A Language for Biochemical Systems

Michael Pedersen and Gordon Plotkin

LFCS, School of Informatics, University of Edinburgh

Abstract. CBS is a *Calculus of Biochemical Systems* intended to allow the modelling of metabolic, signalling and regulatory networks in a natural and modular manner. In this paper we extend CBS with features directed towards practical, large-scale applications, thus yielding LBS: a *Language for Biochemical Systems*. The two main extensions are expressions for modifying large complexes in a step-wise manner and parameterised modules with a notion of subtyping; LBS also has nested declarations of species and compartments. The extensions are demonstrated with examples from the yeast pheromone pathway. A formal specification of LBS is then given through an abstract syntax, static semantics and a translation to a variant of coloured Petri nets. Translation to other formalisms such as ordinary differential equations and continuous time Markov chains is also possible.

Keywords: The Calculus of Biochemical Systems, large-scale, parameterised modules, subtyping, coloured Petri nets.

1 Introduction

Recent years have seen a multitude of formal languages and systems applied to biology, thus gaining insight into the biological systems under study through analysis and simulation. Some of these languages have a history of applications in computer science and engineering, e.g. the pi calculus [26], PEPA [2], Petri nets [25] and P-systems [19], and some are designed from scratch, e.g. Kappa [4], BioNetGen [5], BIOCHAM [3], Bioambients [27], Beta binders [7,24], Dynamical Grammars [15] and the Continuous Pi Calculus [14].

The *Calculus of Biochemical Systems* (CBS) [23] is a new addition to the latter category which allows metabolic, signalling and regulatory networks to be modelled in a natural and modular manner. In essence, CBS describes reactions between modified complexes, occurring concurrently inside a hierarchy of compartments and with possible cross-compartment interactions and transport. It has a compositional semantics in terms of Petri nets, ordinary differential equations (ODEs) and continuous time Markov chains (CTMCs). Petri nets allow a range of established analysis techniques to be used in the biological setting [8], while ODEs and CTMCs enable respectively deterministic and stochastic simulations to be carried out.

This paper proposes extensions of CBS in support of practical, large-scale applications, resulting in LBS: a *Language for Biochemical Systems*. The two main

extensions are *pattern expressions* and *parameterised modules*. Patterns represent complexes and pattern expressions provide a concise way of making small changes to large complexes incrementally, a common scenario in signal transduction pathways. Parameterised modules allow general biological "gadgets", such as a MAPK cascade, to be modelled once and then reused in different contexts. Modules may be parameterised on compartments, rates, patterns and species types, the latter resulting in a notion of parametric type. A notion of subtyping of species types and patterns is also employed, allowing a module to specify general reaction schemes and have a concrete context provided at the time of module invocation. Modules may furthermore return pattern expressions, thus providing a natural mechanism for connecting related modules.

Other improvements over CBS include species and compartment declarations which involve scope and new name generation. While species modification sites in CBS always have boolean type, LBS does not place any limitations of modification site types, allowing, e.g., location (real number pairs) or DNA sequences (strings) to be represented. LBS also allows general rate expressions to be associated with expressions, although mass-action kinetics may be assumed as in CBS.

The syntax and semantics of CBS are outlined informally in Section 2 through some basic examples, including a MAPK cascade module drawn from the yeast pheromone pathway. In Section 3 we introduce LBS by further examples from the yeast pheromone pathway and demonstrate how the features of LBS can be used to overcome specific limitations of CBS. We then turn to a formal presentation of the language. An abstract syntax of LBS is given in Section 4. An overview of the semantics of LBS, including a static semantics, the general approach to translation, and a specific translation to Petri nets, is given in Section 5. Section 6 discusses related work and future directions. Due to space constraints, only selected parts of the semantics are given in this paper. A full presentation, together with the full LBS model of the yeast pheromone pathway, can be found in [21].

A compiler from LBS to the *Systems Biology Markup Language* (SBML) [10] has been implemented and supports the main features of the language presented formally in this paper. We have validated the compiler on the yeast pheromone pathway model by deriving ODEs from the target SBML using the Copasi tool [9]. By manual inspection, the derived ODEs coincide, up to renaming of species, with the ODEs published in [13], although there is some discrepancy between the simulation results.

2 The Calculus of Biochemical Systems

Basic examples of CBS modules with no modifications or complexes are shown in Program 1, with the last line informally indicating how the modules may be used in some arbitrary context. Module `M1` consists of two chemical reactions taking place in parallel as indicated by the bar, |. The first is a condensation reaction, and the second is a methane burning reaction. In `M2` the second reaction

Program 1 Example of three modules in CBS.

```
module M1 { 2 H2 + O2 -> 2 H2O | CH4 + 2 O2 -> CO2 + 2 H2O };
module M2 { 2 H2 + O2 -> 2 H2O | c[CH4 + 2 O2 -> CO2 + 2 H2O] };
module M3 { c[O2] -> O2 };
... | M1 | ... | M2 | ... | M3 | ...
```

takes place inside a compartment `c`, and in `M3` the species `O2` is transported out of compartment `c` to whatever compartment the module is later instantiated inside. In contrast, a language such as BIOCHAM would represent the same model by an unstructured list of five reactions and explicitly specifying the compartments of each species.

Graphical representations of the two individual reactions in module `M1` are shown in Figure 1a; the reader familiar with Petri nets (see [17] for an overview) can think of the pictures as such. When considering two reactions together, *in parallel*, the standard chemical interpretation is that the reactions share and compete for species which have syntactically identical names in the two reactions, in this case `O2` and `H2O`. A graphical Petri net representation of `M1` based on this interpretation is shown in Figure 1b. In the case of module `M2`, the species `O2` and `H2O` are not considered identical between the two reactions because they are located in different compartments. Consequently, none of the species are merged in the parallel composition that constitutes `M2`, see Figures 1c and 1d.

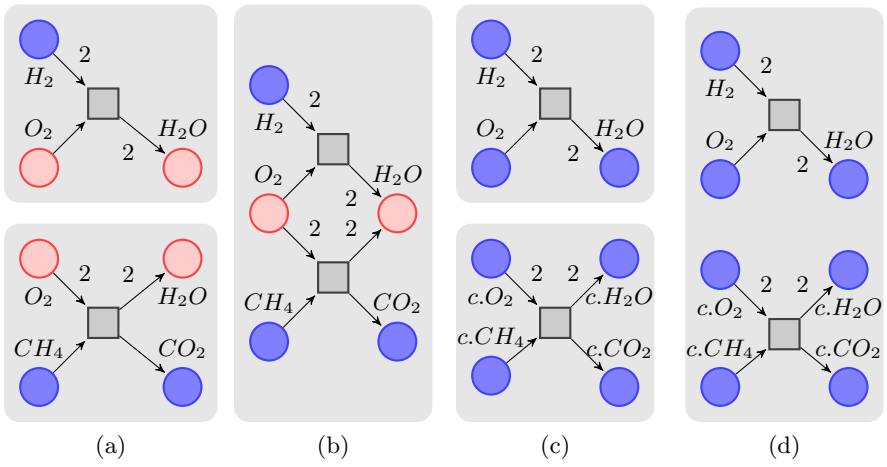

Fig. 1: Petri net representations of reactions and their parallel composition. Places (circles) represent species, transitions (squares) represent reactions, and arc weights represent stoichiometry.

This example illustrates how reactions, or more generally, modules, are composed in CBS, and hints at how a *compositional* semantics in terms of Petri nets can be defined. Similar ideas can be used to define compositional semantics in terms of ODEs and CTMCs, assuming that reactions are equipped with rates; see [23] for the full details.

