A High-Level Language for Rule-Based Modelling

Michael Pedersen, Andrew Phillips and Gordon Plotkin

8th July 2013

Abstract

Rule-based languages such as $\kappa$ excel in their support for handling the combinatorial complexities prevalent in many biological systems, including signalling pathways. But $\kappa$ provides little structure for organising rules, and large models can therefore be hard to read and maintain. This paper introduces a high-level, modular extension of $\kappa$ called LBS-$\kappa$. We demonstrate the constructs of the language through examples and three case studies: a MAPK cascade, a chemotaxis switch ring, and an insulin signalling pathway. We then provide a formal definition of LBS-$\kappa$ through an abstract syntax and a translation to plain $\kappa$. The translation is implemented in a compiler tool which is available as a web application. We finally demonstrate how to increase the expressivity of LBS-$\kappa$ through embedded scripts in a general-purpose programming language, a technique which we view as generally applicable to other domain specific languages.

1 Introduction

Mathematical modelling plays a key role in systems biology, facilitating the generation of new knowledge through the cycle of model analysis, experimental testing of hypotheses, and model refinement. As our biological knowledge-base increases through improvements in experimental techniques, the models under study also increase in size and complexity. Models represented by traditional mathematical formalisms such as ODEs become hard to develop and maintain, since they typically scale exponentially with the number of molecular species. This has prompted the development of a range of new modelling formalisms inspired by computer science, allowing models to be written in a structured fashion and generally supporting multiple types of simulation and analysis based on a single model. These formalisms include process calculi such as the stochastic $\pi$-calculus [22, 25], the continuous $\pi$-calculus [14], Beta binders [12, 26], BlenX [9], PEPA [2] and BioPEPA [4]; and rule-based languages such as BIOCHAM [3], $\kappa$ [6, 7], BioNetGen [11] and Stochastic Multilevel Multiset Rewriting [17];

Some of the above formalisms, in particular the process calculi, have built-in support for modularity, allowing large systems to be described in terms of their components. Others formalisms, however, lack modular features, complicating the development and maintenance of large models. Higher-level modular extensions have thus been developed on top of these formalisms. The Language for Biochemical Systems (LBS) is one such extension, defined on top of standard biochemical reactions. But LBS is more than an extension: it is a general framework which is parameterised on key structures, allowing for instantiation to other languages.

In this paper we show how LBS can be instantiated to yield a modular extension of the rule-based language $\kappa$. Rules in $\kappa$ describe the transformation of complexes at the level of agent (protein) binding sites, effectively handling the combinatorial explosion in the number of species, which is a key problem in the modelling of signalling pathways in particular. The instantiation of LBS to $\kappa$, called LBS-$\kappa$, can be adapted with minor modifications to the closely related BioNetGen rule-based language. The contribution of the instantiation is two-fold, providing a language for writing modular rule-based models and also demonstrating the generality of LBS as a framework. As a third contribution, a tool for LBS-$\kappa$ has been implemented and is available as a web application directly through a browser [20].

LBS is built on a foundation of modularity inherited from an underlying Calculus of Chemical Systems [24]. In addition it provides features such as compartments, complex agent expressions, combinatorial combinators, and parameterised modules with a notion of agent subtyping. We introduce these features informally through examples in Section 2. In Section 3 we demonstrate, through case studies of a MAPK cascade, a chemotactic switch ring, and an insulin signalling pathway, how the LBS-$\kappa$ features give rise to improved readability, maintainability and reusability. We formally define the LBS-$\kappa$ language in Section 4. Since LBS is a general framework, we need only define the necessary instantiation to $\kappa$, and refer to the previously published general semantics of LBS [21]. In Section 5 we introduce the LBS-Kappa web application, and we demonstrate how the expressive power of LBS-Kappa can be increased by introducing embedded F# scripts. We discuss related work in Section 6 with a particular
focus on two different approaches to high-level language development: one via domain-specific languages such as LBS-κ, and the other via embedded languages such as the recent PySB [15] which is based on the general-purpose language Python.

2 Language Overview

2.1 Agents, Rules, Composition and Compartments

An agent expression describes one or more species. For example, RAF{\(x \sim p\)} is an atomic agent with name Raf that is phosphorylated on site \(x\). More generally, post-translational modification is represented by an internal state name following a tilde symbol; we write \(p\) for phosphorylated and \(u\) for unphosphorylated. Agents can be composed into complexes using the agent composition operator, \(-\); for example Raf{\(x \sim p!1\)} - MEK{\(S222 \sim u!1\)} describes a complex of two agents, Raf and MEK. The binding is represented by the label \(1\) following an exclamation mark mark on the respective sites. The label can be any integer, but must distinguish a particular binding from any other bindings that may be present in a rule. A rule describes the transformation of agents at the level of binding sites. Take for example the following rule which expresses the binding of two proteins, Raf and MEK, on their respective sites \(x\) and \(S222\).

\[
\text{Raf}\{x \sim p\} + \text{MEK}\{S222 \sim u\} \rightarrow \text{Raf}\{(x\sim p!1)\}-\text{MEK}\{S222 \sim u!1\}
\]

The left hand side (LHS) of the rule thus specifies that for the rule to apply, Raf must be phosphorylated on \(x\) and MEK must be unphosphorylated on \(S222\). The respective internal states are the same on the LHS and RHS and are hence preserved by the rule. The absence of any binding labels on the LHS states that the agents cannot be bound on their respective sites before applying the rule. But the rule imposes no further conditions; in particular, the rule can be applied in situations where RAF or MEK are bound to other proteins on other sites, regardless of the internal state of any other sites. In this way a rule generally represents many concrete reactions, thus effectively helping alleviate the problem of combinatorial explosion.

We have adopted a syntax for rules which differs slightly from both Kappa and BioNetGen in order to accommodate the higher-level language constructs of LBS-κ. In particular, sites are specified within curly brackets, {}, rather than within parentheses. We explicitly represent complexes using the dash symbol, \(-\), and we represent distinct agents within a rule by separation with the \(+\) symbol. This syntax follows the BioNetGen convention, whereas κ syntax instead has a flat structure of comma-separated agents. Note that rules can generally be equipped with rates, which are written in LBS-κ in brackets following arrows as in e.g. \(A \rightarrow \{0.1\}\) B. We omit rates from the examples for the sake of clarity.

While agents often model proteins they can also represent other objects, for example chemical compounds such as nucleic acids. The following is a high-level example of MEK gene expression in which mRNA is transcribed from a gene and the mRNA polymerase in the first rule, and the mRNA is translated into MEK in the second rule.

\[
\begin{align*}
gene + \text{rnap} & \rightarrow gene + \text{rnap} + \text{mRNA} \\
\text{mRNA} & \rightarrow \text{MEK}
\end{align*}
\]

Observe how the two rules are composed using the parallel composition operator, \(|\), which is the glue used more generally for joining models together. In this example the rules correspond to standard, concrete reactions, given that none of the participating reactant agents have any sites.

