = 0$ and $yW_2 = 0$, i.e. $x \in PF(PN_1)$ and $y \in PF(PN_2)$. □

Lemma 6 (soundness part 1). Take any $z \in Z$. Per definition of Z there is $(\alpha \beta) \in MPF(C)$ s.t. $z = \alpha X \frown \beta Y$, and this join is clearly defined. Any linear combination of flows is a flow, so αX and βY are P-flows of PN_1 and PN_2 , respectively. By Lemma 4, $z \in PF(PN_1 \mid_s PN_2)$ which proves 1). Item 2) follows from 1) and the fact that any flow divided by a common divisor is also a flow. \square

Lemma 7 (completeness) for standard minimality. Take any $z \in MPF(PN_1 \mid_s PN_2)$. By Lemma 5 there are restrictions $x \in PF(PN_1) \cup \{0\}$ and $y \in PF(PN_2) \cup \{0\}$ of z s.t. $z = x \frown y$. **Claim:** there are $(\alpha \beta) \in MPF(C)$ and $d \in \mathbb{N}$ such that

$$dx = \alpha X \text{ and } dy = \beta Y$$

Then $dz = dx \frown dy = \alpha X \frown \beta Y \in Z$. Per assumption z is minimal so there is no other flow $z' \in PF(PN_1 \mid_s PN_2) \supset Z$ (Lemma 6) s.t. $sup(z') \subsetneq sup(z) = sup(dz)$. Hence $z = \frac{dz}{d} \in min(Z) = MPF(X_1, X_2, W^-)$, so we are done.

Proof of claim: By Theorem 2 there are $a, b \in \mathbb{N}$, $\alpha'' \in \mathbb{N}^{|row(X)|}$ and $\beta'' \in \mathbb{N}^{|row(Y)|}$ with $a \neq 0, b \neq 0$ and either α'' or $\beta'' \neq 0$ s.t.

$$\begin{aligned} ax &= \alpha'' X \quad \text{and} \quad by = \beta'' Y \quad \Leftrightarrow \\ abx &= \alpha'' bX \quad \text{and} \quad aby = \beta'' aY \end{aligned}$$

There may generally be many such $(\alpha'' \beta'')$, so pick one which has *minimal decomposition-support* in the sense that its support does not contain the support of any other possible choices.

Now let $c = gcd(a, b)$, $d = \frac{ab}{c}$, $\alpha = \alpha'' \frac{b}{c}$ and $\beta = \beta'' \frac{a}{c}$. Continuing with the equations from above we then get

$$dx = \alpha X \quad \text{and} \quad dy = \beta Y$$

We know that x and y are consistent, i.e. $x^s = y^s$ where x^s and y^s are the restrictions of x and y to the shared places $SPN_1 \cap SPN_2$. Hence also $dx^s = dy^s$. So

$$\begin{aligned} \alpha X^s &= dx^s = dy^s = \beta Y^s \quad \Leftrightarrow \\ \alpha X^s - \beta Y^s &= 0 \quad \Leftrightarrow \\ (\alpha \beta)C &= 0 \end{aligned}$$

It follows that $(\alpha \beta) \in PF(C)$. We may assume that $(\alpha \beta)$ is canonical, for if it is not, it is always possible to divide through by $gcd(\alpha \beta)$ since this always divides d . To see

why this is the case, let $c = \gcd(\alpha\beta)$. Then there are α' and β' s.t. $dx = c\alpha'X$ and $dy = c\beta'Y$. Now let $e = \gcd(c, d)$ and write $\frac{d}{e}x = \frac{c}{e}\alpha'X$ and $\frac{d}{e}y = \frac{c}{e}\beta'Y$. Since x and y have entries in \mathbb{N} , $\frac{d}{e}$ divides all entries in both $\frac{c}{e}\alpha'X$ and $\frac{c}{e}\beta'Y$. From $e = \gcd(c, d)$ and Euclid's lemma we get that $\frac{d}{e}$ divides all entries in $\alpha'X$ and $\beta'Y$. From $\frac{d}{e}x = \frac{c}{e}\alpha'X$ and $\frac{d}{e}y = \frac{c}{e}\beta'Y$ we furthermore get that $\frac{c}{e}$ divides all entries in both x and y , and hence also in $x \frown y = z$. Canonicity of z then forces $c = e$, so c divides d as claimed.

To see that $(\alpha\beta)$ has minimal support in C , suppose towards a contradiction that there is $(\alpha'\beta') \in PF(C)$ with $sup(\alpha'\beta') \subsetneq sup(\alpha\beta) = sup(\alpha''\beta'')$. From the definition of C it follows that $x' = \alpha'X$ and $y' = \beta'Y$ are consistent, i.e. $x'^s = \alpha'X^s = \beta'Y^s = y'^s$. They are also place flows of PN_1 and PN_2 respectively. Lemma 4 then gives that $z' = x' \frown y' \in PF(PN_1 \mid_s PN_2)$. We know that $sup(z') \subseteq sup(z)$, but we cannot have $sup(z') \subsetneq sup(z)$ since z is minimal. Hence $sup(z') = sup(z)$. By Theorem 3, there is some $n \in \mathbb{N}$ s.t.

$$nz = z' = x' \frown y' = \alpha'X \frown \beta'Y$$

But we also know that $nz = n(x \frown y) = nx \frown ny$. Hence

$$\begin{aligned} nx &= \alpha'X \text{ and} \\ ny &= \beta'Y \end{aligned}$$

Per assumption either $sup(\alpha') \subsetneq sup(\alpha'')$ or $sup(\beta') \subsetneq sup(\beta'')$. This contradicts our original choice of α'' or β'' to have minimal decomposition-support. \square

Lemma 7 (completeness) for weak minimality. Take any $z \in M_wPF(PN_1 \mid_s PN_2)$. By Lemma 5 there are restrictions $x \in PF(PN_1)$ and $y \in PF(PN_2)$ of z s.t. $z = x \frown y$. By Theorem 5 there are $\alpha \in \mathbb{N}^{|row(X)|}$ and $\beta \in \mathbb{N}^{|rowY|}$ s.t. $x = \alpha X$ and $y = \beta Y$.

We know that x and y are consistent, i.e. $x^s = y^s$ where x^s and y^s are the restrictions of x and y to the shared places $S_{PN_1} \cap S_{PN_2}$. So

$$\begin{aligned} \alpha X^s &= x^s = y^s = \beta Y^s \Leftrightarrow \\ \alpha X^s - \beta Y^s &= 0 \\ (\alpha\beta)C &= 0 \end{aligned}$$

It follows that $(\alpha\beta) \in PF(C)$. **Claim:** $(\alpha\beta)$ is minimal in C . Then $x \frown y \in Z$. Per assumption z is minimal so there are no $z_1, \dots, z_k \in PF(PN_1 \mid_s PN_2) \supset Z$ (the inclusion is by Lemma 6) and $\gamma_1, \dots, \gamma_k \in \mathbb{N}$ s.t. $z = \gamma_1 z_1 + \dots + \gamma_k z_k$. Hence $z \in min_w(Z)$ and

we are done.

