Technical Report CoSBi 09/2008

The BlenX Language: A Tutorial

Lorenzo Dematté
CoSBi and Universitá di Trento
dematte@cosbi.eu

Corrado Priami
CoSBi and Universitá di Trento
priami@cosbi.eu

Alessandro Romanel
CoSBi and Universitá di Trento
romanel@cosbi.eu

Abstract

This paper presents a new programming language, BlenX. BlenX is inspired to the process calculus Beta-binders and it is intended for modelling any system whose basic step of computation is an interaction between subcomponents. The original development was thought for biological systems. Therefore this tutorial exemplifies BlenX features on biology-related systems.

1 Introduction

In recent times a large effort has been devoted to the application of computer science formal specification approaches in the realm of biological modelling, simulation and analysis. A successful strand of these activities is related to the use of process calculi: simple formalisms made up of a very limited set of operators to describe interaction-driven computations and originated from the CCS [18] and CSP [15] precursors.

Process calculi are usually based on the notion of communication described through a set of actions and reactions (complementary actions, or simply co-actions), temporally ordered. To denote such a chain of events, the action prefix operator is used, which is written as an infix dot. For instance, $a!.b?.P$ denotes a process that may offer a, then offers b, and then behaves as process P. The behaviour of the process consists of sending a signal over a channel named a ($a!$) and waiting for a reply over a channel named b ($b?$). Parallel composition (denoted by the infix operator "|", as in $P|Q$) allows the description of processes which may run independently in parallel and also synchronize on complementary actions (a send and a receive over the same channel). Communication is binary and synchronous. If we have more than one process willing to send a signal over a channel, but only one process willing to receive a signal on the same channel we will select non deterministically the pair of processes that synchronize. Since non deterministic behaviour is inherent to concurrent systems where we cannot make any assumption on the relative speed of processes, we also introduce the summation operator to specify non deterministic behaviour. The process $P + Q$ behaves either as P or as Q. The selection of an alternative discards the other forever. Note that instead parallel composition is such that the non moving process is unaffected and it is still available after the move of the other. To represent a deadlock situation, where the process is unable to perform any sort of action or co-action, the *nil* operator is used.

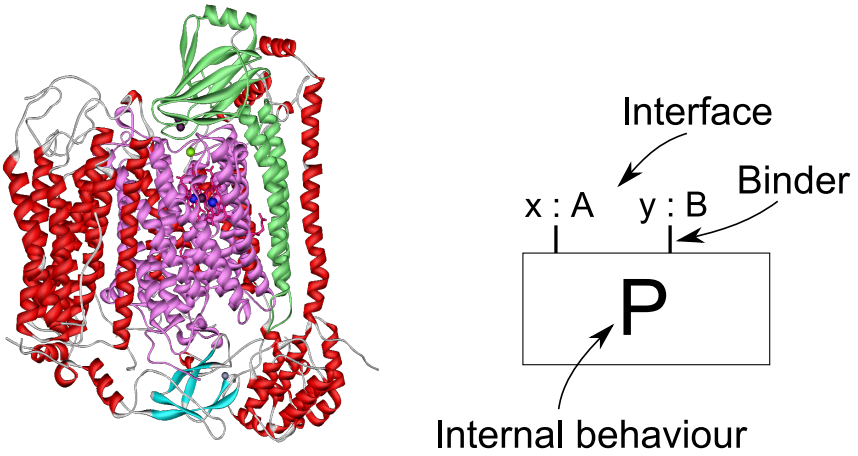
The behavior of a system is given by the ordered sequence of actions and reactions that a system can perform. Despite of its simplicity, the language contains the crucial ingredients for the description of concurrent and cooperating systems. Actions and co-actions, that are usually seen as input and output activities, can be the abstract view of any sort of complementarities. Actions could well correspond to the abstract view of requests sent by an operating system to a printer manager, or the conformational changes that take place in a receptor protein in response to its binding with the signal molecule. What is crucial to notice here is that, whichever is the level of abstraction considered, by its own nature a process algebra describes a system in terms of what its subcomponents can do rather than of what they are.

The first process calculus applied to biological problems has been the stochastic π-calculus [20] for which run-time supports that allow for the simulation of

the models have implemented [19], then followed by other calculi as BioAmbients [23], Brane Calculi [2], CCS-R [4], k-calculus [3], PEPA [14]. For a general introduction to the use of process calculi in biology see [8, 22]. The experience done with the stochastic π-calculus to model biological systems shows limitations of the classical process calculi approach for life science modelling. The main drawbacks are two:

1. the modularity (encapsulation) features needed to limit complexity and for fostering scalability and incremental model building that is implemented through the restriction or scope operator;

2. key-lock mechanism of communication. Two processes can synchronize only if the share exactly the same channel name. The biological situation is quite different. In fact two molecules interact if they have a certain degree of affinity or sensitivity which usually is different from exact complementarity of their structure.



(a) (b)

Figure 1: Boxes as abstractions of biological entities. Active sites, or *domains*, in a protein are represented as binders on the box interface.

An attempt to overcome the above limits has been done through the definition of the calculus Beta-binders [21]. The novelty of this calculus is given by the introduction of *boxes* with interfaces identified by unique identifiers that express the interaction capabilities of the processes encapsulated into the boxes.

Boxes can be interpreted as *biological entities*, i.e. components that interact in a model to accomplish some biological function: proteins, enzymes, organic or inorganic compounds as well as cells or tissues. The interaction sites on boxes are called *binders*; as for biological entities, a box has an interface (its set of binders) and an internal structure that drives its behaviour (see Fig. 1). For example, when a box is used to model a protein domain, binders can be used to represent *sensing domains* and *effecting domains*. Sensing domains are the places where the protein receives signals, effecting domains are the places that a protein uses for propagating signals, and the *internal structure* codifies for the mechanism that transforms an input signal into a protein conformational

change, which can result in the activation or deactivation of another domain. This is inspired by the available knowledge of protein structure and function (see for example [26]).

Signals are represented as messages exchanged over communication channels. Consider the pairs $x : A$ on a binder (see Fig. 1(b)): the binder name x is the name used by the internal process to perform input/output actions, while the binder identifier A expresses the interaction capabilities at x. When composing different boxes together, we use the binder identifier A to express the possible interactions between boxes; in other words, two boxes with binder identifiers A and C can interact only if A is *affine* to C.

Starting from this basic idea we moved from process calculi toward programming languages with run-time stochastic support. In this paper we present an introduction to the BlenX language and to its supporting modelling, analysis and visualization tools. The requirements we followed in the definition of BlenX and that now are the distinguishing features of BlenX are:

- dynamically varying interfaces of biological components;
- sensitivity-based interaction;
- one-to-one correspondence between biological components and boxes specified in the model;
- description of complexes an dynamic generation of complexes;
- spatial information;
- hybrid parameter specification;
- de-coupling of qualitative description from the quantities needed to drive execution;
- events;
- Markov chain generation;
- biochemical reactions generation.

The paper is organized as follows. The next section briefly recall the computational tools built around BlenX to write and edit programs, to execute or transform them, to inspect the outcome of executions. Since the BlenX language is stochastic, Section 3 recalls the basics of stochasticity we need in the following development. Section 4 introduces the primitives and programming ideas of BlenX. Section 5 reports some biological examples modelled in BlenX and simulated through the Beta Workbench.

2 The Beta Workbench

The Beta Workbench (BWB for short), is a set of tools to design, simulate and analyse models written in BlenX[1].

The core of BWB is a command-line application (core BWB) that hosts three tools: the BWB *simulator*, the BWB *CTMC generator* and the BWB *reactions*

[1] BWB is available at `http://www.cosbi.eu/Rpty_Soft_BetaWB.php`

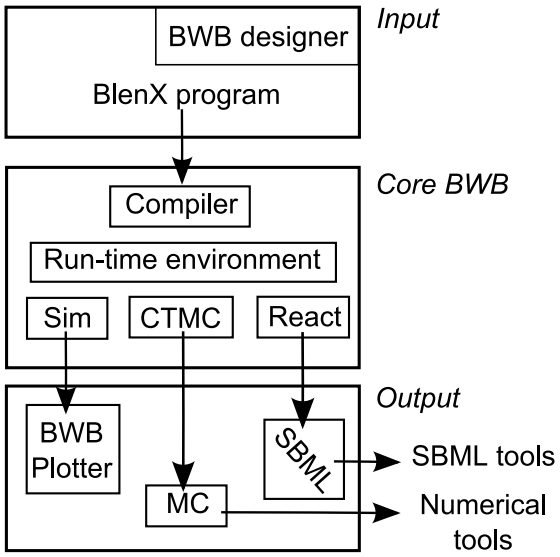


Figure 2: The logical strucure of BWB

generator. These three tools share the BlenX *compiler* and the BlenX *runtime environment*. The core BWB takes as input the text files that represent a BlenX program (see Sec. 4), passes them to the *compiler* that translates these files into a runtime representation that is then stored into the *runtime environment*. The logical arrangement of the computational blocks above is depicted in Fig. 2.

The BWB *simulator* is a *stochastic simulation engine*. The *runtime environment* provides the *stochastic simulation engine* with primitives for checking the current state of the system and for modifying it. The *stochastic simulation engine* drives the simulation handling the time evolution of the environment in a stochastic way and preserving the semantics of the language. The stochastic simulation engine implements an efficient variant of the Gillespie's algorithms described in [12, 13].

When rates are drawn from an exponential distribution (see Sec. 3) and models are finite-state, a BlenX program give rise to a continuous-time Markov process (CTMC). The BWB *CTMC generator* adds to the core blocks a set of *iterators* to exhaustively traverse the whole state space of a BlenX program. The *CTMC generator* also labels all the transitions between states with their exponential rate.

The BWB *reactions generator* identifies state changes that can be performed by *entities* and *complexes* generated by the execution of a BlenX program and produces a description of the system as a list of *species* and a list of chemical reactions in which species are involved. These lists are abstracted as a digraph in which nodes represent species and edges represent reactions (see Fig. 3). This graph can be reduced to avoid presence of reactions with infinite rate. The final result is an SBML description of the original BlenX program (Fig. 4).

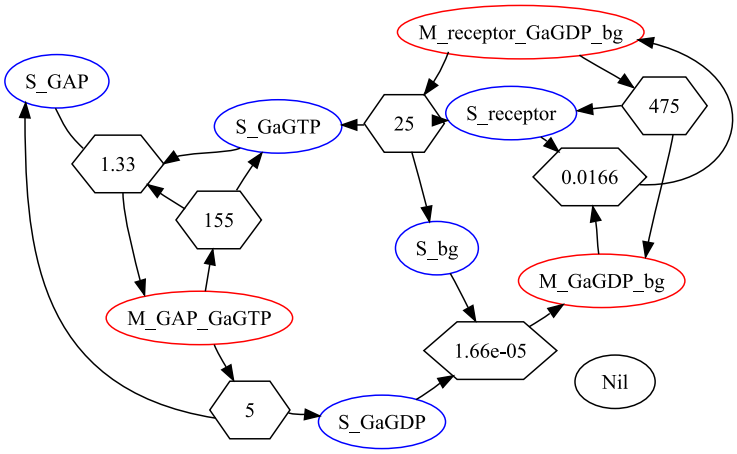


Figure 3: The graph of all the reactions generated by the BWB Reactions generator