The location operator, [ ], allows models to be composed within a tree-structured hierarchy of static compartments, and furthermore allows for cross-compartment transport rules. The following example shows how the above binding example and gene expression example can be composed within a compartment structure. Gene expression now takes place within the nucleus compartment, and we have added an additional rule for transporting mRNA out of the nucleus into the top level cell compartment before translation. Note that multiple occurrence of the same compartment name refer to the same compartment.
2.2 Complex Agent Expressions

The rules in the previous section are relatively short and easy to read. However, larger rules containing many agents can be hard to read, as demonstrated by the following two rules adapted from a model in [7]. The first rule uses the wild card binding symbol _, meaning that the site must be bound to some unspecified agent.

agent  

\begin{align*}
\text{c1} &= \text{EGFR} \{\text{CR}!, Y1148^p!1\} - \text{Shc} \{\text{PTB}!1, Y318^u\}; \\
\text{c1} &\rightarrow \text{c1} < \text{Shc} \{Y318^p\} > \\
\text{agent  c2} &= \text{EGFR} \{Y1148^p!1\} - \text{Shc} \{\text{PTB}!1, Y318^p!2\} - \text{Grb2} \{\text{SH2}!2, \text{SH3}!3\} - \text{SoS} \{a!3, b\} + \text{Ras} \{\text{S1S2}^\text{gdp}\}; \\
\text{c2} &\rightarrow \text{c2} < \text{SoS} \{b!4\} > - \text{Ras} \{\text{S1S2}^\text{gdp}\}
\end{align*}

The first rule expresses phosphorylation of Shc, and the second rule expresses binding of Ras to a large complex. We have used an explicit notation for complexes to simplify reading as compared to the original Kappa rules.

In spite of this, for the second rule in particular it still remains difficult at a first glance to determine what actions the rule expresses. Doing so requires a detailed comparison of the LHS to the RHS. LBS-\kappa addresses this through a richer notion of agent expressions by allowing complexes to be abbreviated and subsequently updated. The above rules can then be re-written as follows.

agent  

\begin{align*}
\text{c1} &= \text{EGFR} \{\text{CR}!, Y1148^p!1\} - \text{Shc} \{\text{PTB}!1, Y318^u\}; \\
\text{c1} &\rightarrow \text{c1} < \text{Shc} \{Y318^p\} > \\
\text{agent  c2} &= \text{EGFR} \{Y1148^p!1\} - \text{Shc} \{\text{PTB}!1, Y318^p!2\} - \text{Grb2} \{\text{SH2}!2, \text{SH3}!3\} - \text{SoS} \{a!3, b\} + \text{Ras} \{\text{S1S2}^\text{gdp}\}; \\
\text{c2} &\rightarrow \text{c2} < \text{SoS} \{b!4\} > - \text{Ras} \{\text{S1S2}^\text{gdp}\}
\end{align*}

In both cases LHS components are abbreviated and assigned to an identifier, c1 and c2, respectively. The abbreviated complex is then referred to and updated on the RHS of the rules using the update operator, < >. The update for the first rule states that the RHS is exactly the same as c1, except that species Shc is phosphorylated on site Y318. The update for the second rule states that the RHS is a complex obtained from c2 by adding a link from SoS to Ras. As a further abbreviation, agent expressions can be defined inline using the as keywords, as follows:

agent  

\begin{align*}
\text{EGFR} \{\text{CR}!, Y1148^p!1\} - \text{Shc} \{\text{PTB}!1, Y318^u\} &\quad \text{as} \quad \text{c1} \rightarrow \text{c1} < \text{Shc} \{Y318^p\}; \\
\text{EGFR} \{Y1148^p!1\} - \text{Shc} \{\text{PTB}!1, Y318^p!2\} - \text{Grb2} \{\text{SH2}!2, \text{SH3}!3\} &\quad \text{as} \quad \text{c2} \rightarrow \text{c2} < \text{SoS} \{b!4\} > - \text{Ras} \{\text{S1S2}^\text{gdp}\}
\end{align*}

All agents must be defined using the new operator before first use, although we have omitted doing so in the preceding examples. The shc agent is for example defined as follows:

agent  shc = \text{new } \{\text{PTB}!\{u\p\}, \text{Y318}!\{u\p\}\};

The new operator specifies the sites of the new agent, together with the internal state values each site can take. Internal state values default to u and p, so could have been omitted in the above example. As the name suggests, the new operator does indeed generate a new species that is universally unique within the model, assigned to the identifier Shc. If the above definition were repeated elsewhere in the model, the resulting agents would be semantically distinct. This can be used to e.g. create agents which are local to a module.

On a technical note, the above new agent definition is in fact an abbreviation for the following, where the agent name is repeated after the new operator:

agent  shc = \text{new } \text{Shc} \{\text{PTB}!\{u\p\}, \text{Y318}!\{u\p\}\};

The first occurrence of Shc is formally an agent identifier which is scoped within programs, whereas the second occurrence of Shc is an agent name which scoped locally to agent expressions and is used when updating the state of sites. This means that an update to the sites of shc could be written as shc < shc \{Y318^p\} > using the update notation from above. In fact, shc \{Y318^p\} is an abbreviation for exactly this. To further illustrate this point, one could in principle use a distinct identifier on the left as follows:

agent  c = shc \{\text{PTB}!\{u\p\}, \text{Y318}!\{u\p\}\};

and then write e.g. c < shc \{Y318^p\} >; but in practice this is not idiomatic.

Sometimes we need to be able to identify one of several agents within a complex that all have the same name. Agent aliases can be created in conjunction with the new operator for this purpose. Consider the following example of a rule from the chemotactic switch ring model case study involving a homo-multimer containing three agents with the same name P; internal states here take values 0 and 1.
2.3 Non-determinism

Another feature of agent expressions is non-determinism, which provides a means of grouping agents which are functionally similar through the or operator. The following example, adapted from [6], shows how the binding between a family of ERK and MEK proteins can be expressed using this mechanism:

```
agent M = MEK1 or MEK2;
agent E = ERK1 or ERK2;
M<MEK{D}> + E<ERK{CD}> -> M<MEK{D!1}>-E<ERK{CD!1}>
```

The line states that MEK can either be MEK1 or MEK2, and the second line is analogous for ERK. The third line expresses binding between these families of proteins variants; it expands to four concrete rules, one for each combination of proteins within the two families. For this example to work, the agents MEK1 and MEK2 must initially be defined to allow updates on the same site D on a common agent name MEK, and similarly for ERK1 and ERK2. This is conceptually the opposite of aliasing, and is achieved by passing the common name to the new operator as follows:

```
agent MEK1 = new MEK{D, S218, S222};
agent MEK2 = new MEK{D, S222, S226};
agent ERK1 = new ERK{CD, T202, Y204};
agent ERK2 = new ERK{CD, T185, Y187};
```

The same effect could alternatively be achieved through explicit renaming, which in turn can be expressed as an abbreviation for a module definition and invocation as shown in [21].