Proof of claim: Suppose towards a contradiction that $(\alpha\beta) = (\alpha'\beta') + (\alpha''\beta'')$ for some $(\alpha'\beta'), (\alpha'',\beta'') \in PF(C)$. We then reason as follows:

$$\begin{aligned} z &= x \frown y \\ &= \alpha X \frown \beta Y \\ &= (\alpha' + \alpha'')X \frown (\beta' + \beta'')Y \\ &= (\alpha'X + \alpha''X) \frown (\beta'Y + \beta''Y) \\ &= (\alpha'X \frown \beta'Y) + (\alpha''X \frown \beta''Y) \end{aligned}$$

The last equality holds because $\alpha'X$ and $\beta'Y$ are consistent per definition of C , and so are $\alpha''X$ and $\beta''Y$. Now Lemma 4 gives us that $z' = \alpha'X \frown \beta'Y \in PF(PN_1 \mid_s PN_2)$ and $z'' = \alpha''X \frown \beta''Y \in PF(PN_1 \mid_s PN_2)$. But $z = z' + z''$, contradicting minimality of z . \square

Lemma 8 (soundness part 2) for standard minimality.

Take any $x \in MPF^{\text{Par}}(X_1, X_2) = \min(Z)$. By Lemma 6, $x \in PF(PN_1 \mid_s PN_2)$. x is canonical per definition of the minimisation function. Suppose towards a contradiction that there is some $x' \in MPF(PN_1 \mid_s PN_2)$ with $\sup(x') \subsetneq \sup(x)$. Then by Lemma 7 also $x' \in \min(Z)$, so $nx' \in Z$ for some $n \in \mathbb{N}$. But this contradicts the definition of minimisation since $\sup(nx') \subsetneq \sup(x)$. \square

Lemma 8 (soundness part 2) for weak minimality.

Take any $x \in MPF^{\text{Par}}(X_1, X_2) = \min(Z)$. By Lemma 6, $x \in PF(PN_1 \mid_s PN_2)$. Suppose towards a contradiction that there are some $x_1, \dots, x_k \in MPF(PN_1 \mid_s PN_2)$ and $a_1, \dots, a_k \in \mathbb{N}$ with $x = a_1x_1 + \dots + a_kx_k$. Then by Lemma 7 also $x_1, \dots, x_k \in \min_w(Z) \subsetneq Z$, contradicting the definition of the minimisation function. \square

Theorem 10. The proofs for both cases relies on the fact that linear independence implies unique decomposition.

1. **For standard minimality:** Suppose towards a contradiction that $z \in Z$ does not have minimal support. By Lemma 1, $z \in PF(PN_1 \mid PN_2)$. By Theorem 7, $MPF(PN_1 \mid PN_2) = \min(Z)$, so by Theorem 2, $z = \frac{1}{a}(a_1z_1 + \dots + a_kz_k)$, $k > 1$ for some distinct $z_i \in \min(Z)$ and $a, a_i \in \mathbb{N}$. Per definition of minimisation there

are $b_i \in \mathbb{N}$ s.t. $z_i b_i \in Z$. Per definition of Z there are $(\alpha_i \beta_i) \in MPF(C)$ s.t. $z_i b_i = \alpha_i X \frown \beta_i Y$, i.e. $z_i = \frac{1}{b_i}(\alpha_i X \frown \beta_i Y)$. Hence

$$\begin{aligned} z &= \frac{1}{a}(a_1 z_1 + \dots + a_k z_k) \\ &= \frac{1}{a}\left(\frac{a_1}{b_1}(\alpha_1 X \frown \beta_1 Y) + \dots + \frac{a_k}{b_k}(\alpha_k X \frown \beta_k Y)\right) \\ &= \frac{1}{a}\left(\frac{a_1}{b_1}\alpha_1 X + \dots + \frac{a_k}{b_k}\alpha_k X \frown \frac{a_1}{b_1}\beta_1 Y + \dots + \frac{a_k}{b_k}\beta_k Y\right) \\ &= \frac{1}{a}\left(\left(\frac{a_1}{b_1}\alpha_1 + \dots + \frac{a_k}{b_k}\alpha_k\right)X \frown \left(\frac{a_1}{b_1}\beta_1 + \dots + \frac{a_k}{b_k}\beta_k\right)Y\right) \end{aligned}$$

But since also $z = \alpha X \frown \beta Y$ for some $(\alpha \beta) \in MPF(C)$ it follows by unique decomposition that

$$\begin{aligned} \alpha &= \frac{1}{a}\left(\frac{a_1}{b_1}\alpha_1 + \dots + \frac{a_k}{b_k}\alpha_k\right) \\ \beta &= \frac{1}{a}\left(\frac{a_1}{b_1}\beta_1 + \dots + \frac{a_k}{b_k}\beta_k\right) \end{aligned}$$

Clearly $sup(\alpha_i) \subseteq sup(\alpha)$ and $sup(\beta_i) \subseteq sup(\beta)$ for all α_i, β_i . All α_i and β_i must have distinct supports, so $sup(\alpha_i) \subsetneq sup(\alpha)$ and $sup(\beta_i) \subsetneq sup(\beta)$ for all α_i and β_i . This contradicts the minimality of $(\alpha \beta)$.

2. **For weak minimality:** Support towards a contradiction that there are $x, x', x'' \in Z$ s.t. $x = x' + x''$. Per definition of Z there are $(\alpha \beta), (\alpha' \beta'), (\alpha'' \beta'') \in MPF(C)$ s.t. $x = \alpha X \frown \beta Y, x' = \alpha' X \frown \beta' Y$ and $x'' = \alpha'' X \frown \beta'' Y$. Hence

$$\begin{aligned} x &= \alpha' X \frown \beta' Y + \alpha'' X \frown \beta'' Y \\ &= \alpha' X + \alpha'' X \frown \beta' Y + \beta'' Y \\ &= (\alpha' + \alpha'') X \frown (\beta' + \beta'') Y \end{aligned}$$

But also $x = \alpha X \frown \beta Y$, so $\alpha X = (\alpha' + \alpha'') X$ and $\beta Y = (\beta' + \beta'') Y$. By unique decomposition, $\alpha = \alpha' + \alpha''$ and $\beta = \beta' + \beta''$. But this contradicts that $(\alpha \beta)$ is minimal in C .