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level1" level="1" version="1">
 <model name="SBMLmodel">
  <listOfSpecies>
   <specie name="S_GAP" compartment="compartment" initialAmount="206"/>
   <specie name="S_GaGTP" compartment="compartment" initialAmount="600"/>
   <specie name="S_GaGDP" compartment="compartment" initialAmount="600"/>
   <specie name="S_bg" compartment="compartment" initialAmount="1500"/>
   <specie name="S_receptor" compartment="compartment" initialAmount="313"/>
   <specie name="M_receptor_GaGDP_bg" compartment="compartment" initialAmount="0"/>
   <specie name="M_GaGDP_bg" compartment="compartment" initialAmount="1100"/>
   <specie name="M_GAP_GaGTP" compartment="compartment" initialAmount="0"/>
  </listOfSpecies>
  <listOfReactions>
   <reaction name="R0" reversible="false">
    <listOfReactants>
     <specieReference specie="M_GAP_GaGTP"/>
    </listOfReactants>
    <listOfProducts>
     <specieReference specie="S_GAP"/>
     <specieReference specie="S_GaGTP"/>
    </listOfProducts>
    <kineticLaw formula="M_GAP_GaGTP * c0">
     <listOfParameters>
      <parameter name="c0" value="155"/>
     </listOfParameters>
    </kineticLaw>
   </reaction>
   <reaction name="R1" reversible="false">
```

Figure 4: The SBML file generated by the BWB Reactions generator

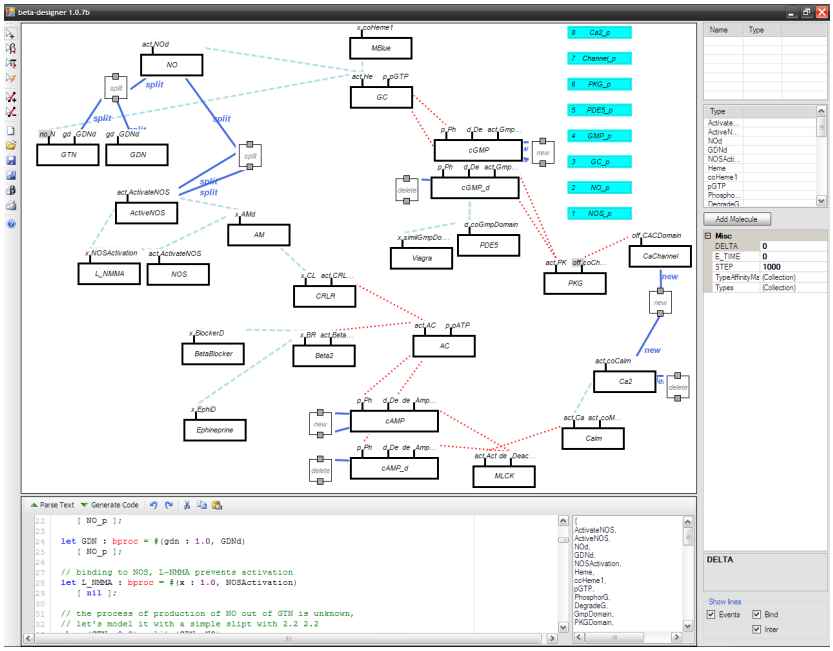


Figure 5: The model of a complex pathway in the designer.

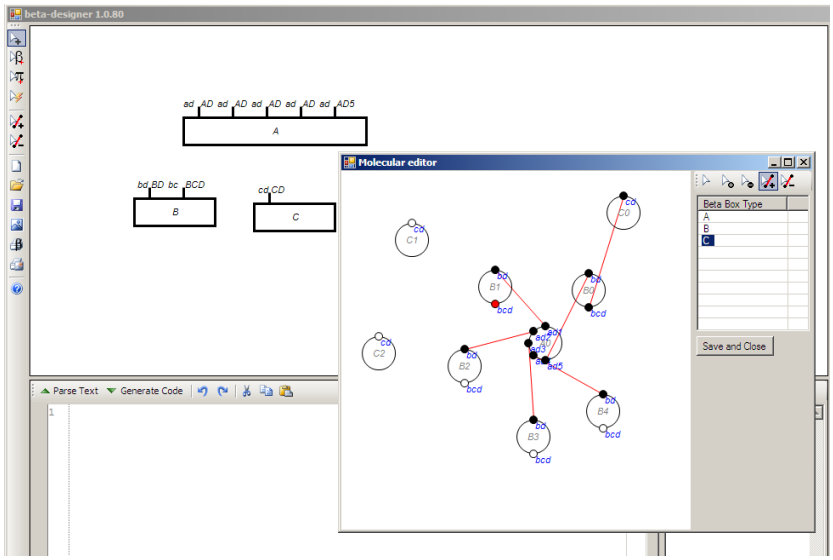


Figure 6: Definition of a complex through the Designer interface.

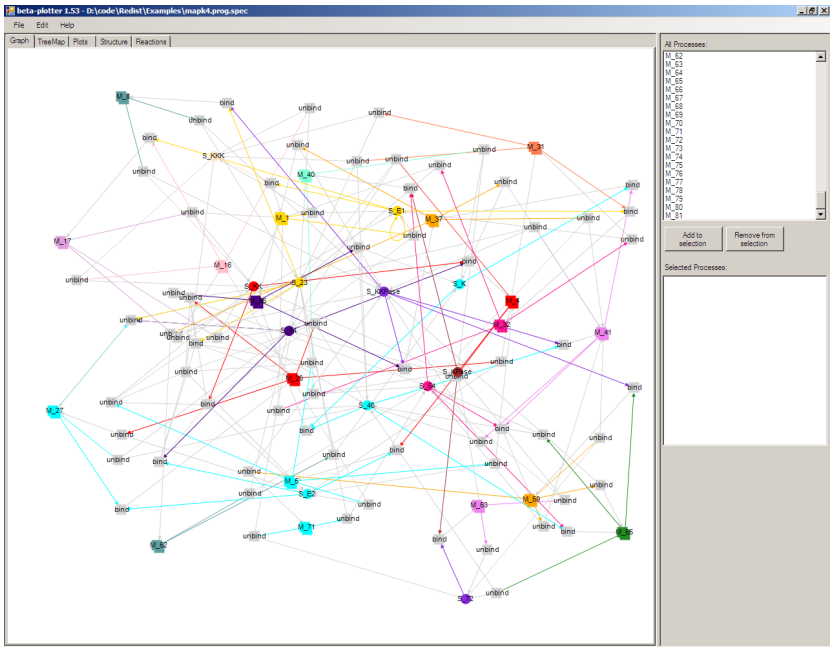


Figure 7: The Plotter displaying the result of a simulation of the MAPK cascade with the BWB simulator.

The core BWB is enriched by two tools for input/output operations: the BWB designer and the BWB plotter.

The BWB designer is a tool that allows to write BlenX programs both in a textual and in a graphical way. The two representations are interchangeable: the tool can parse and generate the graphical representation from any valid BlenX program, and generate the textual representation from the graphical form (see the upper left and the bottom left parts of Fig. 5). In particular, it is possible to draw boxes, pi-processes, interactions, events and to form complexes using graphs (see Figures 5 and 6). The textual representation can then be used as input to the core BWB.

The BWB plotter is a graphical tool that parses and display simulation outputs as changes in concentrations (Fig. 7), graphs of the reactions executed by the simulator (Fig. 8) and other views of the relations between entities and reactions. The BWB plotter provides to the user a picture of the dynamic behaviour of a simulated model and the topology of the network that originated that behaviour.

3 Stochastic rates

The *stochastic approach* to chemical kinetics has been proved to be grounded on a physical base; early experimental studies (see as e.g. [25], [24]) have demonstrated that stochastic effects can be significant in cellular reactions. More recent experimental studies show the importance of noise in gene regulation:

The proliferation of both noise and noise reduction systems is a hall-

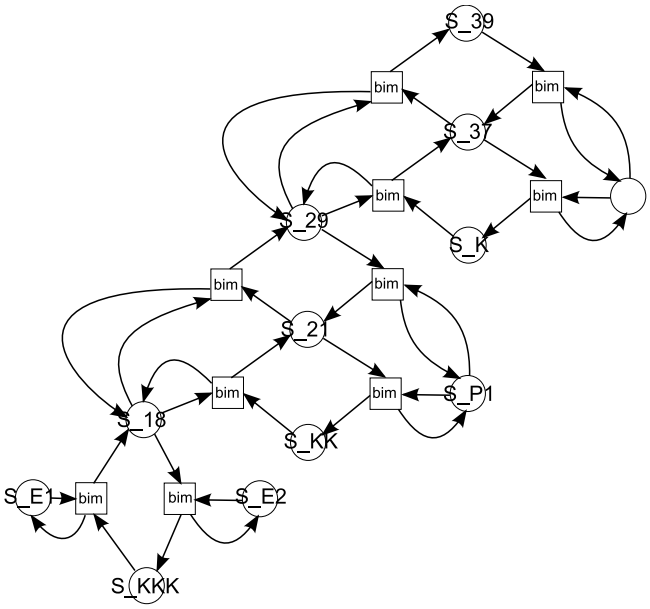


Figure 8: The Plotter displaying the graphs of reactions executed during a simulation.

> *mark of organismal evolution* – Federoff et al. [6]
>
> *Transcription in higher eukaryotes occurs with a relatively low frequency in biologic time and is regulated in a probabilistic manner* – Hume [16]
>
> *Gene regulation is a noisy business* – Mcadams et al. [17]

These studies, together with the success of Monte Carlo stochastic simulation techniques in the quantum physics simulation, have ignited widespread interest in stochastic simulation techniques for biochemical networks.

The stochastic approach to chemical kinetics was first employed by Delbruck in the '40s. The basic assumptions of this approach are that a chemical reaction occurs when two (or more) molecules of the right type collide in an appropriate way, and that these collisions in a system of molecules in thermal equilibrium are *random*.

Moreover, Gillespie in [10, 11] makes some simplifying assumptions to avoid difficulties generated by the usual procedure of estimating the collision volume for each particle; he assumes that the system is in thermal equilibrium. This assumption means that the considered system is a well-stirred mixture of molecules. Furthermore, the assumption that the number of non-reactive collisions is much higher than the number of chemical reactions makes it possible to state that the molecules are randomly and uniformly distributed at all times.

All stochastic methods rely on these assumptions; furthermore, we can observe that biological systems can be modeled on different levels of abstraction, but models at each level follow the same pattern:

- pairs *entity type, quantity*;

- interactions between the entities.

For example, in the case of biochemical models *entities* are molecules and *interactions* are coupled chemical reactions.

Therefore we can reduce the parameters needed to describe a system to:

- the *entities*, usually referred to as *species*, present in the system $S_1, ..., S_N$;
- the number and type of *interactions*, called *reaction channels*, through which the molecules interact $R_1, ..., R_M$;
- the state vector $\mathbf{X}(t)$ of the system at time t , where $X_i(t)$ is the number of molecules of species S_i present at time t.

The state vector $\mathbf{X}(t)$ is a vector of random variables, that does not permit to track the position and velocity of the single molecules.

3.1 Base rate and actual rate

For each reaction channel R_j a function a_j, called the *propensity function* for R_j, is defined as:

$$a_\mu = h_\mu c_\mu \text{ for } \mu = 1, \ldots, M \tag{1}$$

such that h_μ is the number of distinct reactant combinations for reaction R_μ and c_μ is a constant depending on physical properties of the reactants and

$$a_0 = \sum_{\mu=1}^{M} a_\mu$$

The c_μ constant is usually called *base rate*, or simply *rate* of an action, while the value of the function a_μ is called the *actual rate*.

Gillespie derives a physical correct *Chemical Master Equation* (CME) from the above representation of biochemical interactions. Intuitively, this equation shows the stochastic evolution of the system over time, which is indeed a Markov process.

Gillespie also presented in [11] an exact procedure, called *exact stochastic simulation*, to numerically simulate the stochastic time evolution of a biochemical system, thus generating one single trajectory. The procedure is based on the *reaction probability density function* $P(\tau, \mu)$, which specifies the probability that the next reaction is an R_μ reaction and that it occours at time τ. The analytical expression for $P(\tau, \mu)$ is:

$$P(\tau, \mu) = \begin{cases} a_\mu \exp(-a_0 \tau) & \text{if } 0 \leq \tau < \infty \text{ and } \mu = 1, \ldots, M \\ 0 & \text{otherwise} \end{cases}$$

where a_μ is the *propensity function*.

The *reaction probability density function* is used in a stochastic framework to compute the probablility of an action to occour. The way of computing the combinations h_μ, and consequentely the *actual rate* a_μ, varies with the different kind of reactions we consider.

3.1.1 Rate of a *monomolecular* reaction:

the simplest kind of reactions we can encounter are first-order reactions, usually referred to as *monomolecular reactions*, that take the form:

$$S_1 \rightarrow S_2 ... S_n$$

In this case, the number of combinations h_μ is equal to n, where n is the number of entities (the *cardinality*) of S_1.

3.1.2 Rate of a *bimolecular* reaction:

second-order reactions, usually referred to as *bimolecular reactions*, take the form:

$$S_1 + S_2 \rightarrow S_3 ... S_n \quad \text{or} \quad S_1 + S_1 \rightarrow S_2 ... S_n$$

The second case explicitly consider the fact that the two elements reacting are indeed of the same species, as in homodimerization reactions.

To obtain h_μ, we have to compute the number of all possible interactions that can take place between elemets of the first species and elements of the second species. Let n be the *cardinality* of the species S_1, and m the cardinality of the species S_2.

In the former case, the number of combinations h_μ is equal to $n \cdot m$, while in the latter the number of combinations h_μ is equal to $\frac{n \cdot (n-1)}{2}$.

3.1.3 Constant rates:

constant rates are used when the number of combinations h_μ is not meaningful; in this case $h_\mu = 1$, so the base rate constant c_μ is directly used as the exponent of the exponential distribution form which a time of execution will be sampled.

3.1.4 Rate functions:

the computation of the *reaction probability density function* has been proved by Gillespie to be *exact*, in the sense that a Monte Carlo simulation of the method represents a random walk that is an unbiased realization of the master equation.

However, when a specie represents a higher aggregation entity (e.g. a cell) then the input-output relation can exhibit a non-linear behaviour (e.g. sigmoidal dose-responses for signaling molecules). In this case, we let the user specify a *rate functions*, that is used in place of the Gillespie method to compute the propensity function.

Note that in this case the proof that the method, and so the algorithm, is *exact* does not hold anymore. It is up to the user that choose a rate function demonstrate that the assumptions he/she made are realistic and that the produced results are correct. We are only providing the BlenX programmer with the highest flexibility in specifying the quantitative parameters that drive the simulation engine.

4 The Language

A BlenX program is made of an optional *declaration* file for the declaration of user-defined constants and functions, a *binder definition* file that associates

unique identifiers to binders of entities used by the program and a *program* file, that contains the program structure.

All the BlenX files share the syntax definition of identifiers, numbers and rates as reported below:

$$
\begin{array}{lll}
Letter & ::= & [a-zA-Z] \\
Digit & ::= & [0-9] \\
Exp & ::= & [Ee][+\backslash -]?\{Digit\} \\
real1 & ::= & \{Digit\}^{+}\{Exp\} \\
real2 & ::= & \{Digit\}^{*}.\{Digit\}^{+}(\{Exp\})? \\
real3 & ::= & \{Digit\}^{+}.\{Digit\}^{*}(\{Exp\})? \\
 & & \\
Real & ::= & real1 \mid real2 \mid real3 \\
Decimal & ::= & \{Digit\}^{+} \\
Id & ::= & (\{Letter\}|\_)(\{Letter\}|\{Digit\}|\_)^{*} \\
 & & \\
number & := & Real \mid Decimal \\
 & & \\
rate & := & number \mid \textbf{rate} \ (\ Id\ ) \mid \textbf{inf}
\end{array}
$$

Note that in the following sections, during the description of the programming constructs, we prefix qualifying words to Id in order to clarify the kind of identifier that can occour in a given position. We will write $boxId$, $binderId$, $funcId$ and $varId$ to specify identifiers referring to boxes, binders, functions and variables respectively. Syntactically, they are all equal to Id; the disambiguation is done by the BlenX compiler using a symbol table. For examples, if an identifier Id is used in a function declaration, it will be stored as a $funcId$ in the symbol table.

4.1 The declaration file

A declaration file is a file with *.decl* extension that contains the definition of variables, constants and functions. Since these constructs are optional, it is possible to skip the definition of the whole file. The declaration file has the following syntax:

declarations	::=	
		decList
decList	::=	
		dec
	|	*dec decList*
dec	::=	
		let *Id* : **function** = *exp* ;
	|	**let** *Id* : **var** = *exp* ;
	|	**let** *Id* : **var** = *exp* **init** *number*;
	|	**let** *Id* (*number*) : **var** = *exp* ;
	|	**let** *Id* : **const** = *exp* ;
exp	::=	
		number
	|	*Id*
	|	| *Id* |
	|	**log** (*exp*)
	|	**sqrt** (*exp*)
	|	**exp** (*exp*)
	|	**pow** (*exp* , *exp*)
	|	*exp* + *exp*
	|	*exp* − *exp*
	|	*exp* ∗ *exp*
	|	*exp* / *exp*
	|	−*exp*
	|	+*exp*
	|	(*exp*)

An *expression* is made up of operators and operands. The syntax for the expression *exp* and the possible algebraic operators that can be used is given in the previous table. Operator precedence follows the common rules found in every programming language. + and − have the precedence when used as unary operators, while × and / have the precedence w.r.t. + and − when used as binary operators.

A *state variable* or simply *variable* is an identifier that can assume real modifiable values (Real value). The content of a variable is automatically updated when the defining expression *exp* changes; The content of the variable can also be changed by an **update** event (see Sec. 4.7). In this case, the function associated with the event is evaluated and the variable is updated with the resulting value. After the variable identifier and the **var** keyword, the user has to specify the expression used to control the value of the variable and an optional initial value after the **init** keyword. Examples of variable declarations follows:

```
let v1 : var = 10 * |A|;
let mCycB : var = 2 * |X| * log(v1) init 0.1;
```

In addition, we define another type of variables, called *continuous variables*. These variables depend on *time* and their value is still determined by an expression. Consider for example the following equation, commonly used to express

the growth of mass in a cell-cycle model:

$$\frac{\delta m}{\delta t} = \mu \cdot m$$

This equation expresses the continuous variation of mass during time. If we discretize it we obtain:

$$\frac{\Delta m}{\Delta t} = \mu \cdot m \quad \rightarrow \quad \Delta m = \mu \cdot m \cdot \Delta t$$

To update the m variable every Δt, we can write the following expression:

$$m_{t(i)} = m_{t(i-1)} + \Delta m \quad \rightarrow \quad m_{t(i)} = m_{t(i-1)} + (\mu \cdot m \cdot \Delta t)$$

The syntax to write the previous equation, given a Δt of 0.1, is:

```
let m(0.1): var = mu * m init 0.2;
```

More generally, in a *continuous variable* declaration the user has to specify the Id of the variable, immediately followed by the Δt value. The expression after the = sign is used to compute the delta value, with Δt implicit. Therefore, the declaration `let v(t): var = exp;` corresponds to the differential equation $\frac{\delta v}{\delta t} = exp$.

A *constant* is an identifier that assumes a value that cannot be changed at run-time and specified through a constant expression (an expression that does not rely on any variable or concentration $|Id|$ to be evaluated). As an extension, BlenX allows the use of *constant expressions*. Examples of constant declarations and of constant expressions follow:

```
let c1 : const = 1.0;
let pi : const = 3.14;
let c2 : const = (2.5 + 1) / (2.5 - 1);
let c3 : const = (4.0/3.0) * pi * pow(c1, 3);
let e: const = exp(1.0);
```

In the current version of BlenX, *functions* are parameterless and always return a Real value. As is, a function is only a named expression that can be used to evaluate a rate or to update the content of a state variable. An example of function definition follows:

```
let f1 : function =
   (k5s / alpha) / (pow( (J5 / (m * alpha * |X|) ) , 4) + 1);
```

Notice that when a program contains continuous variables, then the CTMC generation is not allowed.

4.2 The binder definition file

The binder definition file is a file with *.types* extension that stores all the binder identifiers that can be used in the declaration of binders (see Sec. 4.4) and the affinities between binders associated with a particular identifier.

Affinities are a peculiar feature of BlenX. The interaction mechanism of many biological modelling languages is based on the notion of exact complement of communication channel names, as in computer science modelling where two programs can interact only if they know the exact address of the interacting

partners. In BlenX instead interactions are guided by affinities between a pair of binder identifiers. There are three advantages in this approach: it allows us to avoid any global policy on the usage of names in order to make components interact; it relaxes the exact, or *key-lock*, style of interaction of exact name pairing; it permits a better separation of concerns, as it allows us to put interaction information in a separate file that can be modified or substituted without altering the program. The usage of affinities in a separate file is comparable to program interactions guided by *contracts* or service definitions, like in some web-service models (see [1]).

$$
\begin{array}{lll}
\textit{affinities} & ::= & \\
 & & \{\ \textit{binderIdList}\ \} \\
 & | & \{\ \textit{binderIdList}\ \}\%\%\{\ \textit{affinityList}\ \} \\
\textit{binderIdList} & : & \\
 & & \textit{binderId} \\
 & | & \textit{binderId},\ \textit{binderIdList} \\
 & & \\
\textit{affinity} & ::= & \\
 & & (\ \textit{binderId},\ \textit{binderId},\ \textit{rate}\ ) \\
 & | & (\ \textit{binderId},\ \textit{binderId},\ \textit{funcId}\ ) \\
 & | & (\ \textit{binderId},\ \textit{binderId},\ \textit{rate},\ \textit{rate},\ \textit{rate}\ ) \\
\textit{affinityList} & := & \\
 & & \textit{affinity} \\
 & | & \textit{affinity},\ \textit{affinityList}
\end{array}
$$

An *affinity* is a tuple of three or five elements. The first two elements are binder identifiers declared in the *binderIdList*, while the other elements can either be rate values or a single function identifier. If the affinity tuple contains a single rate value, then the value is interpreted as the base rate of *inter-communication* (Sec. 4.5.10) between binders with identifier equal to the first and second $binderId$ respectively.
If the affinity tuple contains three rate values, these values are interpreted as the base rate for *complex*, *decomplex* (see Sec. 4.6 for the definition of complexes) and *inter-complex communication* between binders with identifier equal to the first and second $binderId$ respectively.
When the element after the two $binderId$s is a function identifier, the expression associated to the function will be evaluated to yield a value, then interpreted as the rate of *inter-communication*.

4.3 The program file

The central part of a BlenX program is the program file. The program file has a *.prog* extension; it is generated by the following BNF grammar:

$$
\begin{array}{lll}
\mathit{program} & ::= & \\
 & & \mathit{info}\ \langle\langle\ \mathit{rateDec}\ \rangle\rangle\ \mathit{decList}\ \textbf{run}\ \mathit{bp} \\
 & | & \mathit{info}\ \mathit{decList}\ \textbf{run}\ \mathit{bp} \\
 & & \\
\mathit{info} & ::= & \\
 & & [\ \textbf{steps} = \mathit{decimal}\ ] \\
 & | & [\ \textbf{steps} = \mathit{decimal},\ \textbf{delta} = \mathit{number}\ ] \\
 & | & [\ \textbf{time} = \mathit{number}\ ] \\
 & & \\
\mathit{rateDec} & ::= & \\
 & & \mathit{Id} : \mathit{rate} \\
 & | & \textbf{CHANGE} : \mathit{rate} \\
 & | & \textbf{EXPOSE} : \mathit{rate} \\
 & | & \textbf{UNHIDE} : \mathit{rate} \\
 & | & \textbf{HIDE} : \mathit{rate} \\
 & | & \textbf{BASERATE} : \mathit{rate} \\
 & | & \mathit{rateDec}, \mathit{rateDec} \\
 & & \\
\mathit{decList} & ::= & \\
 & & \mathit{dec} \\
 & | & \mathit{dec}\ \mathit{decList} \\
 & & \\
\mathit{dec} & ::= & \\
 & & \textbf{let}\ \mathit{Id} : \textbf{pproc} = \mathit{process}\ ; \\
 & | & \textbf{let}\ \mathit{Id} : \textbf{bproc} = \mathit{box}\ ; \\
 & | & \textbf{let}\ \mathit{Id} : \textbf{complex} = \mathit{complex}\ ; \\
 & | & \textbf{let}\ \mathit{Id} : \textbf{prefix} = \mathit{actSeq}\ ; \\
 & | & \textbf{let}\ \mathit{Id} : \textbf{bproc} = \mathit{Id}\ \langle\langle\ \mathit{invTempList}\ \rangle\rangle\ ; \\
 & | & \textbf{when}\ (\ \mathit{cond}\ )\ \mathit{verb}\ ; \\
 & | & \textbf{template}\ \mathit{Id} : \textbf{pproc}\ \langle\langle\ \mathit{decTempList}\ \rangle\rangle = \mathit{process}\ ; \\
 & | & \textbf{template}\ \mathit{Id} : \textbf{bproc}\ \langle\langle\ \mathit{decTempList}\ \rangle\rangle = \mathit{box}\ ; \\
 & & \\
\mathit{bp} & ::= & \\
 & & \mathit{Decimal}\ \mathit{Id} \\
 & | & \mathit{Decimal}\ \mathit{Id}\ \langle\langle\ \mathit{invTempList}\ \rangle\rangle \\
 & | & \mathit{bp}\ ||\ \mathit{bp}
\end{array}
$$

A *prog* file is made up of an header *info*, an optional list of rate declarations (*rateDec*), a list of declarations *decList*, the keyword **run** and a list of starting entities *bp*.

The *info* header contains information used by the BWB simulator that will execute the program. A stochastic simulation can be considered as a succession of timestamped steps that are executed sequentially, in non-decreasing time order. Thus, the duration of a simulation can be specified as a **time**, intended as the maximum timestamp value that the simulation clock will reach, or as a number of **steps** that the simulator will schedule and execute. The **delta** parameter can be optionally specified to instruct the simulator to record events only at a certain frequency (and not every time and event is simulated).

A BlenX program is a stochastic program: every single step that the program can perform has a *rate* associated to it, representing the frequency at which that step can, or is expected to, occur. The *rateDec* specifies the global rate associations for individual channel names or for four particular *classes* of actions that a program can perform. In addition, a special class **BASERATE** can be used to set a common basic rate for all the actions that do not have an explicit rate set. The explicit declaration of a rate in the definition of an *action* has the precedence on this global association (see Sec. 4.4).

The list of declarations *decList* follows. Each declaration is a small, self-contained piece of code ended by a ';'. A declaration can be named, e.g. it can have an *Id* that designates uniquely the declaration unit in the program, or it can be nameless. Declarations of boxes, processes, sequences of prefixes and complexes must be named[2], while events are *nameless*.

4.4 Processes and Boxes

Boxes are generated by the following BNF grammar:

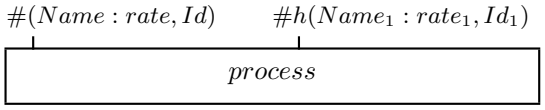
$$
\begin{array}{lll}
box & ::= & \\
 & & binders\ [\ process\ ] \\
 & & \\
binders & ::= & \\
 & & \#\ (\ Id\ :\ rate,\ Id\ ) \\
 & |\ & \#\ (\ Id,\ Id\ ) \\
 & |\ & \#\mathbf{h}\ (\ Id\ :\ rate,\ Id\ ) \\
 & |\ & \#\mathbf{h}\ (\ Id,\ Id\ ) \\
 & |\ & binders,\ binders \\
 & & \\
process & ::= & \\
 & & par \\
 & |\ & sum
\end{array}
$$

The intuition is that a box represents an autonomous biological entity that has its own control mechanism (the *process*) and some interaction capabilities expressed by the *binders*.

A *binders* list is made up of a non empty list of *elementary binders* of the form $\#(Id : rate, Id)$ (active with rate), $\#(Id, Id)$ (active without rate), $\#\mathbf{h}(Id : rate, Id)$ (inactive with rate), $\#\mathbf{h}(Id, Id)$ (inactive without rate), where the first Id is the *subject* of the binder, $rate$ is the stochastic parameter that quantitatively drives the activities involving the binder (hereafter, stochastic rate) and the second Id represents the identifier of the binder. Binder identifiers cannot occur in processes while subjects of binders can. The subject of an elementary beta binder is a binding occurrence that binds all the free occurrences of it in the process inside the box to which the binder belongs. Hidden binders are useful to model interaction sites that are not available for interaction although their status can vary dynamically. For instance a receptor that is hidden by the shape of a molecule and that becomes available if the molecule interacts with/binds to other molecules. Given a list of binders, we denote the set of all its subjects with $sub(binders)$. A box is considered *well-formed* if the

[2]Note that some language constructs, i.e. processes and sequences, can appear throughout a program without a name; they must be named only when they appear as a declaration

list of binders has subjects and identifiers all distinct. Well-formedness of each box defined in a BlenX program is checked statically at compile-time. Moreover, well-formedness is preserved during the program execution. The BlenX graphical representation of a box is:



Boxes are generated by the following BNF grammar:

$$
\begin{array}{lll}
process & ::= & \\
 & & par \\
 & | & sum \\
 & & \\
par & ::= & \\
 & & parElem \\
 & | & sum \mid sum \\
 & | & sum \mid par \\
 & | & par \mid sum \\
 & | & par \mid par \\
 & | & (\ par\ ) \\
 & & \\
sum & ::= & \\
 & & sumElem \\
 & | & sum + sum \\
 & | & (\ sum\ ) \\
 & & \\
sumElem & ::= & \\
 & & \textbf{nil} \\
 & | & seq \\
 & | & \textbf{if}\ condexp\ \textbf{then}\ sum\ \textbf{endif} \\
 & & \\
parElem & ::= & \\
 & & Id \\
 & | & Id\ \langle\langle\ invTempList\ \rangle\rangle \\
 & | & \textbf{rep}\ action\ .\ process \\
 & | & \textbf{if}\ condexp\ \textbf{then}\ par\ \textbf{endif} \\
 & & \\
seq & ::= & \\
 & & action \\
 & | & action\ .\ process \\
 & | & Id\ .\ process
\end{array}
$$

A process can be a *par* or a *sum*. The non-terminal symbol *par* composes through the binary operator | two processes that can concurrently, while the non-terminal symbol *sum* of the productions of *process* is used to introduce guarded choices of processes, composed with the operator +. The + operator act intuitively as an *or* operator, meaning that at a certain step a process offers a choice of different possible actions such that the execution of each of them eliminates the others. By the contrary, the | operator act intuitively as an *and* operator, meaning that processes composed by | run effectively in parallel.

Notice that we can put in parallel processes also with the constructs Id and $Id\ \langle\langle\ invTempList\ \rangle\rangle$, meaning that we are instantiating a template (see Section

4.9) or an occurrence of a process previously defined. As an example, consider the following sequence of processes definition:

```
let p1 : pproc = nil ;
let p2 : pproc = nil | p1 ;
```

Process *p2* is defined as a parallel composition of the **nil** process and an instance of the *p1* process. In BlenX the definition of a process can only rely on identifiers of previously defined processes. Mechanisms of recursive definitions and mutual recursive definitions are not admitted.
The **rep** operator is used to replicate copies of the process passed as argument. Note that we use only guarded replication, i.e. the process argument of the **rep** must have a prefix *action* that forbids any other action of the process until it has been consumed. The **nil** process does nothing (it is a deadlocked process), while the **if-then** statement allows the user to control, through an *expression*, the execution of a *process*. The non-terminal symbol *seq* identifies an action, a process prefixed by an action and a process prefixed by an Id. When in a program we have a process defined using the statement *Id.process* we statically check that the Id corresponds to a previously defined sequence of prefixes.

4.5 Actions

The actions that a process can perform are described by the syntactic category *action*.

$$
\begin{array}{lll}
action & ::= & \\
 & & Id\ !\ (\ Id\ ) \\
 & |\ & Id\ !\ () \\
 & |\ & Id\ ?\ (\ Id\ ) \\
 & |\ & Id\ ?\ () \\
 & |\ & \textbf{delay}\ (\ rate\ ) \\
 & |\ & \textbf{expose}\ (\ Id : rate\ ,\ Id\ ) \\
 & |\ & \textbf{hide}\ (\ Id\ ) \\
 & |\ & \textbf{unhide}\ (\ Id\ ) \\
 & |\ & \textbf{ch}\ (\ Id,\ Id\ ) \\
 & |\ & \textbf{expose}\ (\ rate,\ Id : rate,\ Id\ ) \\
 & |\ & \textbf{hide}\ (\ rate,\ Id\ ) \\
 & |\ & \textbf{unhide}\ (\ rate,\ Id\ ) \\
 & |\ & \textbf{ch}\ (\ rate,\ Id,\ Id\ )
\end{array}
$$

The first four actions are common to most process calculi. The first pair of actions represent an output/send of a value on a channel, while the second pair represent the input/reception of value or a signal on a channel. The remaining actions are peculiar of the BlenX language. The definition of *free names* for processes is obtained by stipulating that $Id?(Id').process$ is a binder for Id' in *process* and that $\textbf{expose}(Id : rate, Id).process$ and $\textbf{expose}(rate, Id : rate, Id).process$ are binders for Id in *process*. The definitions of *bound names* and of *name substitution* are extended consequently. The definition of free and bound names for boxes is obtained by specifying that the set of free names of a box $binders[process]$ is the set of free names of the *process* minus the set $sub(binders)$ of subjects of the binders. Moreover, as usual two processes *process* and *process*′ are α-equivalent if *process*′ can be obtained from *process* by renaming one or more bound names in *process*, and vice versa. As usual renaming avoids name clashes, i.e. a free name never becomes bound after the renaming. More details of this definitions can be found in [5, 21].

4.5.1 species:

In BlenX species are defined as classes of boxes which are *structurally congruent*. The structural congruence for boxes, denoted with $\equiv$, is the smallest relation which satisfies the following laws:

- $process \equiv process'$, if $process$ and $process'$ are α-equivalent
- $process \mid \mathbf{nil} \equiv process$
- $process_1 \mid (process_2 \mid process_3) \equiv (process_1 \mid process_2) \mid process_3$
- $process_1 \mid process_2 \equiv process_2 \mid process_1$
- $sum \mid \mathbf{nil} \equiv sum$
- $sum_1 \mid (sum_2 \mid sum_3) \equiv (sum_1 \mid sum_2) \mid sum_3$
- $sum_1 \mid sum_2 \equiv sum_2 \mid sum_1$
- $!action.process \equiv action.(process \mid !action.process)$
- $binders[process] \equiv binders[process']$, if $process \equiv process'$
- $binders, binders'[process] \equiv binders', binders[process]$
- $\#(Id : rate, Id_1), binders[process] \equiv \#(Id' : rate', Id_1), binders[process\{^{Id'}/_{Id}\}]$ if $Id' \notin sub(binders)$
- $\#(Id, Id_1), binders[process] \equiv \#(Id', Id_1), binders[process\{^{Id'}/_{Id}\}]$ if $Id' \notin sub(binders)$
- $\#\mathbf{h}(Id : rate, Id_1), binders[process] \equiv \#\mathbf{h}(Id' : rate', Id_1), binders[process\{^{Id'}/_{Id}\}]$ if $Id' \notin sub(binders)$
- $\#\mathbf{h}(Id, Id_1), binders[process] \equiv \#\mathbf{h}(Id', Id_1), binders[process\{^{Id'}/_{Id}\}]$ if $Id' \notin sub(binders)$

Consider for example the program:

```
...
let b1 : bproc = #(x:1,A)
  [ ( x!().nil + z?(w).w!().nil ) | x!(z).nil ];
...
let b2 : bproc = #(y:1,A)
  [ y!(z).nil | ( z?(t).t!().nil + y!().nil ) ];
...
```

In the example we have $b1 \equiv b2$, hence the boxes belong to the same species. Notice that if we have multiple definition of boxes that represent the same species, then at run-time they are collected together and the species name is taken from the first definition (e.g. in the example the name of the corresponding species is $b1$). Hereafter, when we say that in a particular state of execution of a program the cardinality of a box species $b1$ is n we mean that in that state of execution the number of boxes structurally congruent to $b1$ is n.

4.5.2 Intra-communication:

consider the following piece of code:

```
let p : pproc =
  x!(m).nil + y?(z).z?().nil + y?().nil ;

let b1 : bproc = #(x:1,A),#h(m,B)
  [ p | x?(z).z!(c).nil + x?().nil + y!().nil ];
```

Box *b1* has a binder $\#(x : 1, A)$ and an internal process defined as a parallel composition of the *sum* process p and the *sum* process $x?(z).z!(c).nil + y!().nil$.

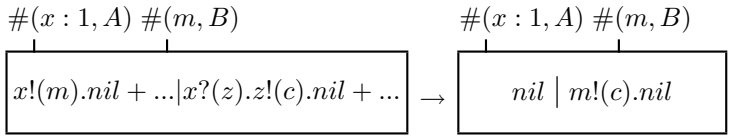
Each *sum* composes processes guarded by input or output actions. Parallel processes that perform complementary actions on the same channel inside the same box can synchronize and eventually exchange a message, generating an *intra-communication*. In the example, several intra-communications can be performed. Indeed, each output in the first *sum* can synchronize with an input on the same channel in the other *sum*, and vice-versa. Consider the input/output pair:

```
x!(m).nil + ... | x?(z).z!(c).nil + ...
```

$x?(z)$ represents an input/reception of something that will instantiate the placeholder z over channel x, while $x!(m)$ represent an output/send of a value m over channel x. The placeholder z in the input is a binding occurrence that binds all the free occurrences of z in the scope of the prefix $x?(z)$ (in this case in $z!(c).nil$). Sometimes the channel name x is called the subject and the placeholder/value z is called the object of the prefix. The execution of the intra-communication consumes the input and output prefixes and the object m of the output flows from the process performing the output to the one performing the input:

```
nil | m!(c).nil
```

The flow of information affects the future behavior of the system because all the free occurrences bound by the input placeholder are replaced in the receiving process by the actual value sent by the output (in the example z is substituted by m). The graphical representation of the intra-communication is



If an input has no object and it is involved in a intra-communication:

```
x!(m).nil + ... | x?().nil + ...
```

then the two prefixes are consumed and no substitution is performed:

```
nil | nil
```

If an output has no object and is involved in an intra-communication:

```
... + y?(z).z?().nil + ... | ... + y!().nil + ...
```

then the two prefixes are consumed and the substitution in the process prefixed by the input is performed by using a reserved string *$emp* on which no further intra-communication is allowed.