2.4 Parameterised Modules

A key aspect of high-level languages is their support for modularity. Modules in LBS-κ can be parameterised on agents and their sites, compartments and rates. The following is an example of a module encapsulating phosphorylation at the level of binding, modification and unbinding.

```
module phosphorylate( agent k: {m}, s: {n} ) {
 k{m} + s{n~u} -> k{m!1}-s{n~u!1} |
 k{m!1}-s{n~u!1} -> k{m!1}-s{n~p!1} |
 k{m!1}-s{n!1} -> k{m} + s{n} }
}
agent r = new Raf{x, y};
agent m = new MEK{t, S218, S222};
phosphorylate( r: Raf{x}, m: MEK{S222} )
```

The formal agent parameters k and s refer to the kinase and substrate, respectively, and include corresponding formal site parameters m and n. A module invocation is shown in line 9, generating the rules for phosphorylating the substrate MEK by the kinase Raf. Observe how sites are specified along with these actual agent parameters: in the case of MEK, there are multiple choices for sites and one must therefore be chosen. This gives rise to a notion of subtyping similar to record subtyping [23]: any actual agent parameter with at least as many sites as the corresponding formal agent parameter can be passed to a module, with the additional condition that types for the corresponding actual and formal sites match.

The above example instantiated a module with atomic agent expressions. More generally, complex agent expressions are given as module parameters. The notation is similar, but with the addition that a correspondence between atomic agents within corresponding formal and actual complex parameters must be given. On a technical note, and to illustrate this notion of complex agent parameters, the above example is in fact an abbreviation for the following:

```
agent P1, P2, P3 = new P{f: {0, 1}, x: {0, 1}, y: {0, 1}, s};
agent c000 = P1{f=0, y=1}-P2{x=1, f=0, y=2, s}-P3{x=2, f=0};
c000 <-> c000<P2{f~1}>
```

Aliases P1, P2 and P3 are created to allow the rule to update only a selected agent, namely the second one, within the complex, but all three aliases refer to the same unique underlying agent. This example also shows the use of reversible rules using the bidirectional arrow, \(\leftrightarrow\).

```plaintext
module flips (agent c000:P1{f:(0 1)}-P2{f:(0 1)}-P3{f:(0 1)}; rate k) {
    module flip(agent cxxx:P{f:(0 1)}; rate r1; rate r2) {
        cxxx<P{f'0}> ↔(r1+k){r2} cxxx<P{f'1}>
    }
    flip (c000: P2{f}, 1, 200) |
    agent c100 = c000<P1{f'1}>>;
    flip (c100: P2{f}, 1, 2) |
    agent c001 = c000<P3{f'1}>>;
    flip (c001: P2{f}, 1, 2) |
    agent c101 = c000<P1{f'1}>>P3{f'1}>>;
    flip (c101: P2{f}, 100, 2) |
}
agent P1, P2, P3 = new P{f:0 1, x:0 1, y:0 1, s};
agent c000 = P1{f'0,y'1}P2{x',0,y'2,s}P3{x'2,f'0};
agent c000b = c000<P2{s'_1}>;
flips (c000: P1{f}P2{f}P3{f}, 1) |
flips (c000b: P1{f}P2{f}P3{f}, 10) |
```

The formal parameter `agent k : k { m }` states that an agent expression is expected with at least one atomic agent containing at least one site. Given the corresponding actual parameter `Raf : Raf { x }`, the identifier `k` is bound to the same expression as `Raf` is bound to at module invocation time, which happens to be an atomic agent expression; the agent name `k` is bound to the agent name `Raf` and the site name `m` is bound to the site name `x`.

One could alternatively instantiate the module with a complex kinase as follows:

```plaintext
agent c = Ras{m!1} - Raf{y!1}
phosphorylate( c: Raf{x}, m: MEK{S222} )
```

Since the module is only concerned with a single agent within possibly larger complex, only a single agent name is listed to the right of the `:` symbol in the first actual parameter. Hence this gives rise to a notion of subtyping at the level of complexes, similar to the notion of subtyping at the level of sites.

3 Case Studies

3.1 The Chemotactic Switch Ring

Chemotaxis is the process whereby bacteria move in their environment, generally towards nutrients or away from toxins. Some bacteria, including *Escherichia coli*, achieve such movement through the mechanical rotation of flagellum attached to the cell surface. The direction of flagella rotation is determined by a switch consisting of a ring of 34 homomers each with two conformations, active and inactive, which collectively determine the overall switch state of the ring [1,10]. Each protomer can switch from inactive to active conformations, and the switch is catalysed by either a phosphorylated CheY protein or by neighbouring homomers being active.

A flat κ model of the chemotactic switch ring is available at www.rulebase.org (under the title of *Ising ring*) and is also reproduced in the Supplementary Material. The flat model consists of 8 reversible rules, each expressing...
conformational change of a protomer within a context of two neighbor protomers. Protomers \( \mathcal{P} \) have four sites: two for binding in the ring \((x\text{ and }y)\), one for binding CheY \((s)\), and one for representing conformational state \((f)\) where the internal state 0 represents inactive and 1 represents active.

An LBS-\(\kappa\) version of the model is shown in Listing 1, but with initial conditions and the rules for CheY binding omitted for clarity. The initial conditions for this model, which defines a ring with 34 protomer agents, are revisited in Subsection 5.2 in the context of embedded scripts. We use identifiers of the form \(cXXX\) to represent agent expressions with three protomers, with the values of the \(X\)s indicating the conformational state of each. The bulk of the model lies in the definition of a module called \(\text{ips}\), which contains four rules for conformational changes in each of the possible states of neighbor conformation. The module is parameterized on a complex with three protomers, with the middle one being subject to conformational change. The module is also parameterized on forward and reverse rate multipliers. Within the \(\text{ips}\) module an additional, nested module called \(\text{ip}\) is defined, which embodies a single-rule flip. Defining a single-rule module may not reduce the size of the model, but it does add clarity by naming a repeated pattern.

Two instantiations of the module are composed in the last two lines: in the second instantiation the middle protomer is bound to CheY, and in the first it is unbound. The binding to CheY is expressed implicitly through a wild card, which is adequate here since protomers never bind other agents on this site. Observe how the rate multipliers differ by an order of magnitude in the two cases. Observe also how the module instantiations make explicit the subtle difference between the two blocks of flip rules, which would not otherwise be immediately apparent from a flat model.

The model demonstrates the use of species update patterns and the use of modules parameterized on agents and rates. Observe that the \(\text{ips}\) module only specifies a single site for each protomer, while the agents in instantiations have multiple sites, demonstrating site subtyping. The nested \(\text{ip}\) module only specifies a single agent in its parameter, while the instantiations provide complexes but with the relevant agent of each complex specified; this demonstrates subtyping at the higher level of complexes.
3.2 A MAPK cascade

The Mitogen-Activated Protein Kinase (MAPK) cascade is a common motif in signalling pathways, and has the functional property of amplifying an upstream signal through three layers of protein modification. A simple $\kappa$ model of a MAPK cascade is given in [5] and reproduced in the supplementary material with some restructuring to highlight the underlying modularity. The first layer here has a single phosphorylation/dephosphorylation cycle, and the two subsequent layers each have two cycles involving two different sites of the same substrate protein. Each cycle is represented by six rules: three for phosphorylation, and three for dephosphorylation.