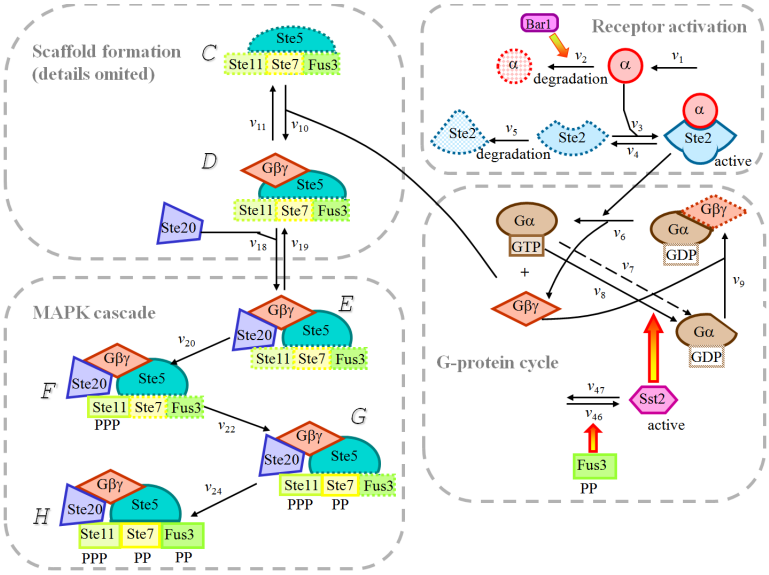

Fig. 2: Selected and reorganised parts of the informal model from [13] of the yeast pheromone pathway. Copyright © 2008 John Wiley & Sons Limited. Reproduced with permission.

Let us turn to a more realistic example featuring modifications and complexes, drawn from a model of the yeast pheromone pathway that will serve as a case study throughout the paper. Figure 2, adapted from [13], shows a graphical representation of the model divided into several interacting modules. We do not discuss the biological details of the model but rather consider the general structure and how this can be represented formally, and we start by focusing on the `MAPKCascade` module. The cascade relies on a scaffolding complex holding `Ste11` (the MAPK3), `Ste7` (the MAPK2) and `Fus3` (the MAPK) into place. Ignoring degradation, this module can be written in CBS as shown in Program 2.

The labels `k20`, `k22` and `k24` associated with reaction arrows represent the mass-action rates given in [13]. `Fus3`, `Ste7`, `Ste11`, `Ste5`, `Ste20` and `Gbg` are the names of *primitive species*, i.e. non-complex species, and can exist in various states of modification. In this case all primitive species except `Ste20` and `Gbg` have a single modification site, p, which can be either phosphorylated or unphosphorylated, indicated by assigning boolean values (**tt**/**ff**) to the sites. For

Program 2 A CBS module of the yeast MAPK cascade.

```
module MAPKCascade {
  Fus3{p=ff}-Ste7{p=ff}-Ste11{p=ff}-Ste5{p=ff}-Ste20-Gbg ->{k20}
              Fus3{p=ff}-Ste7{p=ff}-Ste11{p=tt}-Ste5{p=ff}-Ste20-Gbg |
  Fus3{p=ff}-Ste7{p=ff}-Ste11{p=tt}-Ste5{p=ff}-Ste20-Gbg ->{k22}
              Fus3{p=ff}-Ste7{p=tt}-Ste11{p=tt}-Ste5{p=ff}-Ste20-Gbg |
  Fus3{p=ff}-Ste7{p=tt}-Ste11{p=tt}-Ste5{p=ff}-Ste20-Gbg ->{k24}
              Fus3{p=tt}-Ste7{p=tt}-Ste11{p=tt}-Ste5{p=ff}-Ste20-Gbg
};
... | MAPKCascade | ...
```

example, `Fus3{p=ff}` represents `Fus3` in its unphosphorylated state. Complexes are formed by composing modified primitive species using a hyphen, -.

Complexes such as the above will generally be referred to as *patterns*. As in BIOCHAM, the term "pattern" reflects that modification sites in reactants can be assigned *variables* rather than actual boolean values, hence "matching" multiple physical complexes and thereby ameliorating the combinatorial explosion problem on the level of species modifications.

Two limitations of the CBS representation emerge from this example:

1. **Redundancy.** Many signalling pathways involve making small changes to large complexes. Therefore, patterns are often identical except for small changes in modification, but in CBS we are forced to write all patterns out in full.
2. **Reuse.** The MAPK cascade is a typical example of a "biological gadget" which is utilised in many different contexts but with different participating species [6]. The CBS MAPK module in our example "hard codes" the species and rates involved and hence cannot be used in another context.

The next section shows how LBS offers solutions to these limitations.

3 The Language for Biochemical Systems

We give three examples of LBS programs and informally explain their syntax. The first shows how species declarations and pattern expressions can be used to improve the yeast MAPK cascade module. The second shows how a general, reusable MAPK cascade module can be written by taking advantage of parameters and subtyping. The third shows how modules can communicate by linking an output pattern of a receptor activation module to an input pattern of a G-protein cycle module.

3.1 The Yeast MAPKCascade

Program 3 shows how the model in Program 2 can be re-written in LBS. The first difference is that all primitive species featuring in a program must be declared by

specifying their modification site names and types, if any. For example, `Fus3` has a single modification site named `p` of type **bool**, and `Ste20` has no modification sites. In general, arbitrarily many modification sites may be declared.

Program 3 Species declarations and pattern expressions in the yeast MAPKCascade module.

```
module YeastMAPKCascade {
  spec Fus3{p:bool}, Ste7{p:bool}, Ste11{p:bool}, Ste5{p:bool}, Ste20, Gbg;
  pat e = Fus3{p=ff}-Ste7{p=ff}-Ste11{p=ff}-Ste5{p=ff}-Ste20-Gbg;

  e ->{k20} e<Ste11{p=tt}> as f;
  f ->{k22} f<Ste7{p=tt}> as g;
  g ->{k24} g<Fus3{p=tt}> as h
};
... | YeastMAPKCascade; ...
```

The second difference is that we assign the first pattern to a pattern identifier called `e`. This identifier, and the ones that follow, correspond directly to the names given to complexes in Figure 2. We can then simply refer to `e` in the first reaction instead of writing out the full pattern. The product of the first reaction uses the pattern expression `e<Ste11{p=tt}>` to represent "everything in `e`, except that site `p` in `Ste11` is phosphorylated," and subsequently assigns the resulting pattern to a new identifier `f` which, in turn, is then used as a reactant of the second reaction, and so on. When using such *in-line* pattern declarations, reactions are separated by semi-colons (;) rather than the parallel composition (|), indicating that the order in which reactions are written matters.

The module can be invoked in some parallel context as indicated informally in the last line. Since, e.g., `Fus3` is declared locally, inside the module, multiple instances of the module would give rise to multiple, distinct instances of this species. If we prefer species to be shared between multiple instances of a module, they should either be declared globally or passed as parameters, as we see in the next subsection.

3.2 A General MAPK Cascade Module

The model in the previous subsection still suffers from a lack of reusability. From a more general perspective, a (scaffolded) MAPK cascade is a series of reactions operating on some potentially very big complex but which contains specific species serving the K3, K2 and K1 functions of the cascade. Each of these species must have at least one phosphorylation site (i.e. of boolean type) which in the general case could be be called `ps`. With this in mind, the K3, K2 and K1 become *species parameters* of the MAPK module. The scaffold complex containing these species becomes a pattern parameter which we will call `mk4`, indicating its role as an upstream initiator of the cascade. Reaction rates become

rate parameters. Program 4 shows how the resulting module can be written in LBS. The species parameters follow the structure of species declarations. But

Program 4 Defining a general, scaffolded MAPKCascade module.

```
module MAPKCascade( spec K1{ps:bool}, K2{ps:bool}, K3{ps:bool};
                            pat mk4 : K1-K2-K3; rate r1, r2, r3 ){
  mk4 ->{r1} mk4<K3{ps=tt}> as mk3;
  mk3 ->{r2} mk3<K2{ps=tt}> as mk2;
  mk2 ->{r3} mk2<K1{ps=tt}> as mk1
};
```

the pattern parameter is different: it provides a pattern identifier together with the *type* of the pattern. A pattern type simply represents the names of primitive species in the pattern: no more is needed to determine how the pattern may be used, because the types of the primitive species are defined separately.