```
$emp?().nil | nil
```

Notice that the string *$emp* cannot be generated by the regular expression defining the Id (see Section 4).

If object-free outputs and inputs synchronize in an intra-communication:

```
... + y?().nil + ... | ... + y!().nil + ...
```

then the two prefixes are consumed, generating the process:

```
nil | nil
```

The stochastic nature of BlenX emerges in the above examples through the rates associated to the input/output channels. In particular, if the channel is bound to a binder, the rate is specified in the binder definition; if the binder is $\#(x:1,A)$ (or $\#h(x:1,A)$) the rate associated to an intra-communication over channel x is 1, while if the binder is $\#(x,A)$ (or $\#h(x,A)$) the associated rate is assumed to be 0 and hence no intra-communications over channel x can happen.

If the channel is not bound to a binder, then the rate has to be defined in the global *rateDec*. In particular, if *rateDec* is:

```
<< ... , x : 2.5 , ... >>
```

the rate associated to an intra-communication over channel x is 2.5. Instead, if no specific x rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile time error is generated. In the example, intra-communications over channel y need a specific definition or the **BASERATE** in the *rateDec* list.

Since to each communication channel in a box we can associate an unique rate r, then the overall propensity of performing an intra-communication on a channel x is given by the following formula:

$$r \times ((In(x) \times Out(x)) - Mix(x))$$

where $In(x)$ identifies all the enabled input on x, $Out(x)$ the enabled output on x and $Mix(x)$ all the possible combinations of input/output within the same *sum*. As an example, consider the box:

```
let b1 : bproc = #(x,A),#h(m,B)
   [ x?().nil + x!().nil + x!().nil |
     x?().nil + x!().nil + x!().nil ]
```

Let the rate associated to x be 3, the overall propensity associated to an intra-communication on the channel x is calculated using the previous formula obtaining:

$$3 \times ((2 \times 4) - 4) = 12$$

where term (2×4) represents all the combinations of input/output and the last 4 represents the combinations contained in the same *sum* and hence the ones that cannot give raise to an inter-communication.

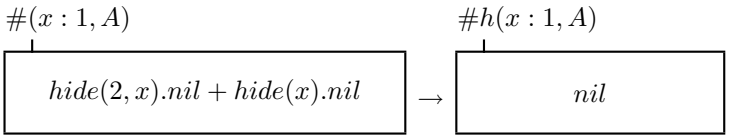
Notice that multiplying 12 by the cardinality of the species $b1$ we obtain the overall propensity that a box of that species performs an intra-communication on channel x.

4.5.3 hide:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ hide(2,x).nil + hide(x).nil ]
```

Box $b1$ can perform two *hide* actions. The execution of both actions cause the modification of the box interface hiding the binder $\#(x:1,A)$. The graphical representation of the actions is



The only difference between the actions is the stochastic rate association. Indeed, the first action specifies its own rate and hence is performed with a rate of value 2. For the second action, a rate has to be defined in the global *rateDec*. In particular, if *rateDec* is:

```
<< ... , HIDE : 4 , ... >>
```

the rate associated to the all hide actions is 4. Instead, if no specific **HIDE** rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile-time error is generated.

To compute the overall propensity associated to *hide* actions performed by boxes of a given species, we need to calculate all the possible combinations. This combination is obtained by multiplying the number of all the enabled hide actions $hide(r, x)$ on the same binder with the same rate r and the number of all the enabled hide actions $hide(x)$ on the same binder by the corresponding base rates. The overall propensity is then obtained by multiplying this combination with the cardinality of the species.

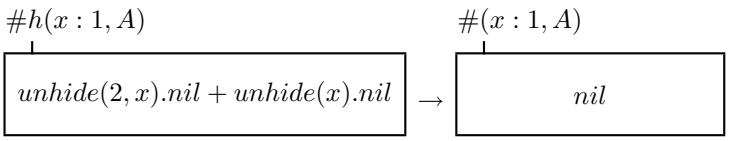
Notice that an hide action on an binder which is already hide is not enabled. A definition of an hide action on a name which is not a binder is not enabled and generates a compile-time warning.

4.5.4 unhide:

consider the following box:

```
let b1 : bproc = #h(x:1,A)
   [ unhide(2,x).nil + unhide(x).nil ]
```

Box $b1$ can perform two *unhide* actions. The execution of both actions cause the modification of the box interface unhiding the binder $\#h(x : 1, A)$. The graphical representation of the actions is



The only difference between the actions is the stochastic rate association. Indeed, the first action specifies its own rate and hence is performed with a rate of value 2. For the second action, a rate has to be defined in the global *rateDec*. In particular, if *rateDec* is:

```
<< ... , UNHIDE : 4 , ... >>
```

the rate associated to the hide action is 4. Instead, if no specific **UNHIDE** rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile time error is generated.

To compute the overall propensity associated to *unhide* actions performed by boxes of a given species, we need to calculate all the possible combinations. This combination is obtained by multiplying the number of all the enabled unhide actions $unhide(r, x)$ on the same binder with the same rate r and the number of all the enabled unhide actions $unhide(x)$ on the same binder by the corresponding base rates. The overall propensity is then obtained by multiplying this combination with the cardinality of the species.

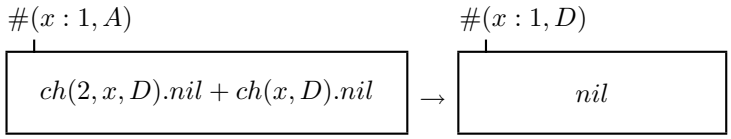
Notice that an unhide action on an binder which is already unhidden is not enabled and that a definition of an unhide action on a name which is not a binder is not enabled and generates a compile-time warning.

4.5.5 change:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ ch(2,x,D).nil + ch(x,D).nil ]
```

Box $b1$ can perform two *change* actions. The execution of both actions cause the modification of the box interface changing the value A of the binder $\#(x : 1, A)$ into D. The graphical representation of the actions is



The first action specifies its own rate and hence is performed with a rate of value 2. For the second action, a rate has to be defined in the global *rateDec*. In particular, if *rateDec* is:

```
<< ... , CHANGE : 4 , ... >>
```

the rate associated to the hide action is 4. Instead, if no specific **CHANGE** rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile time error is generated.

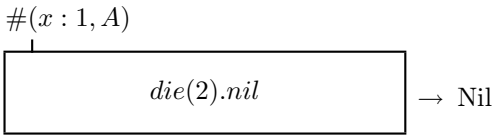
To compute the overall propensity associated to *change* actions performed by boxes of a given species, we need to calculate all the possible combinations. This combination is obtained by multiplying the number of all the enabled change actions $ch(r, x, D)$ on same values and the number of all the enabled change actions $ch(x, D)$ on same binders and with equal substituting types by the corresponding base rates. The overall propensity is then obtained by multiplying this combination with the cardinality of the species.

4.5.6 die:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ die(2).nil ]
```

Box $b1$ can perform a *die* action. The execution of the action eliminates the related box. The graphical representation of the action is



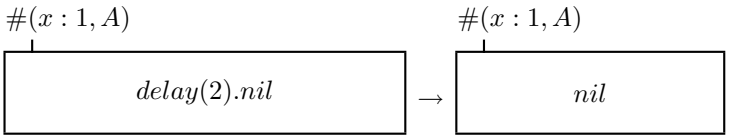
The action is executed with the specified rate of value 2. To compute the overall propensity associated to *die* actions we calculate the number of all the enabled die actions $die(r)$ on same rates and multiply this values by the corresponding base rates and by the cardinality of the species.

4.5.7 delay:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ delay(2).nil ]
```

Box $b1$ can perform a *delay* action. The execution of the action allows the box to evolve internally. The graphical representation of the action is



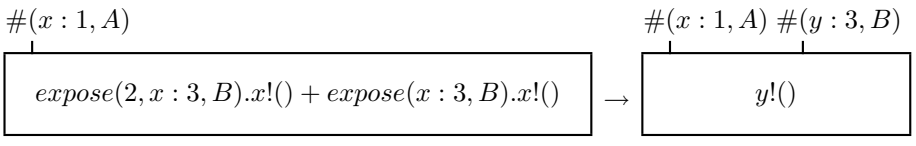
The action is executed with the specified rate of value 2. Moreover, Nil is used to identify a deadlocked box which does nothing. To compute the overall propensity associated to *delay* actions we calculate the number of all the enabled delay actions $delay(r)$ on same rates and multiply this values by the corresponding base rates and by the cardinality of the species.

4.5.8 expose:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ expose(2,x:3,B).x!() + expose(x:3,B).x!() ]
```

Box $b1$ can perform two *expose* actions. The execution of both actions add a new binder $\#(y : 3, B)$ to the interface, by renaming the subject into a new name to avoid clashes of names (x renamed into y with all the occurrences bound by the subject in the expose). The graphical representation of the actions is



The first action specifies its own rate and hence is performed with a rate of value 2. For the second action, a rate has to be defined in the global *rateDec*. In particular, if *rateDec* is

```
<< ... , EXPOSE : 4 , ... >>
```

the rate associated to the hide action is 4. Instead, if no specific **EXPOSE** rate definition appears in the *rateDec* list, then the **BASERATE** definition is used. If also no **BASERATE** definition appears in the *rateDec* list, then a compile-time error is generated. Expose actions are considered separately and hence the overall propensity that a box species perform an expose action is calculated multiplying the rate associated to the action by the action rates and by the cardinality of the box species performing the action.

Notice that an expose action of a binder identifier which is already present in the set binders of the box is not enabled.

4.5.9 if-then statement:

consider the following box:

```
let b1 : bproc = #(x:1,A)
   [ if (x,unhidden) and (x,A) then x!().nil ]
```

Box $b1$ can perform the output action $x!()$ only if the conditional expression is satisfied by the actual configuration of the binders of the box containing the if-then statement. In this example if the binder with subject x is unhidden and its binder identifier is A, then the output can be executed. The general form of the conditional expressions of if-then statements are generated by the following BNF grammar:

condexp ::=
- *atom*
- | *condexp* **and** *condexp*
- | *condexp* **or** *condexp*
- | **not** *condexp*
- | (*condexp*)

atom ::=
- (*Id*, *Id*)
- | (*Id*, **hidden**)
- | (*Id*, **unhidden**)
- | (*Id*, **bound**)
- | (*Id*, *Id*, **hidden**)
- | (*Id*, *Id*, **unhidden**)
- | (*Id*, *Id*, **bound**)

Conditional expressions are logical formulas built atoms (conditions on binder states) connected by classical binary logical operators (*and*,*or*,*not*). In the atoms the first *Id* identifies the subject of a binder, while the second *Id* (if present) identifies the binder identifier. The keywords *hidden*, *unhidden* and *bound* identify the three states in which a binder can be. As an example, the conditional expression:

```
(x,A) and ( not(y,B,hidden) or (z,bound) )
```

is satisfied only if the box has a binder with subject x of type A and has a binder with subject y which is not hidden and with type different from B or has a bound binder with subject z (see Section 4.6). Notice that boxes of the form:

```
let b1 : bproc = #(x:1,A)
   [ if (y,unhidden) and (x,A) then x!().nil ]
```

```
let b1 : bproc = #(x:1,A)
   [ y?(x).if (x,unhidden) and (x,A) then x!().nil ]
```

generates compile-time warnings. Indeed, in the first case the $(y, unhidden)$ do not refer to any binder of the box, while in the second case the atom $(x, unhidden)$ is bound by the input $y?(x)$ and not by the subject of the binder. In general, at run-time atoms on binders which are not present are evaluated as false value.