An LBS-$\kappa$ version of the model is shown in Listing 2. The model defines a cycle module in lines 1-16 which is parameterised on a kinase ($k$), a phosphatase ($p$) and a substrate ($s$); all three species parameters specify just one site, all called $m$. The module defines two sub-modules, one for binding, phosphorylation and unbinding ($\text{bpu}$) which is parameterised on the kinase and the substrate, and one for binding, dephosphorylation and unbinding ($\text{depsh}$) which is parameterised on the phosphatase and substrate. Note how in-line agent definitions are used to abbreviate the substrate complexes: the identity updates $e$ on the RHS of lines 5 and 11 result in the binding state remaining unaffected, i.e. bound in these cases. The body of the cycle module simply consists of instantiations of the two sub-modules in parallel.

Lines 18-24 define the agents used in the model. All agents have the default internal state types (i.e. $u$ and $p$) except for Ras which therefore explicitly declares the type for its site ($\text{gtp}$ and $\text{gdp}$). Lines 26-30 consist of a parallel composition of five module invocations, one for each cycle. Note the subtle distinction between the state of parameter agents and the site parameter: in e.g. line 29, the MEK parameter is phosphorylated on sites $S218$ and $S222$ and can have any internal state on site $m$, but it is the latter site $m$ which is passed to the module for binding.

This example demonstrates the use of parameterised modules with multiple instantiations and with site subtyping. The outer module effectively has six rules and five instantiations, and the inner modules each have three rules and a single instantiation. In addition to clarifying the model through structure, the model is also significantly smaller than the corresponding flat model with 30 rules. Note finally that the inner modules are very general, and could hence be defined in a standard library for wider use as in e.g. [15].

3.3 Insulin Signalling

The insulin signalling pathway responds to rises in blood glucose levels and accompanying rises in insulin levels with the net effect of inducing glucose uptake by cells. We consider a $\kappa$ model of insulin signalling by Isha Antani and Gordon Webster originally available from cellucidate.com, a commercial pre-cursor to rulebase.com. The model is reproduced in the Supplementary Material and contains approximately 120 rules, listed in no apparent order and with no clear a priori modular structure; it hence constitutes a challenging test case for LBS-$\kappa$. The full LBS-$\kappa$ version of the model is also listed in the Supplementary Material.

The LBS-$\kappa$ model exploits modularity at two levels. At the top level are modules with no parameters and only one instance each. These are used purely for structure, allowing the main body of the model to be specified as a parallel composition of module instantiations, each representing a functionally self-contained component:

\[
\text{receptorActivation} | \quad \text{pipAktSignalling} | \quad \text{gluconeogenesis} | \\
\text{glycogenSynthesis} | \quad \text{cellGrowth} | \quad \text{glucoseUptake} \\
\]

At the second level are modules with multiple instances. They are all small, containing up to three rules, many of which represent variations of the phosphorylation ($\text{bpu}$) and dephosphorylation ($\text{bdu}$) modules used in the MAPK cascade; these variations are in turn defined as instances of more general phosphorylation and dephosphorylation modules which are parameterised separately on agents for each of the three rules, allowing agent state to differ both between rules and between instances. Some modules contain just a single rule, e.g. for binding, unbinding and modification. Even though the resulting decrease in model size is inconsequential, these small modules do result in increased clarity by directly exposing the actions associated with one or more rules; they also implicitly expose rules which confer more complicated actions not embodied in standard modules. Consider the following excerpt:

\[
\text{bpu3}( \text{PK1}\{\text{PH}\}: \{\text{PH, pck}\}, \text{PKCz}: \{\text{T410, pdk1}\} ) | \\
\text{Akt}\{\text{as160}, \text{S473}\#p, \text{T308}\#p\} + \text{AS160}\{\text{gap}\#u1\} - \text{Rab10}\{\text{gap}\#u1\} \rightarrow \\
\text{Akt}\{\text{as160}\#1, \text{S473}\#p, \text{T308}\#u\} - \text{AS160}\{\text{gap}\#u1\} + \text{Rab10}\{\text{gap}\#u\} | \\
\text{pho}( \text{Akt}\{\text{as160}\#1}\#\text{AS160}\{\text{gap}\#u1}\}: \text{AS160}\{\text{gap}\} | \\
\text{unbind}( \text{Akt: as160}, \text{AS160: gap} ) )
\]

The first line immediately conveys that PDK1 phosphorylates PKCz through some standard mechanism embodied by the $\text{bpu3}$ module. The following three lines immediately convey that Akt phosphorylates and unbinds AS160...
through the pho and unbind modules (lines 5 and 6), but also that the binding rule (lines 3 and 4) is more complicated and may require further examination. Of course this approach relies on meaningful naming conventions for modules; further tool support could be helpful here, e.g. by providing a short cut for going to the definition of a module from a module instantiation point.

The LBS-κ model uses non-determinism in two cases, namely for the related GSK3a and GSK3b agents, and for the related IRRS1 and IRRS2 agents. It also uses agent expressions in several cases, e.g. for reducing the complexity of long rules as follows:

<table>
<thead>
<tr>
<th>Line</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Act{PH!1, T308} \cdot PIP{three_p!1} \rightarrow a +</td>
</tr>
<tr>
<td>2</td>
<td>PDK1{PH!2, akt} \cdot PIP{three_p!2} \rightarrow b →</td>
</tr>
<tr>
<td>3</td>
<td>a &lt;Act{T308!4}&gt; b &lt;PDK1{akt!4}&gt; \rightarrow c;</td>
</tr>
<tr>
<td>4</td>
<td>c \rightarrow c&lt;Act{T308_p!e}&gt;</td>
</tr>
</tbody>
</table>

This last rule exploits the identity update feature also used by the MAPK cascade: Act\{T308 \_p! e\} expresses that the site T308 should be phosphorylated, but that the binding state should remain unchanged, in this case bound to PDK1.

4 The Syntax and Semantics of LBS-κ

We now turn to the formal definition of LBS-κ. The LBS framework is parameterised on two structures. The first is a syntactic parameter which specifies a syntax for agent site expressions. The second is a semantic parameter which specifies the target semantical objects and relevant operations on these. In the case of LBS-κ, the target semantical objects are flat κ programs. The semantic parameter generally depends on the syntactic one: only agents which have a notion of binding in their site expressions can be translated to κ.

The LBS framework [19, 21] provides a general syntax, for example that of module definitions and invocations, that is independent of any particular choice of agent site expressions. For the sake of completeness, we reproduce selected parts of this general syntax in Subsection 4.2, where we also define the syntax of agent site expressions which is specific to the LBS-κ instantiation.

The LBS framework also provides a general semantics, specifying how language constructs such as module definitions and invocations are translated independently of any particular target semantics such as κ. We do not reproduce the full general semantics here, but refer instead to [19, 21]. We do however outline the general semantics framework in Subsection 4.3 in order to define the structure of the LBS semantic parameter. We then proceed to define the κ semantic parameter in Subsection 4.4, thereby completing the formal definition of the instantiation of LBS to LBS-κ.