In the context of the yeast model, we can simply use the MAPK cascade module by declaring the specific species of interest, defining the scaffold pattern, and passing these together with the rates as arguments to the module. This is shown in Program 5 which is semantically equivalent to Program 3, i.e. the two translate to the same Petri net. A closer investigation of this program tells

Program 5 Using the general MAPKCascade module.

```
spec Fus3{p:bool}, Ste7{p:bool}, Ste11{p:bool}, Ste5{p:bool}, Ste20, Gbg;
pat e = Fus3{p=ff}-Ste7{p=ff}-Ste11{p=ff}-Ste5{p=ff}-Ste20-Gbg;

MAPKCascade(Fus3{p:bool}, Ste7{p:bool}, Ste11{p:bool}, e, k20, k22, k24);
```

us that the pattern `e`, which is passed as an *actual* parameter, has the type `Fus3-Ste7-Ste11-Ste5-Ste20-Gbg`, namely the species contained in the pattern. The corresponding *formal* parameter has type `K1-K2-K3`, which is instantiated to `Fus3-Ste7-Ste11` through the species parameters. This works because the actual parameter type contains *at least* the species required by the formal parameter type. This is all the module needs to know, since these are the only species it is going to manipulate. We say that the type of the actual parameter is a *subtype* of the type of the formal parameter, and hence the module invocation is legal. A similar idea applies at the level of species parameters, although in this example the corresponding formal and actual species parameters have the same number of modification sites. Note that the names of corresponding modification sites need not be the same for actual and formal parameters. In this example, the formal parameters use `ps` while the actual parameters use `p`.

3.3 Receptor Activation and G-protein Cycle Modules

Our last example illustrates how modules can be linked together. If we look at the general structure of the yeast pheromone picture in Figure 2, we notice that many of the modules produce outputs which are passed on to subsequent modules: the scaffold formation module produces a scaffold which is passed on to the MAPK cascade module, and the receptor activation module produces a receptor-ligand complex (consisting of `Alpha` and `Ste2`) which is used to activate the G-protein cycle. The G-protein cycle in turn passes on a beta-gamma subunit.

In order to naturally represent these interconnections, LBS provides a mechanism for modules to return patterns. Program 6 shows this mechanism involving the receptor activation and G-protein cycle modules. The modules are named as in Figure 2. They are commented and should be self-explanatory, except perhaps for three points. Firstly, enzymatic reactions are represented using the tilde operator ($\sim$) with an enzyme (a pattern) on the left and a reaction on the right. Secondly, reversible reactions are represented using a double-arrow (<->) followed by rates for the forward and backward directions. Thirdly, the rate `v46` in the G protein cycle module is defined explicitly because it does not follow mass-action kinetics, and this is indicated by the use of square brackets around the forward rate of the reaction.

Let us consider how the two modules interface to each other. The receptor activation module takes parameters for the cytosol compartment and a pattern which degrades the pheromone. The latter has empty type, indicating that the module does not care about the contents of this pattern. The new feature is the last parameter, `rl` (short for *receptor-ligand* complex). This is an output pattern: it is defined in the body of the module and is made available when the module is invoked. This happens towards the end of Program 6 by first declaring the relevant species and the compartment `cytosol` with volume 1 inside the distinguished top level compartment `T`, which are then passed as actual parameters to the module. Note that a species is a special, non-modified and non-complex case of a pattern, and hence can be used as a pattern parameter. The last actual parameter is the output parameter identifier, here called `link`. This pattern identifier will be assigned the return pattern of the module (namely the receptor-ligand complex) and is then passed as the "activating" parameter for the G-protein cycle module. It follows that the ordering of module invocation matters, which as above is indicated by a semicolon rather than the parallel composition. In this particular example, output patterns have the empty type, but they could have arbitrary types and are subject to a subtyping mechanism similar to that of input patterns.

4 The Abstract Syntax of LBS

Having given an informal introduction to the main features of LBS and its *concrete* syntax, we now present its *abstract* syntax. This is shown in Table 1; each of the three main syntactic categories is explained further in the following subsections.

Program 6 Receptor activation and G-protein cycle modules

```
spec Fus3{p:bool};
module ReceptorAct(comp cyto, pat degrador, patout rl) {
  spec Alpha, Ste2{p: bool};
  (* pheromone and receptor degradation: *)
  degrador ~ Alpha ->{k1} | cyto[Ste2{p=ff}] ->{k5} |
  (* Receptor-ligand binding and degradation: *)
  Alpha + cyto[Ste2{p=ff}] <->{k2,k3} cyto[Alpha-Ste2{p=tt}] as rl;
  cyto[rl] ->{k4}
};
module GProtCycle(pat act, patout gbg) {
  spec Ga, Gbg, Sst2{p:bool};
  pat Gbga = Gbg-Ga-GDP;
  (* disassociation of G-protein complex: *)
  act ~ Gbga ->{k6} Gbg + Ga-GTP |
  (* ... and the G-protein cycle: *)
  Ga-GTP ->{k7} Ga-GDP |
  Sst2{p=tt} ~ Ga-GTP ->{k8} Ga-GDP |
  rate v46 = k46 * (Fus3{p=tt}^2 / (4^2 + Fus3{p=tt}^2));
  Fus3{p=tt} ~ Sst2{p=ff} <->[v46]{k47} Sst2{p=tt} |
  Ga-GDP + Gbg ->{k9} Gbga |
  pat gbg = Gbg; Nil
};
spec Bar1;
comp cytosol inside T vol 1.0;
ReceptorAct(cytosol, Bar1, pat link);
GProtCycle(link, pat link2);
(* rest of model ... *)
```

Table 1: The core abstract syntax of LBS: Pattern expressions and their types (top), programs (middle) and declarations (bottom).

$$PE ::= \beta \mid p \mid PE - PE' \mid PE\langle PE\rangle \mid PE.s \mid PE\backslash s$$
$$\beta ::= \{s_i \mapsto \alpha_i\} \quad \alpha ::= \{l_i \mapsto E_i\} \quad \tau ::= \{s_i \mapsto n_i\} \quad \sigma ::= \{l_i \mapsto \rho_i\}$$

$$P ::= \{LPE_i \mapsto n_i\} \xrightarrow{RE} \{LPE'_j \mapsto n'_j\} \textbf{ if } E_{\textbf{bool}}$$
$$\mid \mathbf{0} \mid P_1 | P_2 \mid c[P] \mid Decl; P \mid m(APars; \textbf{pat } \underline{p}); P$$
$$LPE ::= PE \mid c[LPE] \qquad APars ::= \underline{s:\sigma}; \underline{c}; \underline{PE}; \underline{RE}$$
$$RE ::= k \mid LPE \mid c \mid r(APars) \mid \log(RE) \mid RE \textbf{ aop } RE$$

$$Decl ::= \textbf{comp } c : c', v \mid \textbf{spec } s:\sigma \mid \textbf{pat } p = PE \mid \textbf{rate } r(FPars) = RE$$
$$\mid \textbf{module } m(FPars; \textbf{patout } \underline{p:\tau})\{P\}$$
$$FPars ::= \underline{\underline{s:l:\rho}}; \textbf{comp } \underline{c:c'}; \textbf{pat } \underline{p:\tau}; \textbf{rate } \underline{r}$$

4.1 Notation

Tuples $(x_1, \ldots, x_k)$ are written $\underline{x}$ when the specific elements are unimportant. The set of finite *multisets* over a set S is denoted by $\mathrm{FMS}(S)$, the total functions from S to the natural numbers which take value 0 for all but finite many elements of S. The *power set* of a set S is written 2^S. Partial finite *functions* f are denoted by finite indexed sets of pairs $\{x_i \mapsto y_i\}_{i \in I}$ where $f(x_i) = y_i$, and I is omitted if it is understood from the context. When the ith element of a list or an indexed set is referred to without explicit quantification in a premise or condition of a rule, the index is understood to be universally quantified over the index set I. The *domain of definition* of a function f is denoted by $\mathrm{dom}(f)$; the empty partial function is denoted by $\emptyset$. We write $f[g]$ for the update of f by a partial finite function g; for the sake of readability, we often abbreviate e.g. $f[\{x_i \mapsto y_i\}]$ by $f[x_i \mapsto y_i]$.