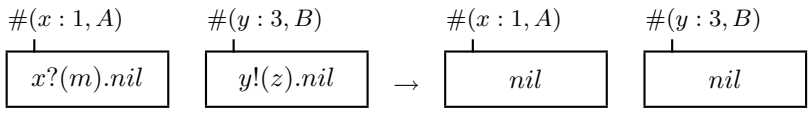
4.5.10 inter-communication:

processes in different boxes can perform an *inter- communication* (distinct from the *intra-communication* described above) if one sends a value y over a link x that is bound to an active binder of the box $\#(x : r, A)$ and a process in another box is willing to receive a value from a *compatible* binder $\#(y : s, B)$ through the action $y!(z)$. The two corresponding binders are compatible if a *compatibility* value (i.e. a stochastic rate) greater than zero is specified in the binder declaration file

```
{...,A,...,B,...}
%%
{ ... , (A,B,2.5), ... }
```

Note that intra-communications occur on perfectly symmetric input/output pairs that share the same subject, while inter-communication can occur between primitives that have different subjects provided that their binder identifiers are compatible. This new notion of communication is particularly relevant in biology where interactions occur on the basis of sensitivity or affinity which is usually not exact complementarity of molecular structures. The same substance can interact with many other in the same context, although with different levels of affinity expressed through different properties.

The graphical representation of an inter-communication is:



If the compatibility is specified by a stochastic rate, the overall propensity of the inter-communication is computed as bimolecular rate (see Section 3), considering all the possible combinations of inputs on channel x in the first box and outputs on y in the second box and multiplying this value with the product of the cardinality of the box species in the system. As an example consider the program:

```
...
let b1 : bproc = #(x:1,A)
  [ x!().nil + x!().nil | x!().nil ];
...
let b2 : bproc = #(y:3,B)
  [ y?().nil | y?().nil ];
...
let b3 : bproc = #(z:2,C)
  [ z?().nil ];
run 10 A || 20 B || 5 b3
```

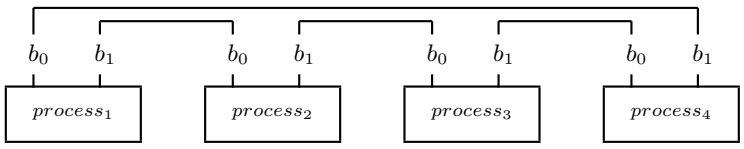


Figure 9: Example of complex.

Assuming boxes $b1$ and $b2$ defines two different species, the overall propensity of the inter-communication on boxes species A and B is

$$(2.5 \times (3 \times 2)) \times (10 \times 20)$$

where 2.5 is the basal rate, (3×2) is the number of combinations of inputs and outputs and (10×20) is the product of the cardinality of the two box species.

If the compatibility is expressed by a function defined in the declaration file:

```
{...,A,...,B,...}
%%
{ ..., (A,B,f1), ... }
```

then the overall propensity of the inter-communication is computed as a *rate function* (see Section 3) and therefore it does not depend directly on the cardinality of the involved species. In the example, if the function $f1$ is as:

```
...
let f1 : function = 2 * pow(|b3|,2);
...
```

the overall propensity of the inter-communication has value 50.

Notice that in an inter-communication, values corresponding to binder subjects cannot be sent.

4.6 Complexes

A complex is a graph-like structure where boxes are nodes and dedicated communication bindings are edges. Figure 9 report an example is reported, where $b_0 = \#(x : r_0, A_0)$ and $b_1 = \#(y : r_1, A_1)$. In BlenX, complexes are not defined as *species*, but as graph-like structures of box species. Complexes can be created automatically during the program execution or they can be instantiated also in the initial program. A complex can be defined using the following BNF

grammar:

$$
\begin{array}{lll}
\textit{complex} & ::= & \\
 & & \{\ (\ \textit{edgeList}\ )\ ;\ \textit{nodeList}\ \} \\
 & & \\
\textit{edgeList} & ::= & \\
 & & \textit{edge} \\
 & | & \textit{edge},\ \textit{edgeList} \\
 & & \\
\textit{edge} & ::= & \\
 & & (\ Id,\ Id,\ Id,\ Id\ ) \\
 & & \\
\textit{nodeList} & ::= & \\
 & & \textit{node} \\
 & | & \textit{node nodeList} \\
 & & \\
\textit{node} & ::= & \\
 & & Id\ :\ Id = (\ \textit{complBinderList}\ )\ ; \\
 & | & Id = Id\ ; \\
 & & \\
\textit{complBinderList} & ::= & \\
 & & Id \\
 & | & Id,\ \textit{complBinderList}
\end{array}
$$

A complex is created by specifying the list of edges (*edgeList*) and the list of nodes (*nodeList*). Each *edge* is a composition of 4 *Ids*. The first and the third identifiers represent node names, while the others represent subject names. Each *node* in the *nodeList* associates to a node name the corresponding box name and specifies the subjects of the bound binders. As an example, consider the program:

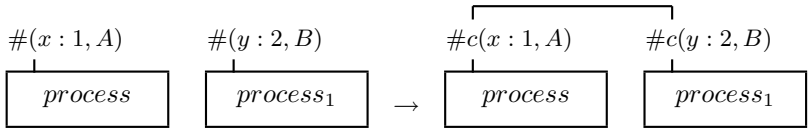
```
...
let b1 : bproc = #(x:r0,A0),#(y:r1,A1)
  [ x!().nil ];
...
let b2 : bproc = #(x:r0,A0),#(y:r1,A1)
  [ y!().nil ];
...
let C : complex =
{
  (
    (Box0,y,Box1,x),(Box1,y,Box2,x),
    (Box2,y,Box3,x),(Box3,y,Box0,x)
  );
  Box0:b1=(x,y);
  Box1:b2=(x,y);
  Box2=Box0;
  Box3=Box1;
}
...
```

The complex C defines a complex with a structure equivalent to the one reported in Figure 9. A complex can also be generated automatically at run-time

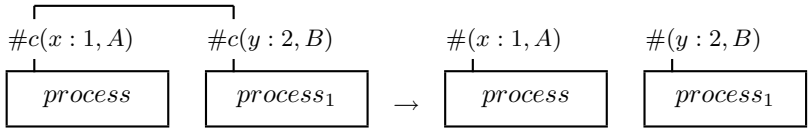
thorough a set of primitives for complexation and decomplexation. The ability of two boxes to form and break complexes is defined in the bind declaration file by specifying for pairs of binder identifiers triples of stochastic rates:

```
{...,A,...,B,...}
%%
{ ..., (A,B,1.5,2.5,10), ... }
```

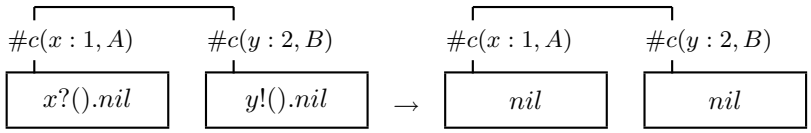
Complex and *decomplex* operations create and delete dedicated communication bindings between boxes. The biological counterpart of this construct is the binding of a ligand to a receptor, or of an enzyme to a substrate through an active domain. Given two boxes with binder with identifiers A and B respectively, the *complex* operation creates, with rate 1.5, a dedicated communication binding:



while the *decomplex* operation deletes, with rate 2.5, an already existing binding:



Finally, the *inter-complex communication* operation enables, with rate 10, a communication between complexed boxes through the complexed binders:



Notice that a binder in *bound* status is identified by $\#c(y : B)_s$ where c means that the corresponding box is part of a complex. It is important to underline that, although the bound status cannot be explicitly specified trough the syntax of the language and is used only as an internal representation, a binder in bound status is different from a hidden or unhidden binder and hence the structural congruence definition has to be extended accordingly:

- $\#\mathbf{c}(Id : rate, Id_1), binders[process] \equiv \#\mathbf{c}(Id' : rate', Id_1), binders[process\{^{Id'}/_{Id}\}]$ if $Id' \notin sub(binders)$
- $\#\mathbf{c}(Id, Id_1), binders[process] \equiv \#\mathbf{c}(Id', Id_1), binders[process\{^{Id'}/_{Id}\}]$ if $Id' \notin sub(binders)$

4.7 Events

Events specify statements, or *verbs*, to be executed with a specified rate and/or when some conditions are satisfied. A single *event* is the composition of a condition *cond* and an action *verb* (recall the syntax of declarations in Sect. 4.3).

$$
\begin{array}{lll}
dec & ::= & \\
 & | & \ldots \\
 & | & \textbf{when} \ (\ cond\ )\ verb\ ; \\
 & | & \ldots
\end{array}
$$

4.7.1 Conditions.

Events are used to express actions that are enabled by global conditions, expressed by *cond*. Conditions are used to trigger the execution of an event when some elements are present in the system, when a particular condition is met, with a given rate, or at a precise simulation time or simulation step.

$$
\begin{array}{lll}
cond & ::= & \\
 & & entityList\ :\ EvExpr\ :\ rate \\
 & | & entityList\ :\ EvExpr\ :\ funcId \\
 & | & entityList\ :\ EvExpr\ : \\
 & | & entityList\ ::\ rate \\
 & | & entityList\ ::\ funcId \\
 & | & :\ EvExpr\ : \\
 & & \\
entityList & ::= & \\
 & & boxId \\
 & | & boxId,\ entityList \\
 & & \\
EvAtom & ::= & \\
 & & |\ Id\ | = Decimal \\
 & | & |\ Id\ | < Decimal \\
 & | & |\ Id\ | > Decimal \\
 & | & |\ Id\ | \ != Decimal \\
 & | & \textbf{time} = Real \\
 & | & \textbf{steps} = Decimal \\
 & | & stateOpList \\
 & & \\
EvExpr & ::= & \\
 & & EvAtom \\
 & | & EvExpr\ \textbf{and}\ EvExpr \\
 & | & EvExpr\ \textbf{or}\ EvExpr \\
 & | & \textbf{not}\ EvExpr \\
 & | & (\ EvExpr\ )
\end{array}
$$

More precisely, a condition *cond* consists of three parts: *entityList*, a list of boxes present in the system; an expression used to enable or disable the event; a *rate* or rate function, used to stochastically select and include them in the set of standard interaction-enabled actions.

EvExpr can be combined through logical operators starting from atoms; furthermore, a condition can specify both an *EvExpr* and a rate (see definition of *cond*), so that we can simultaneously address rates and conditions (e.g. on structures and concentrations of species). As an example, consider the following event:

```
when(A, B : (|A| > 2 and |B| > 2) : rate(r1)) join (C);
```

The entities involved in the event are A and B, as they appear in the *entityList*; moreover, the *EvExpr* requires the cardinality of both the species identified by

boxes A and B to be greater than two, so the event will fire only when there are at least two A and two B in the system. When the condition is satisfied, the event will fire with rate $r1$.

The *EvAtom*s evaluate to the boolean values *true* and *false*, and can be used to express conditions over concentrations of species identified by an Id ($\mid Id \mid op\ Decimal$, where $op \in <, >, =, != $) or over simulation time or simulation steps.