4.1 Notation

We let \( \mathbb{R} \) denote the set of real numbers ranged over by \( r \) and we let \( \mathbb{N} \) denote the set of natural numbers ranged over by \( n \). We write \( x \) for lists, \( x_i \) for the \( i \)th element (starting from 1) of a list, \( |x| \) for the length of a list and \( e \) for the empty list. The concatenation of lists \( x \) and \( y \) is written \( xy \), and the prefix of an element \( a \) to a list \( x \) is written \( ax \).

Given a set \( X \) we write \( X^* \) for the Kleene closure of \( X \). We write \( \{x_i\}_{i \in I} \) for a finite indexed set and omit \( I \) and/or \( i \) and write \( \{x_i\}, \{x_i\} \) or \( \{x\} \) when they are understood from the context. Partial finite functions \( f : X \rightarrow Y \) are denoted by finite indexed sets of pairs \( \{x_i \rightarrow y_i\} \) where \( f(x_i) = y_i \). The domain of definition and image of a function \( f \) are denoted by \( dom(f) \) and \( im(f) \), respectively. 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; the sets are then understood to form the domain and image of \( f \).

We write \( e \triangleq e' \) for definitions where the expression \( e \) equals the expression \( e' \) if \( e' \) is defined, and where \( e \) is undefined otherwise. When a notion of well-typedness applies to \( e' \), we furthermore write \( e \triangleq_{i} e' \) for definitions where \( e \) equals \( e' \) if \( e' \) is defined and well-typed, and where \( e \) is undefined otherwise.

For easy reference some of the key symbols appearing in the syntax and semantics definitions are outlined in Table 1.

4.2 The Abstract Syntax of LBS-κ

The abstract syntax of LBS-κ forms the foundation for its subsequent semantics. A formal definition of the concrete syntax of LBS-κ and its mapping into the abstract syntax are omitted. Both can be deduced without surprises
Table 1: Some of the symbols used in LBS-$\kappa$ syntax and semantics.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id$_c$</td>
<td>Compartment identifier</td>
</tr>
<tr>
<td>e$_a$</td>
<td>Agent expression</td>
</tr>
<tr>
<td>v$_s$</td>
<td>Agent value</td>
</tr>
<tr>
<td>id$_a$</td>
<td>Agent identifier</td>
</tr>
<tr>
<td>n$_a$</td>
<td>Agent name</td>
</tr>
<tr>
<td>v$_s$</td>
<td>Site name</td>
</tr>
<tr>
<td>$\xi$</td>
<td>Agent annotation</td>
</tr>
<tr>
<td>$R$</td>
<td>Normal form rule</td>
</tr>
<tr>
<td>e$_{na}$</td>
<td>Normal form agent value</td>
</tr>
</tbody>
</table>

Table 2: The abstract syntax for general LBS agent expressions, parameterised on site types $\rho$ and site expressions $e_s$.

$$
e_a ::= 
\begin{align*}
& \text{NEW AGENT} \\
& \text{NEW AGENT} \\
& \text{LOCATION} \\
& \text{COMPOSITION} \\
& \text{UPDATE} \\
& \text{CHOICE} \\
& \text{ANNOTATION} \\
& \text{IDENTIFIER} \\
\end{align*}
$$

\begin{align*}
\xi &::= id_a[n_a,n_s] \\
\sigma &::= \{n_s \mapsto \rho\} \\
\alpha &::= \{n_s \mapsto e_s\}
\end{align*}

from the abstract syntax and the examples in the previous section.

### 4.2.1 Agent Expressions

The abstract syntax for agent expressions is given in Table 2, where $n_a$ ranges over a finite set of agent names, $n_s$ ranges over a finite set of site names, $id_a$ ranges over a finite set of agent identifiers and $id_c$ ranges over a finite set of compartment identifiers\(^1\). Agent names identify atomic agents independently of any sites, while agent identifiers refer to possibly complex expressions including both the names and site states of atomic agents in the complex. Table 2 specifically omits the definition of site types $\rho$ and site expressions $e_s$ since these are parameters of the syntax. They are defined separately in the following. Note that several abbreviations can be defined in terms of the basic abstract syntax in Table 2; some of these abbreviations are mentioned below.

New agents are created by specifying a name and a type consisting of a partial finite function from site names to site types; as we have seen in the examples, the agent name can be left out as an abbreviation in definitions in which case the identifier, to which the new agent is assigned, is assumed as the name. Location allows for agents to span compartments within rules. For this reason updates generally include compartments in addition to an agent name and a site update expression; updates on atomic agents may however omit the agent name as an abbreviation. Annotations are used when agents are passed as actual parameters in module instantiations: they specify how located atomic agents and sites should be mapped to the located atomic agents and sites in the corresponding formal agent parameter which also includes an annotation. An informal explanation of this mapping, but without location, was given in Subsection 2.4. The composition, choice, and identifier expressions are straightforward.

---

1 Note that in [19, 21] we used $n_s$ for species/agent names and $n_m$ for (modification) site names; we have adopted a different notation here to better reflect the usual $\kappa$ terminology.
Let $\mathcal{V}$ be a given finite set of internal state values. Binding site types and binding expressions specific to LBS-$\kappa$ are then defined by the abstract syntax in Table 3, where $V \subset \mathcal{V}$ ranges over sets of internal state values, $v \in \mathcal{V}$ ranges over individual internal state values and $b \in \{0, 1\}^*$ is used to create namespaces for confining link labels within modules.

A site type simply consists of a set of allowed internal state values. A site expression is a pair consisting of an internal state $i$ and a link $l$. An internal state can be a value $v$, such as “phosphorylated” or “unphosphorylated”; it can be a wild card indicating “any” value; or it can be the identity $\varepsilon$ for use in updates when the internal state should remain unaffected. A link can be one of two kinds of wild cards, with the more permissive being “either free or bound” and the more restricted being “bound to something”; a link can also be free, i.e. unbound; it can be bound by some specific label $n \in \mathbb{N}$ which includes a namespace $b$; or it can be the identity $\varepsilon$ for use in updates when the link state should remain unaffected. The internal state may be omitted from a site expression as an abbreviation, in which case the wild card, $?$, is assumed, and the link may be omitted as an abbreviation in which case the free link, $\circ$, is assumed. If both are omitted, the pair $(?, \circ)$ is assumed.

### 4.2.2 Programs

The abstract syntax of programs is given in Table 4, where $n \in \mathbb{N}$, $id_m$ ranges over the set of module identifiers, $id_i$ ranges over the set of rate identifiers and $id_c$ again ranges over the set of compartment identifiers. Definitions,
Table 5: The abstract syntax for definitions.

\[
D ::= \\
| \text{id}_a = e_a \\
| \text{id}_c = \text{new comp} \\
| \text{id}_c = \text{new comp inside } \text{id}_c \\
| \text{id}_r = e_r \\
| \text{id}_m(\text{id}_c; \text{id}_a; \xi; \text{id}_l) = P
\]

<table>
<thead>
<tr>
<th>DEFINITION</th>
<th>AGENT</th>
<th>NEW TOP-LEVEL COMPARTMENT</th>
<th>NEW NESTED COMPARTMENT</th>
<th>RATE</th>
<th>MODULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>id\ a = e\ a</td>
<td>Agent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id\ c = new\ comp</td>
<td>New\ top-level\ compartment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id\ c = new\ comp\ inside\ id\ c</td>
<td>New\ nested\ compartment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id\ r = e\ r</td>
<td>Rate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id\ m(\text{id}_c; \text{id}_a; \xi; \text{id}_l) = P</td>
<td>Module</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ranged over by \(D\), are defined below. The grammar closely matches the syntax used in the examples and should be self-explanatory, perhaps with the exception of initial populations and nil programs: the former are specified using the \textit{init} keyword and can occur anywhere in a program, and the latter acts as the identity element under parallel composition: this can be useful for advanced constructions.