4.2 Pattern Expressions

In the abstract syntax for pattern expressions, s ranges over a given set $\textsc{Names}_\mathrm{s}$ of *species names*, l ranges over a given set $\textsc{Names}_\mathrm{mo}$ of *modification site names*, and p ranges over a given set $\textsc{Ident}_\mathrm{p}$ of *pattern identifiers*. The simplest possible pattern expression, a *pattern* β, maps species names to lists of modifications, thus allowing homomers be be represented.

Modifications in turn map modification site names to expressions E, which range over the set $\textsc{Exp} \triangleq \bigcup_{\rho \in \textsc{Types}_\mathrm{mo}} \textsc{Exp}_\rho$; here, $\textsc{Types}_\mathrm{mo}$ is a given set of modification site types, ranged over by ρ, and $\textsc{Exp}_\rho$ is a given set of expressions of type ρ. We assume that $\textsc{Types}_\mathrm{mo}$ contains the boolean type **bool** with values $\{\mathbf{tt}, \mathbf{ff}\}$. Expressions may contain *match variables* from a given set X and we assume a function $\mathrm{FV} \triangleq \textsc{Exp} \to 2^X$ giving the variables of an expression.

Pattern composition, $PE - PE'$, intuitively results in a pattern where the lists of modifications from the first pattern have been appended to the corresponding lists of modifications from the second pattern. This operation is therefore *not* generally commutative, e.g. `s{l=tt}-s{l=ff}` is not the same as `s{l=ff}-s{l=tt}`. This is not entirely satisfying and will be a topic of future work. We have already encountered the pattern update expression $PE\langle PE \rangle$ in the MAPK cascade module in Program 3. The expression $PE.s$ restricts the pattern to the species s and throws everything else away, while the expression $PE \backslash s$ keeps everything except for s. So a dissociation reaction of s from PE can be written (in the concrete syntax) as `PE ->{r} PE\s + PE.s`. If there are multiple instances of `s` in `PE`, only the first in the list of modifications will be affected.

Finally, α and β represent the types of species and patterns, respectively; in the grammar, n ranges over $\mathbb{N}_{>0}$. Henceforth we let $\textsc{Types}_\mathrm{p}$ and $\textsc{Types}_\mathrm{s}$ be the sets generated by these productions.

4.3 Programs

In the abstract syntax for programs, c ranges over a given set $\textsc{Names}_{\text{c}}$ of *compartment names*, m ranges over a given set $\textsc{Ident}_{\text{m}}$ of *module identifiers*, r ranges over a given set $\textsc{Ident}_{\text{r}}$ of *rate identifiers*, $n \in \mathbb{N}_{>0}$ and $k \in \mathbb{R}$. The first line in the grammar for programs is a reaction. Products and reactants are represented as functions mapping located patterns to stoichiometry. A reaction may furthermore be conditioned on a boolean expression. We encountered located patterns in Programs 1 and 6: they are just patterns inside a hierarchy of compartments. Rate expressions associated with reactions can employ the usual arithmetic expressions composed from operators $\mathbf{aop} \in \{+, \texttt{-}, \texttt{*}, /, \hat{}\}$. But they can also include located patterns (which refer to either a population or a concentration, depending on the semantics), compartments (which refer to a volume) and rate function invocations.

In the second line, **0** represents the null process, and we have encountered parallel composition and compartmentalised programs in the examples. Declarations are defined separately in the next subsection. Module invocations are followed sequentially by a program since a new scope will be created if patterns are returned from modules as in Program 6.

Enzymatic reactions, reversible reactions, mass-action kinetics and inline pattern declarations (using the `as` keyword) which we encountered in the examples are not represented in the abstract syntax. They can all be defined, in a straightforward manner, from existing constructs. Take for example the following definitions of reactions from Programs 3 and 6:

- `e ->{k10} e<Ste11{p=tt}> as f` $\triangleq$
  pat `f = e<Ste11{p=tt}>; e ->[k10 * e] f`.
- `Fus3{act=tt}` $\sim$ `Sst2{act=ff}` `<->[v46]{k47} Sst2{act=tt}` $\triangleq$
  `Fus3{act=tt} + Sst2{act=ff} ->[v46] Fus3{act=tt} + Sst2{act=tt}`
  `| Sst2{act=tt} ->[k47 * Sst2{act=tt}] Sst2{act=ff}`

4.4 Declarations

Compartment declarations specify the volume $v \in \mathbb{R}_{>0}$ and parent of the declared compartment. Compartments which conceptually have no parent compartment should declare the "top level" compartment, $\top$, as their parent, so we assume that $\top \in \textsc{Names}_{\text{c}}$. Modules and rate functions may be parameterised on species with their associated type, compartments with their declared parents, patterns with their associated type, and finally on rate expressions. Henceforth we let FormalPars be the set of formal parameter expressions, as generated by the *FPars* production.

5 The Semantics of LBS

Having introduced the syntactic structure of LBS programs, we now turn to their *meaning*.

5.1 Static Semantics

The static semantics tells us which of the LBS programs that are well-formed according to the abstract syntax are also semantically meaningful. This is specified formally by a type system of which the central parts are given in Appendix A, and full details can be found in [21]. In this subsection we informally discuss the main conditions for LBS programs to be well-typed.

The type system for pattern expressions checks that pattern identifiers and species are declared and used according to their declared type. For pattern updates, selections and removals, only species which are present in the target pattern expression are allowed. If a pattern expression is well-typed, its type τ is deduced in a compositional manner. Suptyping for patterns is given by multiset inclusion, and subtyping for species is given by record subtyping as in standard programming languages [22].

For reactions, the type system requires that any match variables which occur in the products also occur in the reactants. This is because variables in the product patterns must be instantiated based on matches in the reactant patterns during execution of the resulting model. It also requires that the reactant and product located patterns agree on parent compartments; for example, the reaction `c1[s] -> c2[s]` is not well-typed if the compartments `c1` and `c2` are declared with different parent compartments. A similar consideration applies for parallel composition. For a compartment program `c[P]` we require that all top-level compartments occurring in `P` are declared with parent `c`. In order for these conditions on compartments to be checked statically, i.e. at time of module declaration rather than invocation, the type system must associate parent compartments with programs in a bottom-up manner as detailed in Appendix A.

For module invocations, the type system first of all ensures that the number of formal and actual parameters match. It also ensures that the type of actual species and pattern parameters are subtypes of the corresponding formal parameter types, both for standard "input" parameters but also for output pattern parameters. Finally, all formal output patterns identifiers must be defined in the body of the module.