□

C.2.4 Proofs for Concrete Flow Semantics

Theorem 11. The proof is by induction on P , but we strengthen the induction hypothesis with the statement that W_{LTF} is the flow matrix of $(PN, \prec_{T_{LTF}}, \prec_{S_{LTF}})$. The relevant

cases are the ones concerned with the concrete semantics as the rest go through trivially. We show only the case of parallel composition for T-flows. The remaining cases, including those for P-flows, are easier.

Case: parallel composition for T-flows. Let $PN_1 \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{\mathcal{PN}}$, $PN_2 \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{\mathcal{PN}}$, $LTF_1 \stackrel{\Delta}{\simeq} \llbracket P_1 \rrbracket_{LTF}$, and $LTF_2 \stackrel{\Delta}{\simeq} \llbracket P_2 \rrbracket_{LTF}$; we here omit the parameters of the general semantical function for the sake of notational simplicity. By the induction hypothesis, W_{LTF_1} and W_{LTF_2} are the flow matrices of PN_1 and PN_2 , respectively. Then the composition W is per definition the flow matrix of $PN_1 \mid_s PN_2$ under the chosen orderings. We then reason as follows:

$$\begin{aligned}
h_{LTF}(\Delta S) &= MTF^{\text{Par}}([MTF_1], [MTF_2], W^{ss}) \\
&= MTF^{\text{Par}}([h_{LTF_1}(\Delta S')], [h_{LTF_2}(\Delta S')], W^{ss}) \\
&= MTF^{\text{Par}}([MTF(PN_1 \setminus \Delta S')], [MTF(PN_2 \setminus \Delta S')], W^{ss}) \\
&\quad (\text{by the induction hypothesis}) \\
&= MTF^{\text{Par}}([MTF(PN_1 \setminus \Delta S'' \setminus \Delta S)], [MTF(PN_2 \setminus \Delta S'' \setminus \Delta S)], W^{ss}) \\
&= MTF(PN_1 \setminus \Delta S \mid_s PN_2 \setminus \Delta S) \\
&\quad (\text{by Theorem 7}) \\
&= MTF((PN_1 \mid_s PN_2) \setminus \Delta S)
\end{aligned}$$

This completes the proof, since per definition $\llbracket P_1 | P_2 \rrbracket_{\mathcal{PN}} = PN_1 \mid_s PN_2$. □

C.3 Proofs for GEC

C.3.1 Injectivity

We write $f \downarrow X$ for the restriction of the function f to the domain $X \subseteq \text{dom}(f)$. We say that $\Theta = \{(\theta_i, \rho_i, \sigma_i, \tau_i)\}$ is ρ -injective if $\text{dom}_s(\theta_i) = \rho_i$; then $\theta_i \downarrow \text{dom}_s(\theta_i)$ is injective per definition of context-sensitive substitutions.

Lemma 9. *If Θ_1 and Θ_2 are ρ -injective, then also $\Theta_1 \otimes \Theta_2$ is ρ -injective.*

Proof. let $\Theta_1 = \{(\theta_i, \rho_i, \sigma_i, \tau_i)\}$ and $\Theta_2 = \{(\theta'_j, \rho'_j, \sigma'_j, \tau'_j)\}$. Take any $\theta_i \cup \theta'_j$ in $\Theta_1 \otimes \Theta_2$. Per assumption, $\text{dom}_s(\theta_i) = \rho_i$ and $\text{dom}_s(\theta'_j) = \rho'_j$. Hence $\text{dom}_s(\theta_i \cup \theta'_j) = \rho_i \cup \rho'_j$, so $\Theta_1 \otimes \Theta_2$ is ρ -injective. □

Lemma 10. Let P be a compartment-free program, let Γ be a module environment and let b be a binary string. If $\Gamma(id_m)(\underline{a}, b')$ is ρ -injective for all $id_m \in \text{dom}(\Gamma) \cap FM(P)$, all b' and all matching \underline{a} , then $\llbracket P \rrbracket_{\text{gs}} \Gamma, b$ is also ρ -injective.

Proof. Suppose that the precondition holds, i.e. that $\Gamma(id_m)(\underline{a}, b)$ is defined and ρ -injective for all $id_m \in \text{dom}(\Gamma) \cap FM(P)$. We then proceed by induction on P with selected cases given below; the remaining cases are similar or easier.

- $P = u : t(Q^t)$. Then $\rho_i = \text{dom}_s(\theta_i)$ per definition.
- $P = \mathbf{0}$. Then $\rho_i = \text{dom}_s(\theta_i) = \emptyset$ per definition.
- $P = (id_m(\underline{u}) = P_1 ; P_2)$. Let $f(\underline{a}, b) \stackrel{\Delta}{=} \llbracket P_1 \{ \underline{u}.i \mapsto \underline{a}.i \} \rrbracket_{\text{gs}} \Gamma, b$. By the induction hypothesis, $f(\underline{a}, b)$ is ρ -injective. Hence the precondition also holds for $\Gamma' \stackrel{\Delta}{=} \Gamma \langle id_m \mapsto f \rangle$. By the induction hypothesis, also $\llbracket P_2 \rrbracket_{\text{gs}} \Gamma', b'$ is ρ -injective.
- $P = id_m(\underline{a})$. Follows immediately from the precondition.
- $P = P_1 | C$. The induction hypothesis gives that $\llbracket P_1 \rrbracket_{\text{gs}} \Gamma, b$ is ρ -injective. We also get that $\llbracket C \rrbracket_{\text{gs}}$ is ρ -injective for all three cases of C per definition of the denotation function. It follows from Lemma 9 that also $\llbracket P_1 \rrbracket_{\text{gs}} \Gamma, b \otimes \llbracket C \rrbracket_{\text{gs}}$ is ρ -injective.
- $P = \text{new}x.P_1$. By the induction hypothesis, $\llbracket P_1 \{ x \mapsto x' \} \rrbracket_{\text{gs}} \Gamma, b$ is ρ -injective for any x' .

□

Let P_0 be a program and let $\Theta \stackrel{\Delta}{=} \{(\theta_i, \rho_i, \sigma_i, \tau_i)\}$. We say that Θ is P_0 -*injective* if $\theta_i \downarrow (\text{dom}_s(\theta_i) \cap \text{FV}(P_0))$ is injective. The following proposition states a strong property of piece-wise injectivity which serves as a useful induction hypothesis in the proof; Proposition 9.2.1 is an immediate corollary.