There are some evident abbreviations. For example, a reversible rule corresponds to the parallel composition of the rules for each direction, and the definition of an agent expression within a rule using the \textit{as} keyword can be expressed using a standard agent definition prior to the rule as detailed in [21].

4.2.3 Definitions

The abstract syntax for definitions is shown in Table 5. Note that compartments, like agents, are defined using the \textit{new} keyword, and unless a compartment is used at the top level (the first case), a parent must be specified (the second case). Formal agent parameters in module definitions have annotations \(\xi\) as defined in the abstract syntax for agent expressions. Together with the corresponding annotation of actual agent parameters which are included in the abstract syntax for agent expressions, this is sufficient to construct a that allows use of the agents inside the module body after invocation.

4.3 The General Semantics Framework

The general semantics framework of LBS is independent of any particular target semantics such as \(\kappa\). The framework hence defines, in general terms, the meaning of e.g. agent updates, location, module definitions and module invocations. As a result, the general semantics need not be redefined for each concrete semantics under study: all that is needed is the definition of the functions on which the general semantics is parameterised. We hence refer to [21] for the definition of the general semantics, which also includes instantiations to concrete semantics for ordinary differential equations, continuous time Markov chains and Petri nets; in these cases agents in the abstract syntax have no binding structure.

The general semantics evaluates individual rules to normal form rules \(R\) as defined by the following grammar:

\[
R ::= \frac{v_{na} \Rightarrow v'_{na}}{}
\]

\[
v_{na} ::= \frac{n_c[n_a, \alpha_\sigma]}{}
\]

\[
\alpha_\sigma ::= \{n_a \mapsto (\rho, e_s)\}
\]

A ground normal form rule is like a rule but with rate expressions evaluated to rate constants and agent expressions evaluated to normal form agent values, \(v_{na}\); normal form agent values are lists of triples with a compartment list for identifying location, an agent name, and a typed site assignment \(\alpha_\sigma\) which to each site assigns a type \(\rho\) and a site expression \(e_s\). Note that the site type and expressions constitute the parameters of the general syntax. Note also that the general semantics includes an additional ground normal form of rules and agents, but that distinction is not needed in the LBS-\(\kappa\) context where site expressions have no variables.

In order to produce normal form rules from a model the general semantics is parameterised on the following relation and functions pertaining to site types \(\rho\) and site expressions \(e_s\):

- A \textit{typing relation} of the form \(e_s : \rho\) giving types to site expressions. This is used for determining well-typedness of agent expressions.
• A default expression function of the form \( \text{default}(\rho) = e_s \) giving default expressions to site types. This is used to assign site expressions to unassigned sites in agent expressions.

• An update function of the form \( e_s(e'_s) = e''_s \) for updating one site expression with another. This is used in the semantics of agent update expressions.

• A seal function of the form \( \text{seal}(e_s, b) = e'_s \) for confining names in site expressions to a namespace given by a binary string \( b \in \{0,1\}^* \). This is used to avoid capture of free names (link labels in the \( \kappa \) context) when agent expressions are passed as module parameters.

In order to produce a concrete semantical object such as a flat \( \kappa \) model from an LBS-\( \kappa \) model, the general semantics framework is further parameterised, on a concrete semantics structure \( (S, |S|, 0_S, R_S, I_S) \) 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 \( 0_S \in S \).

• A partial rule assignment function of the form \( R_S(R, b) = O \) assigning a semantical object to a given normal form rule \( R \) named \( b \).

• A partial initial condition assignment function of the form \( I_S(v_{na}, n) = O \) assigning a semantical object to an initial population \( n \) of a normal form agent value \( v_{na} \).

### 4.4 The Concrete Kappa Semantics

We define the relations and functions pertaining to site types and expressions in the \( \kappa \) instantiation as follows, where \( IV(e_s) \) denotes the (singleton or empty) set of internal state values in the site expression \( e_s \):

- \( e_s : \text{binding}(V) \) for all \( e_s \) and \( V \) with \( IV(e_s) \subseteq V \)
  - \( \text{default}(\text{binding}(V)) \overset{\Delta}{=} (\varepsilon, \varepsilon) \)
  - \( (i,l)(i',l') \) \overset{\Delta}{=} (i(i'), l(l')) \) where
    - \( i(i') \overset{\Delta}{=} \begin{cases} i & \text{if } i' = \varepsilon \\ i' & \text{otherwise} \end{cases} \)
    - \( l(l') \overset{\Delta}{=} \begin{cases} l & \text{if } l' = \varepsilon \\ l' & \text{otherwise} \end{cases} \)
  - \( \text{seal}(i,l,b) \overset{\Delta}{=} \begin{cases} (i, (n, bb)) & \text{if } l = (n, b') \\ (i, l) & \text{otherwise} \end{cases} \)

The typing relation asserts that a binding expression can only use internal state values declared by its type. The default expression for unspecified modification site types has a wildcard internal state and link, which reflects the use of unspecified sites in \( \kappa \). 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 concatenating the given binary string to the binary string already present.

Next we define the concrete semantics structure for \( \kappa \). The abstract syntax for \( \kappa \) rules is given in Table 6 where, as before, \( n_a \) ranges over the set of agent names, \( n_s \) ranges over the set of site names, \( r \in \mathbb{R} \) and \( n \in \mathbb{N} \). Furthermore, \( v \in \mathbb{V} \cup \mathbb{N}_c \) ranges over a given set of internal state values as in LBS-\( \kappa \), but also over the set of compartment name lists; the latter is needed in order to encode LBS-\( \kappa \) compartments in \( \kappa \). The abstract syntax for \( \kappa \) rules follows that of the literature (see e.g. [8]), but is adapted notationally for our purposes.

A \( \kappa \) rule consists of a list of reactant agents and a list of product agents, and the arrow is labelled with a rate constant. A \( \kappa \) agent is similar to an LBS-\( \kappa \) agent, with the exception that site expressions are lists rather than functions and the identity \( \varepsilon \) is omitted from internal and link states.

A \( \kappa \) program \( K \) is then a pair \( (X, I) \) where \( X \) is a set \( \{x_i\} \) of \( \kappa \) rules and \( I \) is a list \( a \) of lists of complexes (which in turn are lists of agents) representing the initial conditions for simulation. We denote by \( K \) the set of all programs.