5.2 The General Translation Framework

A key advantage of formal modelling languages for biology is that they facilitate different kinds of analysis on the same model. This is also the case for CBS which is endowed with compositional semantics in terms of Petri nets, ODEs and CTMCs. The semantics of LBS is complicated by the addition of high-level constructs such as pattern expressions, modules and declarations, but this is ameliorated by the fact that the definition of its semantics is to some extent independent of the specific choice of target semantical objects. The *general translation framework* defines these independent parts of the translation, and concrete translations then tie into this framework by defining the semantic objects associated with the following:

1. *Normal form* reactions, where pattern expressions have been evaluated to patterns, together with the types of species featuring in the reaction.
2. The **0** program.
3. The parallel composition of semantic objects.
4. Semantic objects inside compartments.

Here we only outline the central mechanisms involved; a complete account can be found in [21].

The framework evaluates pattern expressions PE to patterns β according to the intuitions set forth in Section 4.2 and by replacing pattern identifiers with their defined patterns. Located pattern expressions are evaluated to pairs of (immediate) parent compartments and the resulting patterns. Reactions are then evaluated to *normal form* reactions of the form:

$$\{(c_i^{\text{in}}, \beta_i^{\text{in}}) \mapsto n_i^{\text{in}}\} \xrightarrow{RE} \{(c_j^{\text{out}}, \beta_j^{\text{out}}) \mapsto n_j^{\text{out}}\} \textbf{ if } E_{\textbf{bool}}$$

where pattern expressions and rate function invocations in RE have been evaluated, and compartment identifiers have been replaced by their declared volumes. A *species type environment* $\Gamma_{\text{s}} : \textsc{Names}_{\text{s}} \hookrightarrow_{\text{fin}} \textsc{Types}_{\text{s}}$ recording the type of species in reactions is also maintained by the framework.

Modules are evaluated to semantic functions which, given the relevant actual parameters at time of invocation, return the semantic objects of the module body together with the output patterns. The definition of these functions is complicated by the need to handle species parameters and pattern subtyping: the former requires renaming to be carried out, and the latter entails handling type environments for species that are not necessarily within the scope of their declaration.

As an example, using the general translation framework on the module in Program 4 results in a function which, when invoked with the parameters given in Program 5 , computes the (almost) normal form reactions in Program 2, uses the first concrete semantics function to obtain concrete semantic objects for each reaction, and finally applies the third concrete semantic function to obtain the final result of the parallel compositions.

5.3 Translating LBS to Petri nets

This subsection demonstrates the translation framework by giving the definitions needed for a concrete translation to Petri nets. Translations to ODEs or CTMCs are also possible, see [23] or [8] for the general approach.

Petri nets When modelling biological systems with Petri nets, the standard approach is to represent species by places, reactions by transitions and stoichiometry by arc multiplicities as in Figure 1. Complex species with modification sites can be represented compactly using a variant of *coloured Petri nets* where places are assigned *colour types* [11]. In our case, the colour types of places are given

by pairs (c, τ) of compartments and pattern types together with a global primitive species type environment Γ_{s}. Then a pair (c, τ) uniquely identifies a located species, so there is no need to distinguish places and colour types. Arcs are equipped with multisets of *patterns* rather than plain stoichiometry, allowing a transition to restrict the colour of tokens (e.g. modification of a species) that it accepts or produces. Transitions are strings over the binary alphabet. This enables us to ensure that the transitions from two parallel nets are disjoint simply by prefixing 0 and 1 to all transitions in the respective nets.

In the following formal definition of *bio-Petri nets*, we shall need the sets of patterns conforming to specific types (here a type system $E : \rho$ on expressions is assumed):

$$\begin{aligned}
\textsc{Patterns}_{\tau,\Gamma_{\mathrm{s}}} &\triangleq \{\beta \in \textsc{Patterns} \mid \\
&\qquad \mathrm{type}(\beta) = \tau \wedge \forall s \in \mathrm{dom}(\beta).\mathrm{type}(\beta(s)) = \Gamma_{\mathrm{s}}(s)\} \\
\mathrm{type}(\{s_i \mapsto \underline{\alpha}_i\}) &\triangleq \{s_i \mapsto |\underline{\alpha}_i|\} \\
\mathrm{type}(\{l_i \mapsto E_i\}) &\triangleq \{l_i \mapsto \rho_i\} \text{ where } E_i : \rho_i
\end{aligned}$$

We use a standard notation $\prod_{i \in I} X_i$ for dependent sets.

Definition 1. *A bio-Petri net $\mathcal{P}$ is a tuple $(S, T, F_{\mathrm{in}}, F_{\mathrm{out}}, B, \Gamma_{\mathrm{s}})$ where*

- $S \subset \textsc{Names}_{\mathrm{c}} \times \textsc{Types}_{\mathrm{p}}$ *is a finite set of places (located pattern types).*
- $T \subset \{0,1\}^*$ *is a finite set of transitions (reactions).*
- $F_{\mathrm{in}} : \prod_{t,(c,\tau) \in T \times S} \mathrm{FMS}(\textsc{Patterns}_{\tau,\Gamma_{\mathrm{s}}})$ *is the* flow-in *function (reactants).*
- $F_{\mathrm{out}} : \prod_{t,(c,\tau) \in T \times S} \mathrm{FMS}(\textsc{Patterns}_{\tau,\Gamma_{\mathrm{s}}})$ *is the* flow-out *function (products).*
- $B : T \to \textsc{Exp}_{\mathbf{bool}}$ *is the* transition guard *function.*
- $\Gamma_{\mathrm{s}} : \bigcup\{\mathrm{dom}(\tau) \mid (c, \tau) \in S\} \to \textsc{Types}_{\mathrm{s}}$ *is the* species type *function.*

We use the superscript notation $S^{\mathcal{P}}$ to refer to the places S of Petri net $\mathcal{P}$, and similarly for the other Petri net elements. For a formal definition of behaviour (qualitative semantics) of bio-Petri nets, please refer to [21].

The Concrete Translation of LBS to Bio-Petri nets Following [23], the concrete translation to bio-Petri nets is given by the following four definitions.

1. Let $P = \{(c_i^{\mathrm{in}}, \beta_i^{\mathrm{in}}) \mapsto n_i^{\mathrm{in}}\} \xrightarrow{RE} \{(c_j^{\mathrm{out}}, \beta_j^{\mathrm{out}}) \mapsto n_j^{\mathrm{out}}\}$ **if** $E_{\mathbf{bool}}$ be an LBS reaction in normal form and let Γ_{s} be a species type environment. Then define $\mathcal{P}(P, \Gamma_{\mathrm{s}})$ as follows, where ϵ denotes the empty string:

   - $S^{\mathcal{P}} \triangleq \{(c_i^{\mathrm{in}}, \mathrm{type}(\beta_i^{\mathrm{in}}))\} \cup \{(c_j^{\mathrm{out}}, \mathrm{type}(\beta_j^{\mathrm{out}}))\}$
   - $T^{\mathcal{P}} \triangleq \{\epsilon\}$
   - $F^{\mathcal{P}}_{\mathrm{io}}(\epsilon, (c, \tau)) \triangleq \{(\beta_h^{\mathrm{io}} \mapsto n_h^{\mathrm{io}}) \mid c_h^{\mathrm{io}} = c \wedge \mathrm{type}(\beta_h^{\mathrm{io}}) = \tau)$ for $\mathrm{io} \in \{\mathrm{in}, \mathrm{out}\}$
   - $B^{\mathcal{P}}(\epsilon) \triangleq E_{\mathbf{bool}}$
   - $\Gamma_{\mathrm{s}}^{\mathcal{P}} \triangleq \Gamma_{\mathrm{s}}$