Proposition C.3.1 (Piece-wise injectivity). *Let P_0 be a compartment-free program with $FM(P_0) = \emptyset$, let $\mathcal{C}(\cdot)$ be a well-formed context, let Γ be an environment and let b be a binary string. If $\mathcal{C}(\cdot)$ has a hole, or if there is an $id_m \in \text{dom}(\Gamma) \cap FM(\mathcal{C}(\cdot))$ s.t $\Gamma(id_m)(\underline{a}, b')$ is P_0 -injective for all b' and matching \underline{a} , then also $\llbracket \mathcal{C}(P_0) \rrbracket_{\text{gs}} \Gamma, b$ is P_0 -injective.*

Proof. Suppose that the precondition holds for $\mathcal{C}(\cdot)$ and Γ . We then proceed by induction on $\mathcal{C}(\cdot)$ with selected cases given below; the remaining cases are similar or easier.

- $\mathcal{C}(\cdot) = \cdot$. Then $\llbracket \mathcal{C}(P_0) \rrbracket_{\text{gs}} \Gamma, b = \llbracket P_0 \rrbracket_{\text{gs}} \Gamma, b$ is ρ -injective by Lemma 10, and ρ -injectivity implies P_0 -injectivity.
- $\mathcal{C}(\cdot) = \mathcal{C}_1(\cdot) \parallel \mathcal{C}_2(\cdot)$. If the precondition holds for $\mathcal{C}(\cdot)$ then it must hold for at least one side, say for $\mathcal{C}_1(\cdot)$ without loss of generality. Then $\llbracket \mathcal{C}_1(P_0) \rrbracket_{\text{gs}} \Gamma, b$ is P_0 -injective by the induction hypothesis. The union of a function injective on some interval with any other function, when defined, is also injective on this interval.
- $\mathcal{C}(\cdot) = (id_m(\underline{u}) = \mathcal{C}_1(\cdot); \mathcal{C}_2(\cdot))$. There are two cases to consider:
 1. The precondition holds for $\mathcal{C}_1(\cdot)$ and Γ . Then

$$f(\underline{a}, b) = \llbracket \mathcal{C}_1(P_0) \{ \underline{u}.i \mapsto \underline{a}.i \} \rrbracket_{\text{gs}} \Gamma, b = \llbracket \mathcal{C}_1 \{ \underline{u}.i \mapsto \underline{a}.i \} (P_0) \rrbracket_{\text{gs}} \Gamma, b$$
 per assumption that context instantiations are capture free, i.e. no free variables of P_0 become bound by formal parameters. By the induction hypothesis, $f(\underline{a}, b)$ is P_0 -injective. Then the precondition holds for $\mathcal{C}_2(\cdot)$ and $\Gamma' = \Gamma \langle id_m \mapsto f \rangle$ since $id_m \in FM(\mathcal{C}_2(\cdot))$ by the assumption of well-formedness. By the induction hypothesis, $\llbracket \mathcal{C}_2(P_0) \rrbracket_{\text{gs}} \Gamma', b$ is P_0 -injective.
 2. The precondition holds for $\mathcal{C}_2(\cdot)$ and Γ . It then also holds for $\mathcal{C}_2(\cdot)$ and Γ' as defined above because $id_m \notin \text{dom}(\Gamma)$ by the assumption of well-formedness. By the induction hypothesis, $\llbracket \mathcal{C}_2(P_0) \rrbracket_{\text{gs}} \Gamma', b$ is P_0 -injective.

- $\mathcal{C}(\cdot) = id_m(\underline{a})$. There is no hole in $id_m(\underline{a})$, so if the precondition holds, there is an $id'_m \in \text{dom}(\Gamma) \cap FM(\mathcal{C}(\cdot))$ s.t $\Gamma(id'_m)(\underline{a}, b)$ is P_0 -injective for all b and matching \underline{a} . Since $FM(\mathcal{C}(\cdot)) = \{id_m\}$, we must have that $id'_m = id_m$.
- $\mathcal{C}(\cdot) = id_c[\mathcal{C}'(\cdot)]$. If the precondition holds for $\mathcal{C}(\cdot)$ then it must also hold for $\mathcal{C}'(\cdot)$. Per definition, $\llbracket id_c[\mathcal{C}'(P_0)] \rrbracket_{\text{gs}} \Gamma, b$ has the same substitutions as $\llbracket \mathcal{C}'(P_0) \rrbracket_{\text{gs}} \Gamma, b$, and the latter is P_0 -injective by the induction hypothesis.
- $\mathcal{C}(\cdot) = \text{new}_x \mathcal{C}'(\cdot)$. Again we rely on context instantiations being capture-free so that $x \notin \text{FV}(P_0)$. Hence $\mathcal{C}'(P_0) \{ x \mapsto x' \} = \mathcal{C}' \{ x \mapsto x' \} (P_0)$ and the induction hypothesis applies.

□

C.3.2 Non-interference

Let $P_0 = u : t(Q^t)$ be a basic program. We say that $\Theta = \{(\theta_i, \rho_i, \sigma_i, \tau_i)\}$ is P_0 -sound if $u\theta_i : t(Q) \in \mathcal{K}_b$ for some Q and $FS(Q) \setminus FS(Q^t\theta_i) \subseteq \tau_i$.

Lemma 11. Let $P_0 = u : t(Q^t)$ be a basic program and let Θ, Θ' be two substitutions with ΘP_0 -sound. Then $\Theta \otimes \Theta'$ is also P_0 -sound.

Proof. Let $\Theta = \{(\theta_i, \rho_i, \sigma_i, \tau_i)\}$, $\Theta' = \{(\theta'_j, \rho'_j, \sigma'_j, \tau'_j)\}$ and take any $(\theta_i \cup \theta'_j, \rho_i \cup \rho'_j, \sigma_i \cup \sigma'_j, \tau_i \cup \tau'_j) \in \Theta \otimes \Theta'$. Per assumption that Θ is P_0 -sound, there is a Q s.t. $u\theta_i : t(Q) \in \mathcal{K}_b$, $Q^t\theta_i \subseteq Q$ and $FS(Q) \setminus FS(Q^t\theta_i) \subseteq \tau_i$. Note that $FV(Q^t) \subseteq dom(\theta_i)$ and $FV(u) \subseteq dom(\theta_i)$ because \mathcal{K}_b consists of ground terms, so $Q^t\theta_i = Q^t(\theta_i \cup \theta'_j)$ and $u\theta_i = u(\theta_i \cup \theta'_j)$. Therefore also $u(\theta_i \cup \theta'_j) : t(Q) \in \mathcal{K}_b$, $Q^t(\theta_i \cup \theta'_j) \subseteq Q$ and $FS(Q) \setminus FS(Q^t(\theta_i \cup \theta'_j)) \subseteq \tau_i \subseteq (\tau_i \cup \tau'_j)$.