We are now in a position to define two elements of the concrete semantics structure for \( \kappa \), namely the the parallel composition and the nil object:
Table 6: The abstract syntax for $\kappa$ rules.

$$
\begin{align*}
  x & ::= a \rightarrow^r a' & \text{ $\kappa$ rule} \\
  a & ::= (n_a, g) & \text{ Agent} \\
  s & ::= (n_s, i^-, l^-) & \text{ Site} \\
  i^- & ::= v & \text{ Internal state} \\
  & | ? & \text{ Internal state value} \\
  l^- & ::= ? & \text{ Wild card} \\
  & | _ & \text{ Bound to something} \\
  & | o & \text{ Free} \\
  & | n & \text{ Link label}
\end{align*}
$$

\begin{itemize}
  \item $K_1 \mid_K K_2 \overset{\Delta}{=} K$ where
    \begin{align*}
      - X_K & \overset{\Delta}{=} X_{K_1} \cup X_{K_2} \\
      - I_K & \overset{\Delta}{=} I_{K_1} \cap I_{K_2}
    \end{align*}
  \item $0_K \overset{\Delta}{=} (\emptyset, \varepsilon)$
\end{itemize}

We define the remaining elements below.

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; initial conditions are well-typed if all agents are ground, i.e. they contain no wild cards; and finally, a $\kappa$ program is well-typed if all its rules and its initial conditions are well-typed.

The sites of an LBS-$\kappa$ agent are represented by finite functions rather than lists as in $\kappa$. The translation to $\kappa$ must therefore “linearise” these functions, for which we assume a linear ordering, $\leq$, on site names. It must also convert restricted link labels to natural-number link labels, for which we assume an injective function of the form $\text{enc}(n, b) = n'$; a definition could e.g. be based on a Gödel numbering. Given an LBS-$\kappa$ modification $\alpha = \{n_s \mapsto (\rho, (i, l))_j\}$ we then define $\text{kap}_m(\alpha)$ to be the list with the element $(n_{s j}, i_j, \text{enc}(l_j))$ at index $|S|$ where

$$
S \overset{\Delta}{=} \{n_s \in \text{dom}(\alpha) \mid n_s \leq n_{s j}\}
$$

We here assume $\text{enc}$ extended to LBS-$\kappa$ links in an evident manner. Note that site types are not needed for the $\kappa$ translation, and they are hence discarded.

The translation of an LBS-$\kappa$ agent to a $\kappa$ agent simply translates the agent’s sites and adds an additional site with an internal state representing the enclosing compartments. For the latter we assume a distinguished site name, $\text{comp}$. This ensures that agents in different compartments are distinguished. We then define a kappa translation function of the form $\text{kap}(v_{na}) = a$ for translating normal form agent values to lists of $\kappa$ agents as follows:

$$
\text{kap}\left(n_{\text{na}}[n_{a}, \alpha]\right) \overset{\Delta}{=} (n_a, \text{kap}_m(\alpha)(\text{comp}, n_{s j}, o))
$$

where $\text{kap}_m(\alpha)(\text{comp}, n_{s j}, o)$ following our notational conventions is the postfixing of the triple $(\text{comp}, n_{s j}, o)$ to the list $\text{kap}_m(\alpha)$. LBS-$\kappa$ rules are translated into $\kappa$ rules by applying the above function to each normal form agent value and flattening the lists representing reactants and products. In the following we therefore assume a given function flatten for flattening lists, and also a given function of the form $n \times a = a'$ which generates a list $a'$ with $n$ copies of the agent list $a$.

The concrete semantics for LBS-$\kappa$ in terms of $\kappa$ is then given by the tuple $(K, \mid_K, 0_K, G_K, I_K)$ where the first three elements are defined above and the last two elements are defined as follows:
Figure 1: A screenshot of the web application for LBS-κ. The left hand side provides a syntax-highlighting editor, and the right hand side shows time course simulation plots.

\[
\begin{align*}
  G_K(v_{na} \rightarrow^r v'_{na}, b) & \triangleq_t K \\
  & \text{where} \\
  & - X_K \triangleq_t \{\text{flatten}(\text{kap}(v_{na}) \rightarrow^r \text{flatten}(\text{kap}(v'_{na})))\} \\
  & - I_K \triangleq \varepsilon \\
  I_K(v_{na}, n) & \triangleq_t K \text{ where} \\
  & - X_K \triangleq \emptyset \\
  & - I_K \triangleq_t n \times \text{kap}(v_{na})
\end{align*}
\]

Note that the rule assignment function is only defined if the resulting κ rule is well-typed, and the initial conditions assignment function is only defined if the resulting agent list is well-typed. Note too that well-typedness of agent expressions with respect to link labels is only determined by the general semantics when the concrete semantics is applied to rules. A dedicated type system would be needed to determine well-typedness earlier, e.g. at species definition time.

The translation can be adapted with minor modifications to target BioNetGen. There, agents in rules are separated by connectivity as in LBS-κ, meaning that rules need not be flattened during translation.

5 Tools

The LBS-Kappa language is supported by a web application which we describe in the first subsection below. We then demonstrate in the second subsection how the expressive power of LBS-Kappa can be increased through embedded scripts.

5.1 A Web Application

The LBS compiler is implemented in the F# language, as is its instantiation to LBS-κ. The implementation uses standard parser generator libraries (Lex and Yacc), and the code generator is implemented in a functional style naturally aligned with the LBS-κ semantics. Furthermore, the compiler is translated to JavaScript using the WebSharper kit, and is embedded in a web application which is available directly through a browser [20]. The web application also provides a syntax-highlighting editor and a simulator for plain Kappa models resulting from compilation of LBS-Kappa models. This simulator is implemented in F# and translated to JavaScript using WebSharper. A screenshot of the application is shown in Figure 1.

5.2 Embedded Scripts in LBS-Kappa

LBS-Kappa is a domain specific language: it is designed to naturally and succinctly capture processes in the specific domain of biochemistry. But at times more expressive power is needed, e.g. for generating large initial conditions as in the chemotactic switch ring case study. We here propose a simple solution, embedding a general-purpose language within LBS-κ. This allows the domain specific features of LBS-κ to be used for the bulk of a model, while
seamlessly calling upon the general-purpose language when needed for more complicated or ad hoc constructions. Our general purpose language of choice is F#, although any other language is in principle possible. We refer to segments of general purpose code as scripts, enclosed by script blocks within LBS-κ; scripts evaluate to strings which themselves are LBS-κ programs.

Below is an example of how 100 variants of an agent can be created and combined into a single non-deterministic agent $s$ using a script. The resulting non-deterministic agent is subsequently used in a degradation rule outside of the script, using the standard domain specific features of LBS-κ.

```fsharp
1 script {
2   let agentNames = seq {for i in 1 .. 100 -> "S" + string(i)} in
3   let newDefsList = Seq.map (fun s -> "agent_" + s + "_new{}") agentNames in
4   let newDefsStr = String.concat "\n" newDefsList in
5   let choiceStr = "agent_" + name + "_~" + (String.concat "_or_" agentNames) in
6   newDefsStr + "\n" + choiceStr
};
7 $ ->
```

Line 2 creates a sequence of a hundred strings representing agent names, $S_1$ to $S_{100}$. Lines 3-4 create a list of new agent definition strings, one for each agent name. Line 5 combines this list into a string. Line 6 creates a definition string for a non-deterministic agent $s$. The new agent definitions and non-deterministic agent definitions are finally combined and returned in line 7. The net effect of the script is, informally, to replace the script by the string resulting from evaluating the script. Hence the rule in line 8 is evaluated in an environment where the identifier $s$ is bound to a non-deterministic species with a hundred variants, thus resulting in a hundred different degradation rules.