2. Define $\mathcal{P}(\mathbf{0}) \triangleq (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
3. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be Petri nets with $\Gamma_{\mathrm{s}}^{\mathcal{P}_1}(s) = \Gamma_{\mathrm{s}}^{\mathcal{P}_2}(s)$ for all $s \in \mathrm{dom}(\Gamma_{\mathrm{s}}^{\mathcal{P}_1}) \cap \mathrm{dom}(\Gamma_{\mathrm{s}}^{\mathcal{P}_2})$. Define parallel composition $\mathcal{P} = \mathcal{P}_1 | \mathcal{P}_2$ as follows, where $b \in \{0, 1\}$:
   - $S^{\mathcal{P}} \triangleq S^{\mathcal{P}_1} \cup S^{\mathcal{P}_2}$
   - $T^{\mathcal{P}} \triangleq \{0t \mid t \in T^{\mathcal{P}_1}\} \cup \{1t \mid t \in T^{\mathcal{P}_2}\}$
   - $F_{\mathrm{io}}^{\mathcal{P}}(bt, p) \triangleq \begin{cases} F_{\mathrm{io}}^{\mathcal{P}_1}(t,p) & \text{if } t \in T^{\mathcal{P}_1} \wedge p \in S^{\mathcal{P}_1} \\ F_{\mathrm{io}}^{\mathcal{P}_2}(t,p) & \text{if } t \in T^{\mathcal{P}_2} \wedge p \in S^{\mathcal{P}_2} \\ \emptyset & \text{otherwise} \end{cases}$ for $\mathrm{io} \in \{\mathrm{in}, \mathrm{out}\}$
   - $B(bt) \triangleq \begin{cases} B^{\mathcal{P}_1}(t) & \text{if } t \in T^{\mathcal{P}_1} \\ B^{\mathcal{P}_2}(t) & \text{if } t \in T^{\mathcal{P}_2} \end{cases}$
   - $\Gamma_{\mathrm{s}}^{\mathcal{P}} \triangleq \Gamma_{\mathrm{s}}^{\mathcal{P}_1} \cup \Gamma_{\mathrm{s}}^{\mathcal{P}_2}$
4. First define $c[(c', \tau)] \triangleq \begin{cases} (c, \tau) & \text{if } c' = \top \\ (c', \tau) & \text{otherwise} \end{cases}$
   Let $\mathcal{P}'$ be a Petri net. Then define the compartmentalisation $\mathcal{P} = c[\mathcal{P}']$ as follows:
   - $S^{\mathcal{P}} \triangleq \{c[p] \mid p \in S^{\mathcal{P}'}\}$
   - $T^{\mathcal{P}} \triangleq T^{\mathcal{P}'}$
   - $F_{\mathrm{io}}^{\mathcal{P}}(t, c[p]) \triangleq F_{\mathrm{io}}^{\mathcal{P}'}(t, p)$ for $\mathrm{io} \in \{\mathrm{in}, \mathrm{out}\}$
   - $B^{\mathcal{P}}(t) \triangleq B^{\mathcal{P}'}(t)$
   - $\Gamma_{\mathrm{s}}^{\mathcal{P}} \triangleq \Gamma_{\mathrm{s}}^{\mathcal{P}'}$

Observe that for programs where species have no modification sites and do not form complexes, the above definitions collapse to the simple cases of composition illustrated in Program 1 and Figure 1 for standard Petri nets.

6 Related Work and Future Directions

Compared to the other languages for biochemical modelling mentioned in the introduction, CBS is unique in its combination of two features: it explicitly models *reactions* rather than individual agents as in process calculi, and it does so in a *compositional* manner. BIOCHAM, for example, also models reactions explicitly using a very similar syntax to that of CBS, but it does not have a modular structure.

Whether the explicit modelling of reactions is desirable or not depends on the particular application. Systems which are characterised by high combinatorial complexity arising from complex formations are difficult or even impossible to model in CBS and LBS; an example is a model of scaffold formation which considers all possible orders of subunit assembly, where one may prefer to use Kappa or BioNetGen. But for systems in which this is not an issue, or where the combinatorial complexity arises from modifications of simple species (which CBS and LBS deal with in terms of match variables), the simplicity of a reaction-based approach is attractive. It also corresponds well to graphical representations of biological systems, as we have seen with the yeast pheromone pathway example.

To our knowledge, no other languages have abstractions corresponding to the pattern expressions and nested declarations of species and compartments of LBS. The notion of parameterised modules is however featured in the *Human-Readable Model Definition Language* [1], a draft textual language intended as a front-end to the *Systems Biology Markup Language* (SBML) [10]. The modules in this language follow an object-oriented approach rather than our functional approach, but there is no notion of subtyping or formal semantics.

Tools for visualising LBS programs and, conversely, for generating LBS programs from visual diagrams, are planned and will follow the notation of [12,16] or the emerging *Systems Biology Graphical Notation* (SBGN). We also plan to use LBS for modelling large scale systems such as the EGFR signalling pathway [18], although problems with interpretation of the graphical diagrams are anticipated. While some parts of the EGFR map are well characterised by modules, others appear rather monolithic and this is likely to be a general problem with modular approaches to modelling in systems biology. In the setting of *synthetic* biology, however, systems are *programmed* rather than *modelled*, so it should be possible to fully exploit modularity there.

With respect to language development, it is important to achieve a better understanding of homomers, enabling a commutative pattern composition operation. We also anticipate the addition of descriptive features for annotating species with e.g. Gene Ontology (GO) or Enzyme Commission (EC) numbers. One may also consider whether model analyses can exploit modularity, e.g. for Petri net invariants. Results for a subset of CBS without complexes and modifications, corresponding to plain place/transition nets, can be found in [20], and extensions to LBS and coloured Petri nets would be of interest.

Acknowledgements

The authors would like to thank Vincent Danos and Stuart Moodie for useful discussions, and Edda Klipp for supplying the diagram in Figure 2. This work was supported by Microsoft Research through its European PhD Scholarship Programme and by a Royal Society-Wolfson Award. The second author is grateful for the support from The Centre for Systems Biology at Edinburgh, a Centre for Integrative Systems Biology funded by BBSRC and EPSRC, reference BB/D019621/1. Part of the research was carried out at the Microsoft Silicon Valley Research Center.

References

1. F. Bergmann and H. Sauro. Human-readable model definition language (first draft, revision 2). Technical report, Keck Graduate Institute, 2006.
2. M. Calder et al. 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.
3. N. Chabrier-Rivier et al. The biochemical abstract machine BIOCHAM. In V. Danos and V. Schächter, editors, *Proc. CMSB*, volume 3082 of *LNCS*, pages 172–191. Springer, 2004.