□

The following proposition states a strong property of non-interference which serves as a useful induction hypothesis in the proof. Proposition 9.2.2 follows as an immediate corollary.

Proposition C.3.2 (Non-interference). *Let $P_0 = u : t(Q^t)$ be a basic program, let $C(\cdot)$ be a compartment-free, well-formed context, let Γ be an environment and let b be a binary string. If $C(\cdot)$ has a hole, or if there is an $id_m \in dom(\Gamma) \cap FM(C(\cdot))$ s.t $\Gamma(id_m)(\underline{a}, b')$ is P_0 -sound for all b' and matching \underline{a} , then also $\llbracket C(P_0) \rrbracket_{gs} \Gamma, b$ is P_0 -sound.*

Proof. Suppose that the precondition holds for $C(\cdot)$ and Γ . We then proceed by induction on $C(\cdot)$ with selected cases given below; the remaining cases are similar or easier, and the cases for module definition, module invocation and new variables are shown as in the proof of Proposition C.3.1.

- $C(\cdot) = \cdot$. Then $\llbracket C(P_0) \rrbracket_{gs} \Gamma, b = \llbracket u : t(Q^t) \rrbracket_{gs} \Gamma, b$ and the result follows directly from the definition of the denotation function.
- $C(\cdot) = C_1(\cdot) \parallel C_2(\cdot)$. If the precondition holds for $C(\cdot)$ then it must hold for one of the sides, say for $C_1(\cdot)$ without loss of generality. Then $\llbracket C_1(P_0) \rrbracket_{gs} \Gamma, b$ is P_0 -sound by the induction hypothesis. By Lemma 11 also $\llbracket C_1(P_0) \rrbracket_{gs} \Gamma, b \otimes \llbracket C_2(P_0) \rrbracket_{gs} \Gamma, b$ is P_0 -sound.

□

Table of Notation

General

2^X	power set of X
b	binary string
$f\langle g \rangle$	update of function f with g
$MS(X)$	set of multisets of X
n	natural number
$\prod_{i \in I} X_i$	dependent set
r	real number
$x \stackrel{\Delta}{=} y$	definition (undefined if y undefined)
$x \stackrel{\Delta}{\simeq_t} y$	definition (undefined if y ill-typed)
$\{x_i \mapsto y_i\}$	indexed set of pairs-representation of a partial finite function
$ x $	size/length of a set/list
\underline{x}	list
\mathcal{x}	list with set interpretation
$\underline{x}.Q$	sublist of \underline{x} with indices Q

LBS

$adapt(v_s)$	species value interface adaptation function, page 82
$close(v_s)$	species annotation closing function, page 82

$\llbracket e_a \rrbracket_a \Gamma_c, \Gamma_s, \Gamma_a, v_c$	denotation function for algebraic rate expressions, page 84
$\llbracket e_b \rrbracket_b \Gamma_x$	denotation function for boolean expressions, page 67
$\llbracket e_c \rrbracket_c \Gamma_c, b$	denotation function for compartment expressions, page 69
$\llbracket D \rrbracket_d \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b$	denotation function for definitions, page 95
$\llbracket P \rrbracket_{dp}$	denotation function for derived LBS programs, page 92
$\llbracket e_m \rrbracket_m \Gamma_x$	denotation function for modification site expressions, page 66
$\llbracket P \rrbracket_p \Gamma_c, \Gamma_s, \Gamma_a, \Gamma_m, b, v_c$	denotation function for LBS programs, page 86
$\llbracket e_s \rrbracket_s \Gamma_c, \Gamma_s$	denotation function for species expressions, page 76
$\llbracket e_{s+} \rrbracket_{s+} \Gamma_c, \Gamma_s, b$	denotation function for extended species expressions, page 78
$\llbracket \rho \rrbracket_t$	denotation function for modification site types, page 66
$default(\rho)$	default value assignment function, page 66
δ_n^m	binary string of length m with 1 in the n th position and 0 elsewhere, page 86
e_a	algebraic expression, page 59
e_b	boolean expression, page 56
e_c	compartment expression, page 55
e_m	modification site expression, page 56
e_r	rate expression, page 59
e_s	species expression, page 56
e_{s+}	extended species expression, page 56
Γ	generic environment, page 84
Γ_a	algebraic rate function environment, page 84
Γ_c	compartment environment, page 69
Γ_m	module environment, page 86

Γ_s	species environment, page 76
Γ_{so}	species output environment, page 86
Γ_x	variable environment, page 66
$e_s \text{ or } e'_s$	nondeterministic choice, page 56
$FS(P)$	free species function, page 92
id_a	algebraic expression identifier, page 59
id_c	compartment identifier, page 54
id_m	module identifier, page 59
id_s	species identifier, page 56
n_c	compartment name, page 69
n_m	modification site name, page 56
n_s	species name, page 56
1_c	nil compartment, page 69
0_p	nil program, page 59
0_s	nil species, page 56
P	program, page 59
$P \mid P'$	parallel composition, page 59
$P \parallel P'$	variation composition, page 59
R	normal form reaction, page 84
$seal(e_m, b)$	modification site sealing function, page 66
\top	top-level (world) compartment, page 55
$e_m : \rho$	typing relation on modification site expressions, page 66
$e_m \langle e'_m \rangle$	modification site update function, page 66

v_a	algebraic value, page 84
v_c	compartment value, page 69
v_{ga}	ground normal form algebraic value, page 100
v_{gns}	ground normal form species value, page 81
v_{gr}	ground normal form rate value, page 100
v_m	modification site value, page 66
v_{ns}	normal form species value, page 81
v_r	rate value, page 84
v_s	species value, page 72
v_{us}	unboxed species value, page 72
α_σ	typed modification site assignment, page 72
β_σ	ground typed modification site assignment, page 81
ι_m	modification site interface, page 72
ξ	species annotation, page 56
ρ	modification site type, page 53
ι	species interface, page 72
$\underline{x} \times_\circ \underline{y}$	Cartesian product of lists with given pairing, page 53

Concrete Semantics

$F^{\text{in}}(t, s)$	Petri net flow-in function, page 102
$F^{\text{out}}(t, s)$	Petri net flow-out function, page 102
S	set of Petri net places, page 102
T	set of Petri net transitions, page 102
e_{bd}	binding expression, page 114