A condition on *simulation time* will be satisfied as soon as the simulation clock is greater or equal to the specified time; a conditions on *simulation steps* will be satisfied as soon as the step count will exceed the number specified in the *EvAtom*. In both cases, the condition will remain *true* until the event is fired. So, events for which the only condition specified is the number of steps or the execution time are guaranteed to fire exactly once. For example, the event:

```
when(A : time = 3.0 : inf) delete;
```

will fire as soon as the simulation clock reaches 3.0, removing one A form the system.

It is important to make a remark: *Id*s that can appear in the *EvExpr* must be entities that appear in the *entityList*. The following code:

```
when(A, B : (|C| > 2) : rate(r1)) join (C);
```

will produce a compilation error. The only exception is when the *entityList* is empty (the sixth case in the BNF declaration of *cond*). In this case, the *Id*s in the expression can be chosen among all the $betaId$s or $varId$s already declared, with no restrictions.

If more complex expressions are needed (i.e. for expressing conditions on more species in the system) it is possible to use a *rate function* instead (see Sect. 4.1).

Note that the number of *Id*s specified in the *entityList* depends on the event verb that is used for the current event. See the next section for more details on this point.

Events, like all the other actions that can trigger an execution in a BlenX program, can have an associated rate. It is possible to specify both rate constants (form 1, 4 in the BNF specification of *cond*) or rate functions (form 2, 5 in the BNF specification of *cond*). The rate constants are treated differently in the case of events with or without explicit *EvExpr*s. When there is no *EvExpr*, the rate is computed as a monomulecular or bimolecular rate, using the concepts introduced in Sec. 3. In the monomolecular case, the number h_μ of reactant combinations is equal to the cardinality of the species designated by the unique box in the *entityList*, in the bimolecular case the number h_μ of reactant combinations is the product of the cardinalities of the species designated by the first and second box in the *entityList*.

When a condition is present, the rate is a *constant rate* (see Sec. 3.1.3). This is to avoid the case in which a decimal value used in a comparison operation in a *EvAtom* can influence the rate of that action. Consider the two following pieces of code:

```
when (A : |A| > 2 : r) delete(2);
```

and

```
when (A : |A| > 10 : r) delete(2);
```

The second event will be triggered when there is an higher concentrations of boxes of species A, ten in this case. If we use the monomulecular way of computing the actual rate, the second event will be triggered with an higher rate than the first one, as monomolecular rates are proportional to the reactants concentration. What we intuitively expect, however, is that the two actions will take place with the same *actual rate*, hence the event rate is considered as a constant rate. Consider also the following example:

```
when (A : |A| = 0 : r) new;
```

Intuitively, this event introduces a box of species A with a given rate when there are no such entities in the system. If we compute the rate in the usual way, the event will be never executed (which is clearly different form what we expect).

For the case in which rates are specified as functions (form 2, 5 in the BNF specification of *cond*), the function is evaluated and the resulting value is used directly to compute the propensity function (see Sec. 3.1.4).

4.7.2 Verbs.

Events can split an entity into two entities, join two entities into a single one, inject or remove entities into/from the system. Events are feature is essential to program perturbation of the systems triggered by particular conditions emerging during simulation and to observe how the overall behaviour is affected. An example could be the knock-out of a gene at a given time.

verb ::=
- **split** (*boxId*, *boxId*)
- | **join** (*boxId*)
- | **new** (*Decimal*)
- | **delete** (*Decimal*)
- | **new**
- | **delete**
- | **update** (*varId*, *funcId*)

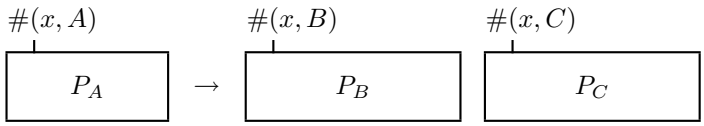
Verbs and conditions have some dependencies: not all verbs can apply to all conditions. The *entityList* in *cond* is used by the event to understand which species the event will modify; at the same time, the *verb* dictates which action will take place. Indeed, a *verb* specify how many entities will be present in the *entityList*:

- the **split** verb requires exactly one entity to be specified in the condition list;
- the **join** verb requires exactly two entities to be specified in the condition list;
- the **new** and **delete** verbs requires exactly one entities to be specified in the condition list;
- the **update** verb requires that the condition list is empty (form 6 in the BNF specification of *cond*).

The **split** verb removes one box of the specified species from the system, and substitutes it with the two other entities specified in the (*boxId*, *boxId*) pair. In the following piece of code:

```
when(A :: r) split(B, C);
```

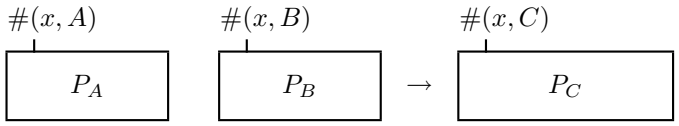
One A will be substituted by one B and one C, leading to the following behaviour:



The **join** verb removes two boxes, one for each of the species specified in the list, from the system, and introduces on box of the species specified in its ($boxId$) argument:

```
when(A, B :: r) join(C);
```

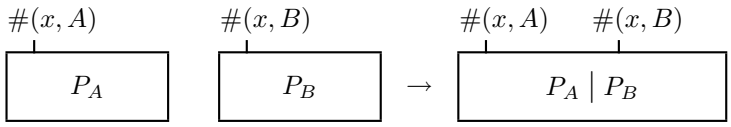
One A and one B will be joined in one C, leading to the following behaviour:



The target of the join, i.e. the box specified as argument, is optional:

```
when(A, B :: r) join;
```

If no box is specified, a new box, automatically generated form two originating boxes, will be introduced into the system:



The new box will have as the interface the union of the interfaces of boxes A and B, and as its internal process the parallel composition of the internal processes of A and B.

The **new** and **delete** verbs introduce and remove boxes. **New** will introduce into the system one copy (in its parameterless variant) or n copies (in its second variant) of the single entity present in the event list. As for the other events, the event is triggered with a certain rate and/or with a condition expression is met. The behaviour of **delete** is complementary: it will remove one or more boxes from the system when its *cond* triggers the event. Note that in the case of **delete** a box of the species specified in the entity list must be present:

```
when(A : |A| = 0 : inf) delete;
when(A : |A| = 0 : inf) new(2);
```

The first event will never fire, while the second one will fire as soon as there are no more boxes of species A in the system. Other examples of valid events are:

```
when(A : (|A| > 1 and |A| < 10)  : inf) new(100);
when(A :: r) delete;
when(A : (|A| = 2) and (steps = 3000) : inf) delete(2);
```

This set of event will produce oscillations of the concentrations of A, by introducing some boxes when the concentrations falls under a threshold and deleting them with a *decay* of rate r, until the simulation reaches 3000 steps; after that, all As are deleted from the system and no further evolution is possible.

The **update** verb is used to modify the value of a variable in the system. When the event is fired, the function *funcId* and the resulting value is assigned to the variable *varId*. Functions and variables are explained in greater detail in Sec. 4.1; here it is sufficient to know that variables are global *Id*s bound to real values, and that functions are mathematical expressions on variables and cardinality of entities that evaluate to a real value.

The condition of an **update** event has no entities in its *entityList*, and no rate or rate function in its *rate* part: the event is triggered as soon as its *EvExpr* evaluates to *true*. Jointly to an **update** event it is possible to use a particular kind of condition, based on the traversal of successive states.

$$
\begin{array}{lll}
\mathit{statOpList} & : & \\
 & & \mathit{stateOp} \\
 & | & \mathit{stateOp}, \mathit{stateOpList} \\
 & & \\
\mathit{statOp} & : & \\
 & & \mathit{Id} \leftarrow \mathit{Real} \\
 & | & \mathit{Id} \rightarrow \mathit{Real}
\end{array}
$$

The list of states to be traversed are expressed in a *stateOpList*; each *stateOp* element in the list expresses a condition on the quantity of an *Id* (i.e. cardinality of boxes for *boxId* or the *value* bound to a variable for *varId*).

StateOps are examined in sequence, one after the other. We say that a *stateOp* becomes *valid* when the condition on its *Id* is met for the first time. The '$\rightarrow$' operator recognizes when the quantity bound to *Id* becomes greater than the specified real value, while the '$\leftarrow$' operator recognizes when the quantity bound to *Id* becomes smaller than the specified real value.

When a *stateOp* becomes valid, the *EvExpr* passes to the evaluation of the following *stateOp* of the list. As soon as the last state in the *stateOpList* becomes valid, the *EvExpr* evaluates to *true*, so the event (update, in this case) can be fired. Once fired, the *EvExpr* restart its evaluation from the beginning of the *stateOpList*, waiting for the first *stateOp* to become valid again.

For instance, to recognize the oscillatory behaviour in Fig. 10, we can use the following piece of BlenX code:

```
let n : var = 1;
let f : function = n + 1;
...
when (: A -> 20, A <- 20 :) update (n, f);
```

This code updates the variable n, also depicted in the figure, by incrementing it at every oscillation.
The concatenation of an arbitrary succession of states allows to overcome possible limitations that are often encountered when dealing with a stochastic approach, mainly noise. As an example, look at Fig. 11: the simple state list just introduced is not enough to capture the correct period of oscillations, as highlighted in the upper-right corner of the figure.

It is easy to solve this issues adding more states to the *stateOpList*:

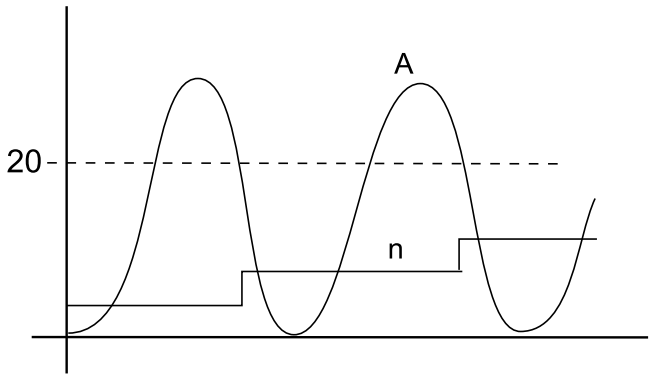


Figure 10: The species A exhibits an oscillating behaviour, captured by a state-list condition. n is a variable that "counts" the number of oscillations.

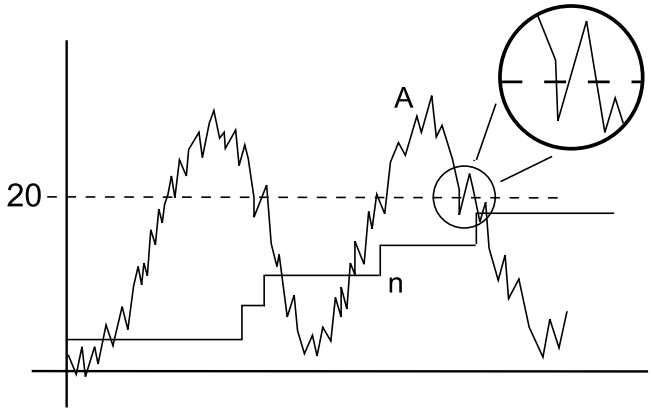


Figure 11: The species A exhibits an oscillating behaviour, but data has some noise: the state-list condition cannot capture it and n is updated in a wrong way.

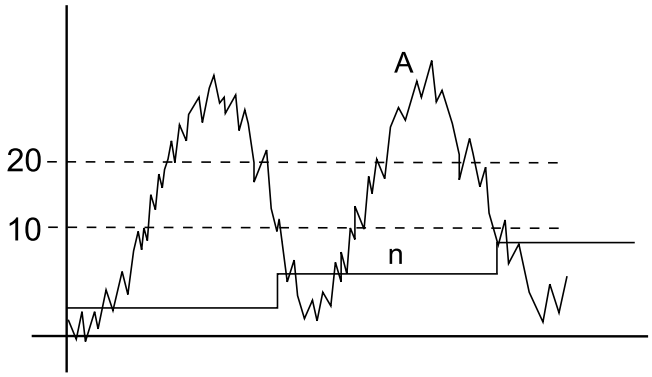


Figure 12: The new state-list condition is able to capture the oscillations correctly.

```
let n : var = 1;
let f : function = n + 1;
...
when (:A -> 10, A -> 20, A <- 20, A <- 10:) update (n, f);
```

This event can capture correctly the behaviour of the noisy oscillating system, as depicted in Fig. 12.