4. V. Danos et al. 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.
5. J. R. Faeder et al. 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.
6. T. P. Garrington and G. L. Johnson. Organization and regulation of mitogen-activated protein kinase signaling pathways. *Current Opinion in Cell Biology*, 11:211–218, 1999.
7. M. L. Guerriero et al. 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.
8. M. Heiner et al. 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.
9. S. Hoops et al. COPASI – a COmplex PAthway SImulator. *Bioinformatics*, 22(24):3067–74, 2006.
10. M. 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.
11. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer, 1992.
12. H. Kitano et al. Using process diagrams for the graphical representation of biological networks. *Nat Biotechnol*, 23(8):961–966, 2005.
13. B. Kofahl and E. Klipp. Modelling the dynamics of the yeast pheromone pathway. *Yeast*, 21(10):831–850, 2004.
14. M. Kwiatkowski and I. Stark. The continuous pi-calculus: a process algebra for biochemical modelling. In M. Heiner and A. M. Uhrmacher, editors, *Proc. CMSB*, LNCS. Springer, 2008.
15. E. Mjolsness and G. Yosiphon. Stochastic process semantics for dynamical grammars. *Ann. Math. Artif. Intell.*, 47(3-4):329–395, 2006.
16. S. L. Moodie et al. A graphical notation to describe the logical interactions of biological pathways. *Journal of Integrative Bioinformatics*, 3(2), 2006.
17. T. Murata. Petri nets: properties, analysis and applications. *Proc. IEEE*, 77(4):541–580, 1989.
18. K. Oda et al. A comprehensive pathway map of epidermal growth factor receptor signaling. *Molecular Systems Biology*, 2005.
19. G. Paun and G. Rozenberg. A guide to membrane computing. *Theor. Comput. Sci.*, 287(1):73–100, 2002.
20. M. Pedersen. Compositional definitions of minimal flows in Petri nets. In M. Heiner and A. M. Uhrmacher, editors, *Proc. CMSB*, LNCS. Springer, 2008.
21. M. Pedersen and G. Plotkin. A language for biochemical systems. Technical report, University of Edinburgh, 2008. http://www.inf.ed.ac.uk/publications/report/1270.html.
22. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
23. G. Plotkin. A calculus of biochemical systems. In preparation.
24. 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.
25. V. N. Reddy et al. Petri net representation in metabolic pathways. In *Proc. Int. Conf. Intell. Syst. Mol. Biol.*, pages 328–336, 1993.

26. A. Regev et al. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.
27. A. Regev et al. BioAmbients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

A The Type System for LBS

This section presents the central parts of the type system for LBS.

A.1 Type Environments

The type system relies on the following type environments corresponding to each of the possible declarations in the abstract syntax.

- Γ_s : $\textsc{Names}_\mathrm{s} \hookrightarrow_\mathrm{fin} \textsc{Types}_\mathrm{s}$ for species declarations.
- Γ_c : $\textsc{Names}_\mathrm{c} \hookrightarrow_\mathrm{fin} \textsc{Names}_\mathrm{c}$ for compartment declarations.
- Γ_p : $\textsc{Ident}_\mathrm{p} \hookrightarrow_\mathrm{fin} \textsc{Types}_\mathrm{p}$ for pattern declarations.
- Γ_r : $\textsc{Ident}_\mathrm{r} \hookrightarrow_\mathrm{fin} \textsc{FormalPars}$ for rate function declarations.
- Γ_m : $\textsc{Ident}_\mathrm{m} \hookrightarrow_\mathrm{fin} \textsc{FormalPars} \times \textsc{Types}_\mathrm{p}^* \times 2^{\textsc{Names}_\mathrm{s}}$ for module declarations.

The Γ_m environment stores the type of defined modules which includes a set of compartments in which the module may be legally instantiated according to the compartment hierarchy specified through compartment declarations. For a well-typed module this set will either be a singleton compartment, indicating that the module can only be instantiated in a particular compartment, or the entire set of compartment names indicating that the module can be instantiated inside any compartment. The type system also needs to check that output patterns are used appropriately. For this purpose, two additional environments are required.

- $\Gamma_\mathrm{p}^\mathrm{out}$: $\textsc{Ident}_\mathrm{p} \hookrightarrow_\mathrm{fin} \textsc{Types}_\mathrm{p}$. This stores the declared types of module output patterns.
- $R_\mathrm{p} \in 2^{\textsc{Ident}_\mathrm{p}}$. This records pattern identifiers which have an entry in the $\Gamma_\mathrm{p}^\mathrm{out}$ environment *and* are defined inside a module.

The R_p environment is necessary to ensure that a module does in fact define the pattern identifiers which it has declared as outputs.

The type systems to be given in the following use and modify the above type environments. In general, type judgements for a given type system may have the form $\Gamma_\mathrm{s}, \Gamma_\mathrm{c}, \Gamma_\mathrm{p}, \Gamma_\mathrm{r}, \Gamma_\mathrm{m} \vdash \ldots$ but for the sake of readability we may rather write, e.g., $\Gamma, \Gamma_\mathrm{p} \vdash \ldots$ where Γ represents all environments to the left of the turnstile except Γ_p.

Table 2: The type system for patterns.

$$\textsc{STSpec}\quad \frac{\mathrm{dom}(\sigma') \subseteq \mathrm{dom}(\sigma) \wedge \forall l \in \mathrm{dom}(\sigma').\sigma(l) = \sigma'(l)}{\sigma <: \sigma'}$$

$$\textsc{STPat}\quad \frac{\forall s \in \textsc{Names}_{\mathrm{s}}.\tau(s) \geq \tau'(s)}{\tau <: \tau'}$$

$$\textsc{TPat}\quad \frac{E_{ij} : \rho_{ij} \text{ and } \rho_{ij} = \Gamma_{\mathrm{s}}(s)(l_{ij})}{\Gamma, \Gamma_{\mathrm{s}} \vdash \{s_i \mapsto \underline{\{l_i \mapsto E_i\}}\} : \{s \mapsto |\underline{\{l_i \mapsto E_i\}}|\}}$$

$$\textsc{TPatComp}\quad \frac{\Gamma \vdash PE : \tau \text{ and } \Gamma \vdash PE' : \tau'}{\Gamma \vdash PE - PE' : \tau + \tau'}$$

$$\textsc{TPatUpd}\quad \frac{\Gamma \vdash PE : \tau \text{ and } \Gamma \vdash PE' : \tau'}{\Gamma \vdash PE\langle PE' \rangle : \tau}\ \tau <: \tau'$$

$$\textsc{TPatSel}\quad \frac{\Gamma \vdash PE : \tau}{\Gamma \vdash PE.s : \{s \mapsto 1\}}\ s \in \mathrm{dom}(\tau)$$

$$\textsc{TPatRem}\quad \frac{\Gamma \vdash PE : \tau + \{s \mapsto 1\}}{\Gamma \vdash PE \backslash s : \tau} \qquad \textsc{TPatIden}\quad \frac{}{\Gamma, \Gamma_{\mathrm{p}} \vdash p : \Gamma_{\mathrm{p}}(p)}$$

A.2 Pattern Expressions

Subtyping is formalised by the first two rules in Table 2. The remaining rules use the environments for primitive species and patterns, so judgements are of the form $\Gamma_{\mathrm{s}}, \Gamma_{\mathrm{p}} \vdash PE : \tau$. The first rule furthermore relies on a type system (not given here) for basic expressions with judgements of the form $E : \rho$. We use pattern types as multisets with the obvious extension to $\mathbb{N}$ and the standard multiset operation $+$.

A.3 Programs

The type system for programs in Table 3 tells us if a program is well-typed and, if so, gives the compartments where a program can legally reside according to the compartment hierarchy specified in compartment declarations. Judgements are of the form $\Gamma_{\mathrm{s}}, \Gamma_{\mathrm{c}}, \Gamma_{\mathrm{p}}, \Gamma_{\mathrm{r}}, \Gamma_{\mathrm{m}}, \Gamma_{\mathrm{p}}^{\mathrm{out}} \vdash P : \gamma \dashv R_{\mathrm{p}}$ where $\gamma \in 2^{\textsc{Names}_{\mathrm{c}}}$, and thus rely on all available environments. Rules for rate expressions, located patterns and declarations are omitted.

The conditions on compartment hierarchies are enforced in the TReac rule by using standard set theoretic notation, and the condition on free match variables uses the obvious extension of FV to located pattern expressions. The TPar rule imposes similar conditions on compartments, and also ensures that parallel compartments do not define the same return pattern identifiers. The declaration rule, TDec, relies on the declaration type system which is omitted. If the declaration is a pattern flagged for return, the pattern identifier will be in R_{p}.

Module invocation is checked by the final rule, TModInv which relies on a separate rule for checking that actual and formal parameters match. The first

Table 3: The type system for programs.