O	generic semantical object, page 65
$G_S(G, b)$	ground normal form reaction assignment, page 99
$I_S(v_{\text{gns}}, r)$	initial population assignment, page 65
$\mathbf{0}_S$	nil program assignment, page 65
$O_1 _S O_2$	parallel composition assignment, page 65
$R_S(R, b)$	normal form reaction assignment, page 65
S	set of generic semantical object, page 65
CPN	coloured Petri net, page 105
\mathcal{CPN}	set of all Coloured Petri nets, page 105
D	structure of LBS ODEs, page 108
\mathcal{D}	set of all structures of LBS ODEs, page 108
K	LBS- κ program, page 113
\mathcal{K}	set of all LBS- κ programs, page 113
LPF	LBS P-flow structure, page 136
\mathcal{LPF}	set of all LBS P-flow structures, page 136
LTF	LBS T-flow structure, page 134
\mathcal{LTF}	set of all LBS T-flow structures, page 134
PN	Petri net, page 102
\mathcal{PN}	set of all Petri nets, page 102
V	LBS CTMC with initial conditions, page 110
\mathcal{V}	set of all LBS CTMCs with initial conditions, page 110

Petri Net Flows

$gcd(x)$	greatest common divisor of entries in a vector x , page 122
----------	---------------------------------------------------------------

$\min(X)$	flow minimisation function, page 122
$MPF(PN)$	minimal P-flow function, page 121
$M_wPF(PN)$	weakly minimal P-flow function, page 122
$MTF(PN)$	minimal T-flow function, page 121
$M_wTF(PN)$	weakly minimal T-flow function, page 122
$PF(PN)$	P-flow function, page 121
OPN	ordered Petri net, page i
$sup(x)$	support of a vector x , page 121
$TF(PN)$	T-flow function, page 121
W^{in}	Petri net flow-in matrix, page 120
W^{out}	Petri net flow-out matrix, page 120
W	Petri net (net) flow matrix, page 120
$x \frown y$	join of P-flows, page 131
$(x)^T$	vector/Matrix transposition, page 121
GEC	
a	actual parameter, page 162
$C(\cdot)$	GEC program context, page 171
C	constraint, page 162
$\llbracket P \rrbracket_{\text{gd}} \Gamma$	device denotation function, page 172
$\llbracket P \rrbracket_{\text{gr}} \Gamma$	reaction denotation function, page 176
$\llbracket P \rrbracket_{\text{gs}} \Gamma$	substitution denotation function, page 168
Γ	module environment, page 168
$FM(P)$	free module identifiers function, page 171

$FV(P)$	free variables function, page 162
$FS(P)$	free species function, page 162
Δ	set of device templates, page 172
δ	device template, page 172
$\Theta_1 \odot \Theta_2$	context-sensitive substitution composition, page 167
L	GEC reaction program, page 175
ρ	injective domain (set of variables), page 167
σ	set of used species, page 167
τ	set of excluded species, page 167
Θ	set of context-sensitive substitutions, page 167
θ	substitution, page 167
T	transport reaction, page 162
T^\downarrow	transport reaction without compartment identifiers , page 167
id_p	part identifier, page 162
id_m	module identifier, page 162
$\mathbf{0}$	nil program, page 162
K	numerical constraint, page 162
P	program, page 162
$P ; P'$	sequential composition, page 162
$P C$	constraint composition, page 162
$P P'$	parallel composition, page 162
Q	set of properties, page 162
R	reaction, page 162

S	species, page 162
u	formal parameter, page 162
v_b	boolean value, page 162
x	variable, page 162

Bibliography

- [1] Hassan Afif, Noureddine Allali, Martine Couturier, and Laurence Van Melderen. The ratio between CcdA and CcdB modulates the transcriptional repression of the ccd poison-antidote system. *Molecular Microbiology*, 41(1):73–82, 2001.
- [2] Ozgur E. Akman, Federica Ciocchetta, Andrea Degasperi, and Maria Luisa Guerriero. Modelling biological clocks with Bio-PEPA: Stochasticity and robustness for the *Neurospora crassa* circadian network. In *Proc. CMSB*, pages 52–67, Berlin, Heidelberg, 2009. Springer-Verlag.
- [3] B. Alberts, D. Bray, A. Johnson, J. Lewis, K. Roberts M. Raff, and P. Walter. *Essential Cell Biology – An Introduction to the Molecular Biology of the Cell*. Garland Publishing, 1998.
- [4] E. Andrianantoandro, S. Basu, D.K. Karig, and R. Weiss. Synthetic biology: new engineering rules for an emerging discipline. *Molecular Systems Biology*, 2, 2006.
- [5] F.K. Balagaddé, H. Song, J. Ozaki, C.H. Collins, M. Barnet, F.H. Arnold, S.R. Quake, and L. You. A synthetic *Escherichia coli* predator-prey ecosystem. *Molecular Systems Biology*, 4(187), 2008.
- [6] Frank T. Bergmann and Herbert M. Sauro. SBW - a modular framework for systems biology. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, pages 1637–1645. Winter Simulation Conference, 2006.
- [7] Paul Blanchard, Robert L. Devaney, and Glenn R. Hall. *Differential Equations*. Brooks/Cole, 2002.
- [8] Michael L. Blinov, Jin Yang, James R. Faeder, and William S. Hlavacek. Graph theory for rule-based modeling of biochemical networks. *Trans. on Comput. Syst. Biol.*, 4230:89–106, 2006.

- [9] Ralf Blossey, Luca Cardelli, and Andrew Phillips. Compositionality, stochasticity and cooperativity in dynamic models of gene regulation. *HFSP Journal*, 2(1):17–28, 2008.
- [10] Benjamin J. Bornstein, Sarah M. Keating, Akiya Jouraku, and Michael Hucka. LibSBML: An API library for SBML. *Bioinformatics*, 24(6):880–881, 2008.
- [11] A. Bourjij, M. Boutayeb, and T. Cecchin. A decentralized approach for computing invariants in large scale and interconnected Petri nets. In *Proc. IEEE International Conference on Systems, Man, and Cybernetics*, volume 2, pages 1741–1746, 1997.
- [12] E. C. Butcher, E. L. Berg, and E. J. Kunkel. Systems biology in drug discovery. *Nature biotechnology*, 22(10):1253–1259, October 2004.
- [13] Yizhi Cai, Brian Hartnett, Claes Gustafsson, and Jean Peccoud. A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. *Bioinformatics*, 23(20):2760–2767, 2007.
- [14] Yizhi Cai, Matthew W. Lux, Laura Adam, and Jean Peccoud. Modeling structure-function relationships in synthetic DNA sequences using attribute grammars. *PLoS Comput Biol*, 5(10):e1000529, 2009.
- [15] Muffy Calder, Stephen Gilmore, and Jane Hillston. Modelling the influence of RKIP on the ERK signalling pathway using the stochastic process algebra PEPA. *Trans. on Comput. Syst. Biol. VII*, 4230:1–23, 2006.
- [16] Luca Cardelli. Brane calculi. In Vincent Danos and Vincent Schächter, editors, *Proc. CMSB*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–280. Springer, 2005.
- [17] Luca Cardelli, Emmanuelle Caron, Philippa Gardner, Ozan Kahramanogullari, and Andrew Phillips. A process model of Rho GTP-binding proteins. *Theor. Comput. Sci.*, 410(33-34):3166–3185, 2009.
- [18] Luca Cardelli and Gianluigi Zavattaro. On the computational power of biochemistry. In *AB '08: Proceedings of the 3rd international conference on Algebraic Biology*, pages 65–80, Berlin, Heidelberg, 2008. Springer-Verlag.