It is furthermore possible to define and reuse embedded scripts. This enables a second, general-purpose notion of modularity in LBS-κ through F# function definitions. The following example shows how the above F# code for generating a non-deterministic agent definition can be abstracted into an F# function called generate which is parameterised on an agent name and a number of variants; the function definition is then embedded within a scriptdef block, and subsequently invoked from within a separate script block.

```fsharp
1 scriptdef {
2   let generate name num =
3     let agentNames = seq {for i in 1 .. num -> name + string(i)} in
4     let newDefsList = Seq.map (fun s -> "agent_" + s + "_new{}") agentNames in
5     let newDefsStr = String.concat "\n" newDefsList in
6     let choiceStr = "agent_" + name + "_~" + (String.concat "_or_" agentNames) in
7     newDefsStr + "\n" + choiceStr
};
8 script {
9   generate "S" 100
};
10 $ ->
```

Technically, the contents of scriptdef blocks are collected into a script environment during compilation of an LBS-κ model; the environment is simply a string, consisting of the concatenation of any script definitions in scope at a given point during compilation. A script of the form script { $s$ ; $P$ }, where $P$ is the sequentially following LBS-κ program, is evaluated by first appending $s$ to the current script environment and then compiling the resulting F# code using the standard F# compiler (through the F# CodeDom library). The compiled code is then executed to obtain an LBS-κ program string. The string is extended to the right with a “hole”. The extended string is then parsed using the LBS-κ parser, resulting in an LBS-κ abstract syntax tree (AST) for the script. The AST for $P$, i.e. the sequential LBS-κ program, is then inserted into the hole of the first AST, and the compiler recurses on the resulting AST.

Finally, we revisit the chemotactic ring switch case study from Subsection 3.1. Recall that the case study involves an initial condition of a ring with 34 protomers, explicitly written out in the flat model in the Supplementary Material. This initial condition can be defined with an embedded script as follows:

```fsharp
1 script {
2   "init_" +
```
6 Discussion

We discuss related work in the first subsection before finally concluding.

6.1 Related Work

The languages Antimony and PySB mentioned in the introduction are closely related to LBS-κ in that they both address the problem of modularity in a rule-based setting. Being independently developed, the three languages have emerged with different sets of features and strengths. We compare each language to LBS-κ below; but common to both Antimony and PySB is that neither have language support for the LBS-κ features of agent expressions, agent subtyping in modules, compartment hierarchies, or non-determinism; further, neither has tools accessible directly through a web browser.

Antimony

Antimony is designed to be “a modular human-readable, human-writeable model definition language” [27]. A key focus is on translation to SBML, and the language therefore directly supports SBML features such as events and inhibition/activation reaction types. Antimony also supports synthetic biology features such as DNA parts. A tool for editing models and compiling them to SBML is available for Windows and Mac OS. As a consequence of its SBML focus, Antimony models express concrete reactions rather than κ-style rules.

The LBS framework, in contrast, is designed with formal foundations in mind, based on the Calculus of Chemical Systems. While formal foundations may be of little interest to the general modelling practitioner, there are some key advantages. One is that the language itself is precisely defined, so there is no ambiguity in the meaning of any of the language constructs; the intended meaning can be obtained by reference to the formal semantics. A second is that new instantiations of the framework, such as LBS-κ, can be defined with relative ease, without having to redefine the full language. And a third advantage is that key properties of an LBS instantiation can be studied independently of the full language, as was demonstrated in [18] for Petri net invariants. In the κ context, this may for example prove useful in studies of modular analysis techniques.

PySB

Both LBS-κ and Antimony can be considered domain specific languages. PySB, in contrast, can be described as an embedded language: models are written in the syntax of a general-purpose programming language, here Python, but in a style that fits the domain as closely as possible. This extends previous work on a Python-embedded language called Little b [16]. PySB provides seamless integration with tools for simulation and analysis of κ and BioNetGen models. A number of mature Python libraries for scientific computing are also readily available, which is a key strength of the embedded approach. Another strength is flexibility: since Python is a general-purpose programming language, there are no limitations on the models which can be generated.

Flexibility, however, also has a downside. Since anything is possible, bad things may also be possible. For example, a module may include a rule in which two reactant agents are bound using the binding label 1, and two additional reactant agents are parameters of the module. If these parameters are instantiated with agents bound using the same label 1, this would result in an error when the generated κ model is processed by the κ tools.

Another downside of embedded languages is their syntax, which, by definition, cannot escape the heritage of their general purpose host language. The following is an example of a PySB catalysis module adapted from the supplementary material of [15], with rule labels removed for clarity:
While the meaning of this module is fairly clear due to the use of Python operator overloading for e.g. `+`, `%` (complexes), `>>` (reactions) and `<>` ( reversible reactions), some syntactic overhead is still needed: the Rule identifier must be used, and reactions must be collected into arrays which must be explicitly manipulated throughout a model. Note also how site parameters are independent from the corresponding agent parameter, so in order to use the module it may be necessary to inspect the module body to determine which site belongs to which agent. Finally, site values are Python primitives, meaning that a single site cannot have both internal state and binding state; it may hence be necessary to artificially separate one site into two distinctly named sites. We note that although these points do apply to PySB, they are not necessarily general points against embedded languages: embeddings into other languages with different trade-offs may be possible.

6.2 Conclusion

We have introduced the LBS-κ language for writing high-level, modular rule-based models, and we have illustrated its use via a number of small examples and three larger case studies. The first two case studies are characterised by repeated structure which can be expressed naturally using LBS-κ modularity. The third, and largest, case study model of insulin signalling exhibits no obvious large scale modularity, but the model nevertheless benefits from the use of modules for common rules and triplets of rules, and from other language constructs such as agent expressions and non-determinism.

The LBS-κ language has been formally defined, and a compiler from LBS-κ models to plain κ models has been implemented and made available through a web application. We have contrasted two approaches to language-based modelling, namely through DSLs and embedded languages, and we have demonstrated one way of combining both approaches through the use of F# scripts directly within LBS-κ. We view this discussion as being of wider relevance to modelling language design, as the notion of embedded scripts is not LBS-κ specific.

As regards future work, the question of whether large multi-instance modules of the kind discussed here really exist in natural, evolved systems remains open. But in synthetic biology, where novel systems are engineered for useful purposes, abstraction through modularity is likely to become an important means for coping with complexity. Modular DSLs targeted to the synthetic biology domain may hence be of increasing interest. With respect to language design, one promising direction is to extend κ sites with a notion of colour as in coloured Petri nets [13,17], allowing more elaborate conditions on internal state and changes thereof during rule application. With respect to tooling, the current Kappa simulator could benefit from optimisation.

References


TODO

1. Currently the syntax agent s = new{m:(u p)} doesn’t parse, it must be written s = new{m:kappa(u p)}. Either change the parser or the text.