TReac
$$\frac{\Gamma_\mathrm{s}, \Gamma_\mathrm{c}, \Gamma_\mathrm{p} \vdash LPE_i : \tau_i, \gamma_i \text{ and } \Gamma_\mathrm{s}, \Gamma_\mathrm{c}, \Gamma_\mathrm{p}, \Gamma_\mathrm{r} \vdash RE \text{ and } \Gamma_\mathrm{s}, \Gamma_\mathrm{c}, \Gamma_\mathrm{p} \vdash LPE'_j : \tau'_j, \gamma'_j}{\Gamma, \Gamma_\mathrm{s}, \Gamma_\mathrm{c}, \Gamma_\mathrm{p}, \Gamma_\mathrm{r} \vdash \{LPE_i \mapsto n_i\} \xrightarrow{RE} \{LPE'_j \mapsto n'_j\} \textbf{ if } E_{\textbf{bool}} : \gamma'' \dashv \emptyset}$$
if 1) $\gamma'' \neq \emptyset$ and 2) $\bigcup \mathrm{FV}(LPE'_j) \subseteq \bigcup \mathrm{FV}(LPE_i)$ and
3) $\mathrm{FV}(E_{\textbf{bool}}) \subseteq \bigcup \mathrm{FV}(LPE'_j) \cup \bigcup \mathrm{FV}(LPE_i)$ where $\gamma'' = \bigcap\{\gamma_i\} \cap \bigcap\{\gamma'_j\}$

TPar
$$\frac{\Gamma \vdash P_1 : \gamma \dashv R_\mathrm{p} \text{ and } \Gamma \vdash P_2 : \gamma' \dashv R'_\mathrm{p}}{\Gamma \vdash P_1 \mid P_2 : \gamma \cap \gamma' \dashv R_\mathrm{p} \cup R'_\mathrm{p}} \quad \gamma \cap \gamma' \neq \emptyset \text{ and } R_\mathrm{p} \cap R'_\mathrm{p} = \emptyset$$

TNil
$$\frac{}{\Gamma \vdash \mathbf{0} : \textsc{Names}_\mathrm{c} \dashv \emptyset}$$

TComp
$$\frac{\Gamma, \Gamma_\mathrm{c} \vdash P : \gamma' \dashv R_\mathrm{p}}{\Gamma, \Gamma_\mathrm{c} \vdash c[P] : \{\Gamma_\mathrm{c}(c)\} \dashv R_\mathrm{p}} \quad c \in \gamma'$$

TDec
$$\frac{\Gamma \vdash Decl \dashv \Gamma', R_\mathrm{p} \text{ and } \Gamma' \vdash P : \gamma \dashv R'_\mathrm{p}}{\Gamma \vdash Decl; P : \gamma \dashv R_\mathrm{p} \cup R'_\mathrm{p}} \quad R_\mathrm{p} \cap R'_\mathrm{p} = \emptyset$$

TModInv
$$\frac{\Gamma, \Gamma_\mathrm{c}, \Gamma_\mathrm{p}, \Gamma_\mathrm{m}, \Gamma_\mathrm{p}^{\mathrm{out}} \vdash APars <: FPars \text{ and } \Gamma, \Gamma_\mathrm{c}, \Gamma'_\mathrm{p}, \Gamma_\mathrm{m}, \Gamma_\mathrm{p}^{\mathrm{out}} \vdash P : \gamma'' \dashv R''_\mathrm{p}}{\Gamma, \Gamma_\mathrm{c}, \Gamma_\mathrm{p}, \Gamma_\mathrm{m}, \Gamma_\mathrm{p}^{\mathrm{out}} \vdash m(APars; \textbf{pat } \underline{p^{\mathrm{o}\prime}}); P : \gamma' \cap \gamma'' \dashv R'_\mathrm{p} \cup R''_\mathrm{p}}$$
if 1) $\gamma' \cap \gamma'' \neq \emptyset$ and 2) $R'_\mathrm{p} \cap R''_\mathrm{p} = \emptyset$ and 3) $|\underline{p^{\mathrm{o}}}| = |\underline{p^{\mathrm{o}\prime}}|$ and
4) if $p_i^{\mathrm{o}\prime} \in \mathrm{dom}(\Gamma_\mathrm{p}^{\mathrm{out}})$ then $\tau_i^{\mathrm{o}\prime} <: \Gamma_\mathrm{p}^{\mathrm{out}}(p_i^{\mathrm{o}\prime})$ where
$\Gamma'_\mathrm{p} \triangleq \Gamma_\mathrm{p}[p_i^{\mathrm{o}\prime} \mapsto \tau_i^{\mathrm{o}\prime}]$
$\tau_i^{\mathrm{o}\prime} \triangleq \tau_i^{\mathrm{o}}\Theta$ where $\Theta \triangleq \{s_i \mapsto s'_i\}$
$\gamma' \triangleq \gamma\Theta$ where $\Theta \triangleq \{c_i \mapsto c'_i, c''_i \mapsto \Gamma_\mathrm{c}(c_i)\}$
$R'_\mathrm{p} = \{p_i^{\mathrm{o}\prime} \mid p_i^{\mathrm{o}\prime} \in \mathrm{dom}(\Gamma_\mathrm{p}^{\mathrm{out}})\}$
$(FPars, \underline{\tau^{\mathrm{o}}}, \gamma) \triangleq \Gamma_\mathrm{m}(m)$
$\textbf{spec } \underline{s : \underline{\underline{l : \rho}}}; \textbf{comp } \underline{c : c''}; \textbf{pat } \underline{p : \tau}; \textbf{rate } \underline{r} \triangleq FPars$
$\underline{s' : \underline{\underline{l' : \rho'}}; \underline{c'}; \underline{PE}; \underline{RE}} \triangleq APars$

TAFPar
$$\frac{\Gamma_\mathrm{s}, \Gamma_\mathrm{p} \vdash PE_l : \tau'_l \text{ and } \Gamma_\mathrm{s}, \Gamma_\mathrm{c}, \Gamma_\mathrm{p}, \Gamma_\mathrm{r} \vdash RE_m}{\Gamma_\mathrm{s}, \Gamma_\mathrm{c}, \Gamma_\mathrm{p}, \Gamma_\mathrm{r} \vdash APars <: FPars}$$
if 1) $|\underline{x}| = |\underline{x'}|$ for $x \in \{s, \underline{l_i}, c, p, r\}$ and 2) $s'_i \in \mathrm{dom}(\Gamma_\mathrm{s}), c'_k \in \mathrm{dom}(\Gamma_\mathrm{c})$ and
3) $\rho_{ij} = \rho'_{ij}, \Gamma_\mathrm{s}(s'_i) <: \{l'_{ij} \mapsto \rho'_{ij}\}$ and 4) $\tau'_i <: \tau''_i$ where
$\tau''_i \triangleq \tau_i\Theta$ where $\Theta \triangleq \{s_i \mapsto s'_i\}$
$\textbf{spec } \underline{s : \underline{\underline{l : \rho}}}; \textbf{comp } \underline{c : c''}; \textbf{pat } \underline{p : \tau}; \textbf{rate } \underline{r} \triangleq FPars$
$\underline{s' : \underline{\underline{l' : \rho'}}; \underline{c'}; \underline{PE}; \underline{RE}} \triangleq APars$

two conditions are similar to the TPar rule, and the third checks that actual and formal output parameters have matching length. The fourth condition requires that any of the actual output patterns which are also declared as outputs at a higher level are subtypes of the declared outputs at that higher level. The program following module invocation is evaluated in a pattern environment updated with entries for the actual output patterns. There is however one catch: pattern types may contain the names of formal species parameters, and these must be substituted for the corresponding actuals using a substitution Θ. Similar ideas apply to the set of legal parent compartments.