- [19] Nathalie Chabrier-Rivier, Marc Chiaverini, Vincent Danos, François Fages, and Vincent Schächter. Modeling and querying biomolecular interaction networks. *Theor. Comput. Sci.*, 325(1):25–44, 2004.
- [20] Nathalie Chabrier-Rivier, François Fages, and Sylvain Soliman. The biochemical abstract machine BIOCHAM. In V. Danos and V. Schächter, editors, *Proc. CMSB*, volume 3082 of *LNCS*, pages 172–191. Springer, 2004.
- [21] William W. Chen, Birgit Schoeberl, Paul J. Jasper, Mario Nipel, Ulrik B. Nielsen, Douglas A. Lauffenburger, and Peter K. Sorger. Input-output behavior of ErbB signaling pathways as revealed by a mass action model trained against dynamic data. *Mol Syst Biol*, 5(239), 2009.
- [22] Marc Chiaverini and Vincent Danos. A core modeling language for the working molecular biologist (abstract). In *Proc. CMSB*, page 166. Springer-Verlag, 2003.
- [23] S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.
- [24] Federica Ciocchetta and Jane Hillston. Bio-PEPA: An extension of the process algebra PEPA for biochemical networks. *Electron. Notes Theor. Comput. Sci.*, 194(3):103–117, 2008.
- [25] Federica Ciocchetta and Jane Hillston. Bio-PEPA: a framework for the modelling and analysis of biological systems. *Theor. Comput. Sci.*, 410(33-34):3065–3084, 2009.
- [26] Troels C. Damgaard, Vincent Danos, and Jean Krivine. A language for the cell. Technical Report TR-2008-116, IT University of Copenhagen, 2008.
- [27] Troels C. Damgaard and Jean Krivine. A generic language for biological systems based on Bigraphs. Technical Report TR-2008-115, IT University of Copenhagen, 2008.
- [28] Vincent Danos. Agile modelling of cellular signalling. In *Computation in Modern Science and Engineering*, volume 2, Part A 963, pages 611–614, 2007.
- [29] Vincent Danos, Jérôme Feret, Walter Fontana, Russ Harmer, and Jean Krivine. Rule-based modelling and model perturbation. *Trans. on Comput. Syst. Biol.*, 5750(11):116–137, 2009.

- [30] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-based modelling of cellular signalling. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 17–41. Springer, 2007. Tutorial paper.
- [31] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Scalable simulation of cellular signaling networks. In *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 139–157. Springer, 2007.
- [32] Lorenzo Dematté, Corrado Priami, and Alessandro Romanel. Modelling and simulation of biological processes in BlenX. *SIGMETRICS Performance Evaluation Review*, 35(4):32–39, 2008.
- [33] M. B. Elowitz and S. Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335–338, 2000.
- [34] D. Endy. Foundations for engineering biology. *Nature*, 438(24):449–453, 2005.
- [35] J. R. Faeder, Michael L. Blinov, and William S. Hlavacek. Graphical rule-based representation of signal-transduction networks. In L. M. Liebrock, editor, *Proc. 2005 ACM Symp. Appl. Computing*, pages 133–140. ACM Press, 2005.
- [36] Dan Ferber. Synthetic biology: Microbes made to order. *Science*, 303(5655):158–161, January 2004.
- [37] Jérôme Feret, Vincent Danos, Jean Krivine, Russ Harmer, and Walter Fontana. Internal coarse-graining of molecular systems. *Proceedings of the National Academy of Sciences*, 106(16):6453–6458, April 2009.
- [38] Hartmann Genrich, Robert Küffner, and Klaus Voss. Executable Petri net models for the analysis of metabolic pathways. *J STTT*, 3(4):394–404, 2001.
- [39] David Gilbert, Monika Heiner, Susan Rosser, Rachael Fulton, Xu Gu, and Maciej Trybilo. A case study in model-driven synthetic biology. In *Biologically-inspired cooperative computing*, volume 268 of *IFIP International Federation for Information Processing*, pages 163–175. Springer, 2008.
- [40] P. J. E. Goss and J. Peccoud. Quantitative modeling of stochastic systems in molecular biology by using stochastic Petri nets. *PNAS*, 95(12):6750–6755, 1998.

- [41] Maria Luisa Guerriero, John K. Heath, and Corrado Priami. An automated translation from a narrative language for biological modelling into process algebra. In M. Calder and S. Gilmore, editors, *Proc. CMSB*, volume 4695 of *LNCS*, pages 136–151. Springer, 2007.
- [42] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
- [43] Monika Heiner, David Gilbert, and Robin Donaldson. Petri nets for systems and synthetic biology. In M. Bernardo, P. Degano, and G. Zavattaro, editors, *SFM 2008*, volume 5016 of *LNCS*, pages 215–264, 2008.
- [44] Monika Heiner and Ina Koch. Petri net based model validation in systems biology. In *Applications and Theory of Petri Nets 2004*, volume 3099 of *LNCS*, pages 216–237. Springer, 2004.
- [45] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
- [46] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. Copasi—a complex pathway simulator. *Bioinformatics*, 22(24):3067–3074, 2006.
- [47] Michael Hucka et al. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.
- [48] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer, 1992.
- [49] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 2. Springer, 1995.
- [50] Hiroaki Kitano. Systems biology: A brief overview. *Science*, 295(5560):1662–1664, March 2002.
- [51] B. Kofahl and E. Klipp. Modelling the dynamics of the yeast pheromone pathway. *Yeast*, 21(10):831–850, 2004.
- [52] Fritz Krückeberg and Michael Jaxy. Mathematical methods for calculating invariants in Petri nets. In *Advances in Petri Nets 1987, covers the 7th European*

- Workshop on Applications and Theory of Petri Nets*, pages 104–131, London, UK, 1987. Springer.
- [53] Marek Kwiatkowski and Ian Stark. The continuous π -calculus: a process algebra for biochemical modelling. In M. Heiner and A. M. Uhrmacher, editors, *Proc. CMSB*, number 5307 in LNCS, pages 103–122. Springer-Verlag, 2008.
- [54] Nicolas Le Novère et al. The systems biology graphical notation. *Nature Biotechnology*, 27:735–741, 2009.
- [55] Dong-Yup Leea, Ralf Zimmere, Sang Yup Leea, and Sunwon Park. Colored Petri net modeling and simulation of signal transduction pathways. *Metab Eng*, 8(2):112–22, 2006.
- [56] Chen Li, Qi-Wei Ge, Mitsuru Nakata, Hiroshi Matsuno, and Satoru Miyano. Modelling and simulation of signal transductions in an apoptosis pathway by using timed petri nets. *Journal of Biosciences*, 32(1):113–127, January 2007.
- [57] James C. W. Locke, Megan M. Southern, László Kozma-Bognár, Victoria Hibberd, Paul E. Brown, Matthew S. Turner, and Andrew J. Millar. Extension of a genetic network model by iterative experimentation and mathematical analysis. *Molecular Systems Biology*, 1(1):msb4100018–E1–msb4100018–E9, June 2005.
- [58] I. C. Rojas M. *Compositional construction and analysis of Petri net systems*. PhD thesis, School of Informatics, University of Edinburgh, 1998.
- [59] A. Mallavarapu, M. Thomson, B. Ullian, and J. Gunawardena. Programming with models: modularity and abstraction provide powerful capabilities for systems biology. *J. R. Soc. Interface*, 2008.
- [60] Aneil Mallavarapu. *Little b web site*, Accessed September 4, 2009.
<http://www.littleb.org/>.
- [61] M. A. Marchisio and J. Stelling. Computational design of synthetic gene circuits with composable parts. *Bioinformatics*, 24:1903–1910, 2008.
- [62] Gérard Memmi and Gérard Roucaïrol. Linear algebra in net theory. In *Proc. Advanced Course on General Net Theory of Processes and Systems*, pages 213–223. Springer, 1980.

- [63] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
- [64] Registry of Standard Biological Parts, Accessed July 6 2009.
<http://partsregistry.org>.
- [65] Tadao Murata. Petri nets: properties, analysis and applications. *Proc. IEEE*, 77(4):541–580, 1989.
- [66] Josselin Noirel, Saw Y. Ow, Guido Sanguinetti, and Phillip C. Wright. Systems biology meets synthetic biology: a case study of the metabolic effects of synthetic rewiring. *Mol. BioSyst.*, 5(10):1214–1223, 2009.
- [67] Gheorghe Paun and Grzegorz Rozenberg. A guide to membrane computing. *Theor. Comput. Sci.*, 287(1):73–100, 2002.
- [68] Michael Pedersen. Compositional definitions of minimal flows in Petri nets. In M. Heiner and A. M. Uhrmacher, editors, *Proc. CMSB*, volume 5307 of *LNCS*, pages 288–307. Springer-Verlag, 2008.
- [69] Michael Pedersen and Andrew Phillips. Towards programming languages for genetic engineering of living cells. *J. R. Soc. Interface special issue*, 2009.
- [70] Michael Pedersen and Gordon Plotkin. A language for biochemical systems. In M. Heiner and A. M. Uhrmacher, editors, *Proc. CMSB*, volume 5307 of *LNCS*, pages 63–82. Springer-Verlag, 2008.
- [71] Michael Pedersen and Gordon Plotkin. A language for biochemical systems: design and formal specification. 2009. To appear in *Trans. on Comput. Syst. Biol.*
- [72] C Peyronnaux and A Eychne. The Raf/MEK/ERK pathway: new concepts of activation. *Biol Cell*, 93(1-2):53–62, 2001.
- [73] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [74] Gordon Plotkin. A calculus of biochemical systems. In preparation.
- [75] C. Priami. Stochastic pi-calculus. *The Computer Journal*, 38(7):578–589, 1995.

- [76] C. Priami and P. Quaglia. Beta binders for biological interactions. In V. Danos and V. Schächter, editors, *Proc. CMSB*, volume 3082 of *LNBI*, pages 20–33. Springer, 2005.
- [77] Venkatramana N. Reddy, Michael L. Mavrovouniotis, and Michael N. Liebmant. Petri net representations in metabolic pathways. In *Proc. Int. Conf. Intell. Syst. Mol. Biol.*, pages 328–336, 1993.
- [78] A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.
- [79] Aviv Regev, Ekaterina M. Paninab, William Silverman, Luca Cardelli, and Ehud Shapiro. BioAmbients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.
- [80] W. Reisig. *Petri nets*. EATCS Monographs on Theoretical Computer Science. Springer, 1982.
- [81] T. Runge. Application of Coloured Petri Nets in Systems Biology. In K. Jensen, editor, *Proceedings of the Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 77–96, 2004.
- [82] Andrea Sackmann, Dorota Formanowicz, Piotr Formanowicz, Ina Koch, and Jacek Blazewicz. An analysis of the Petri net based model of the human body iron homeostasis process. *Comput. Biol. Chem.*, 31(1):1–10, 2007.
- [83] V. Sassone. The algebraic structure of Petri nets, 2004.
- [84] S. Schuster and C. Hilgetag. On elementary flux modes in biochemical reaction systems at steady state. *J. Biol. Syst.*, 2:165–182, 1994.
- [85] Lucian P. Smith, Frank T. Bergmann, Deepak Chandran, and Herbert M. Sauro. Antimony: a modular model definition language. *Bioinformatics*, 25(18):2452–2454, 2009.
- [86] L. Jason Steggles, Richard Banks, Oliver Shaw, and Anil Wipat. Qualitatively modelling and analysing genetic regulatory networks: a Petri net approach. *Bioinformatics*, 23(3):336–343, 2007.

- [87] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# (Expert's Voice in .Net)*. APRESS ACADEMIC, 2007.
- [88] C. Taubner et al. Modelling and simulation of the TLR4 pathway with coloured Petri nets. In *Proc. Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 2009–2012. Engineering in Medicine and Biology Society, 2006.
- [89] Darren J. Wilkinson. *Stochastic Modelling for Systems Biology*. Chapman & Hall/CRC, 2006.
- [90] Yosef Yarden and Mark X. Sliwkowski. Untangling the ErbB signalling network. *Nature Reviews Molecular Cell Biology*, 2(2):127–137, February 2001.
- [91] D. A. Zaitsev. Decomposition-based calculation of Petri net invariants. In Corradella and Yakovlev, editors, *Proc. Workshop on Token based Computing (ToBaCo), Satellite Event of the 25-th International conference on application and theory of Petri nets*, pages 79–83, 2004.