4.8 Prefixes

Prefixes are generated by the following BNF grammar:

$$
\begin{array}{lll}
dec & ::= & \\
 & & \ldots \\
 & | & \textbf{let } Id : \textbf{prefix} = actSeq\ ; \\
 & & \\
actSeq & ::= & \\
 & & action \\
 & | & action\ .\ prefix
\end{array}
$$

In other words, a *prefix* is an object bound to a sequence of actions. Prefixes are used exclusively in templates (see Sec. 4.9). Templates can contain variable parts; among these parts, it is possible to specify a variable *prefix* that can be substituted with a custom sequence of actions when instantiated. An example of the usage of prefixes for easing template definitions is given in the next Section.

4.9 Templates

Templates, often referred to as *generics* or *parametric processes*, are a feature of many programming languages that allows code in an extended grammar in which code can contain variable parts that are then instantiated later by the compiler with respect to the base grammar.

In BlenX template code is *specialized* and *instantiated* at compile time using binder identifiers, code or names that are passed as template arguments. Therefore, BlenX provides a grammar for defining templates and code to instantiate and use them.

4.9.1 Template declaration.

It is possible to define templates for processes, boxes and sequences. The BNF for template declaration and definition is the following:

$$
\begin{array}{lll}
dec & ::= & \\
 & & \ldots \\
 & | & \textbf{template } Id : \textbf{pproc } \langle\langle\ formList\ \rangle\rangle = piProcess\ ; \\
 & | & \textbf{template } Id : \textbf{bproc } \langle\langle\ formList\ \rangle\rangle = betaProcess\ ; \\
form & ::= & \\
 & & \textbf{name } Id \\
 & | & \textbf{pproc } Id \\
 & | & \textbf{binder } Id \\
 & | & \textbf{prefix } Id \\
 & & \\
formList & ::= & \\
 & & form \\
 & | & form,\ formList
\end{array}
$$

The declaration of a template **bproc** or **pproc** follows closely the declaration of their standard counterparts, with the **let** keyword substituted by **template**, and an additional list of template formal parameters enclosed by double angular parenthesis.

The template parameter *formList* is a comma-separated list of *form*s; each *form* declares a template argument made up of a keyword anong name, pproc, binder, prefix followed by an Id. The Id will be added to the environment of the object being defined, acting as a placeholder for the object that will be used during parameter instantiation. For example, in the following code:

```
template P : pproc<<pproc P1, name N1, name N2, binder T1>> =
   x?().N1!().ch(N2, T1).P1;
```

we do not have to define the pproc $P1$, nor we have to insert the binder identifier $T1$ into the type file: this piece of code will compile without errors, as the process $P1$ and the binder identifier $T1$ are inserted into P's environment as template arguments. P will be treated by the compiler as pproc with four template arguments: a process, two names and a binder identifier. Note that the notion of "name" is pretty general: it can be any name appearing into the template, being it a channel name, an action argument or a binder name.

4.9.2 Template instantiation.

A declared template (pproc or bproc) is held by the compiler in its symbol-table in order to satisfy following *invocations* or *instantiations* of that template. Template instantiation is the compile time procedure that substitute the template formal parameters with the actual parameter with which the template object will be used. For example, the following code is a possible instantiation of the previous pproc template:

```
let NilProc : pproc = nil;
let B : bbproc  = #(z, Z)
   [ P<<NilProc, y, z, Z2>> | y?().nil ];
```

The code generate by the compiler as the result of this instantiation is equivalent to the following hand-written code:

```
let NilProc : pproc = nil;
let B : bbproc  = #(z, Z)
   [ x?().y!().ch(z, Z2).NilProc | y?().nil ];
```

More precisely, a template is instantiated by using the Id of the template (pproc or bproc) and providing it with a list *invTempList* of comma-separated template invocations *invTempElem*s, whose kind has to match the kind of the template formal parameters.

$$
\begin{array}{lll}
\mathit{invTempElem} & ::= & \\
 & & \mathit{Id} \\
 & | & \mathit{Id}\ \langle\langle\ \mathit{invTempList}\ \rangle\rangle \\
 & | & (\ \mathit{Id},\ \textbf{unhidden}\ ) \\
 & | & (\ \mathit{Id},\ \textbf{hidden}\ ) \\
 & & \\
\mathit{invTempList} & ::= & \\
 & & \mathit{invTempElem} \\
 & | & \mathit{invTempElem},\ \mathit{invTempList} \\
 & & \\
\mathit{bp} & ::= & \\
 & & \ldots \\
 & | & \mathit{Decimal}\ \langle\langle\ \mathit{invTempList}\ \rangle\rangle
\end{array}
$$

Note that templates do not increase the expressive power of the language, they only make it easier to write generic and reusable code. Consider the following code:

```
template rep : pproc<<name x, pproc P>> = !x?().(P.nil);

template detach : pproc<<name x, prefix P, binder T, name y>> =
   x?().P.ch(x, UN).hide(x).ch(x, T).unhide(x).y!().nil;
```

The first template is the general pattern of a replicating process, that performs some actions and then gets back to its original state. The second template is the general pattern of an entity that waits for a signal on a binder, responds by performing some action and then forces an unbind.

Enzymes that catalyse a reaction with a substrate and then detach from it can then be written as follows:

```
let E1p : prefix = delay(rate).p!(). ... ;
let E1p : prefix =  ... ;

let E1 : bproc = #(p, TyrDomain) =
   [ rep<<y, detach<<p, E1p, TyrDomain, y>> >> ];
let E2 : bproc = #(q, XYDomain) =
   [ rep<<r, detach<<q, E2p, XYDomain, r>> >> ];
```

The programmer has only to define the prefix that codifies for the response ($E1p$ and $E2p$), without having to worry how to write code for forcing the detachment of the substrate.

5 Examples

This section reports some classical examples inspired by biology and it shows how BlenX can easily used to model them.

5.1 Enzymatic Reactions

Most of the chemical reactions that happen in living organism are very slow, even when thermodynamically favored. The common way to speed up a reaction is to add a *catalyst* to the reaction itself; in cells, enzymes play the role of catalysts.

5.1.1 Enzyme - Substrate:

a simplistic mechanism for the catalysis of a product P from a substrate S is the following:

$$E + S \xrightarrow{k} E + P$$

This very simple bimolecular reaction can be modelled using an *inter* communication between the box representing the enzyme E and the box representing the substrate S. Basically, the E box outputs a message through its binder, while the S box waits an input on its binder. When an input is received, S reacts by changing its structure or interface (the identifier of its binder, for example) and becomes a new species codifying for the product P:

```
[steps = 1000]
<<BASERATE:inf>>

let Enzyme : bproc = #(x,DE)
   [ rep x!().nil ];

let S : bproc = #(y,DS)
   [ y?().ch(y, DP).nil ];

run 1 Enzyme || 100 S
```

Complementary shape of molecules domains are responsible for enzymes selectivity. In our model, domains are represented as binders and their specificity is represented by the affinity between the binder identifiers DE and DS. Hence, the affinity drives the ability of the *Enzyme* to interact with the *Substrate*. We want the enzyme E to catalyse the product P with rate k, so the identifier of the binder DE on E is set to have an affinity k with the identifier DS of the binder on S:

```
{ DE, DS, DP }
%%
{ DE, DS, k }
```

where k is a Real value or inf.

5.1.2 Michaelis-Menten:

the mechanism just introduced is too simplistic and do not approximate well the dynamics of enzymatic reactions. Realistically, the substrate must somehow bind to the enzyme before the enzyme can do its work:

$$E + S \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} ES \xrightarrow{k_2} E + P$$

where ES is an enzyme-substrate complex. This behaviour is captured by the *Michaelis-Menten kinetics*, one of the most important chemical reaction mechanisms in biochemistry used to describe the catalysis of biological chemical

reactions. The most convenient derivation of the Michaelis-Menten equation is based on the *quasi steady state* approximation, where it is assumed that the concentration of the substrate-bound enzyme (and hence also the unbound enzyme) change much more slowly than those of the product and substrate.

Due to this assumption, it is possible to express the relationship between the substrate concentration and the bound and unbound enzyme concentrations in terms of the various rate constants:

$$v = \frac{v_{max}[S]}{K_m + [S]}$$

where the Michaelis constant K_m is defined as $\frac{k_{-1}+k_2}{k_1}$.

As introduced in Sec. 4.1, BlenX allows programmers to define the affinity between binder identifiers using a triple of values or a function. The previous example can be easily changed to use Michaelis-Menten kinetics instead of the standard mass-action law, by changing the type file:

```
{ DE, DS, DP }
%%
{ DE, DS, f1 }
```

where $f1$ is defined in the declaration file as:

```
let VMax : const = 100;
let Km : const = 1.0;
let f1: function = VMax * (|S| / (Km + |S|));
```

5.1.3 Michaelis-Menten with inhibitor:

enzyme inhibitors are molecules that bind to an enzyme, blocking or decreasing enzymatic activity. Since this way of regulating the enzymatic activity is easy to obtain and can correct a metabolic imbalance, many drugs are enzyme inhibitors.

There are several possibilities for an inhibitor I to interfere with enzymatic reactions: the binding of an inhibitor can stop a substrate from entering the enzyme's active site, or alternatively hinder the enzyme from catalysing its reaction:

$$E + I \underset{k'_{-1}}{\overset{k'_1}{\rightleftharpoons}} EI$$

$$ES + I \underset{k''_{-1}}{\overset{k''_1}{\rightleftharpoons}} ES + I$$

In the second case, the inhibitor binds to the enzyme-substrate complex and alters the action of the enzyme on the substrate. The derivation of the Michaelis-Menten equation is the same as for the uninhibited mechanism except for an additional term in the expression for the total enzyme concentration and a new transient, EI. The derived equation is:

$$v = \frac{v_{max}[S]}{K_m + [S] + K_m \frac{k'_{-1}}{k'_1}[I]}$$

Note how even in this case the concentration of intermediate complexes EI and ES, along with the concentration of product P, are not present in the final equation. It is straightforward to modify our BlenX model to introduce the inhibitor, by changing the $f1$ function.

```
let ki1 : const = ... ;
let ki2 : const = ... ;
let f1 : function = (VMax * |S|)/(Km + |S| + Km * (ki1/ki2) * Ci);
```

where Ci is the constant concentration of the inhibitor I and $ki1$ and $ki2$ are the constants of dissociation and association of the enzyme E with the inhibitor I. The simulated system exhibits the dynamic behaviour of Fig. 13(a).

5.1.4 Enzyme with inhibitor - detailed model:

consider again the bio-chemical representation of an enzymatic reaction, adding a little more detail:

$$E + S \rightleftharpoons_{K_{ES}^{-1}}^{K_{ES}} ES \rightharpoonup^{K_{EP}} EP \rightharpoonup^{K_P} E + P$$

We consider every intermediate complex and conformation in this model. As before, we define boxes for the enzyme and the substrate:

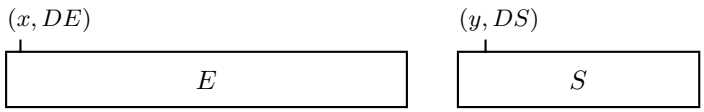
```
[steps = 1000]
<< BASERATE:inf >>

let E : bproc = #(x,DE)[ rep x!().nil ];
let S : bproc = #(y,DS)[ y?().ch(y,P).nil ];

run 1 E || 100 S
```

and we set their interaction capabilities and affinities in the corresponding *type* file:

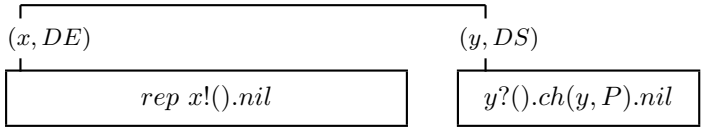
```
{ DS, DP, DE }
%%
{ (DS, DE, 1.0, 1.0, 1.0),
  (DP, DE, 0.0, 1.0, 0.0) }
```

We set the affinity between DE and DS as $(K_{ES}, K_{ES}^{-1}, K_{EP})$, as they represent respectively the rate of binding, unbinding and communication between E and S; in the same way, we define the affinity between DE and Dp as $(0, K_P, 0)$, as the enzyme E and the product P can only dissociate.

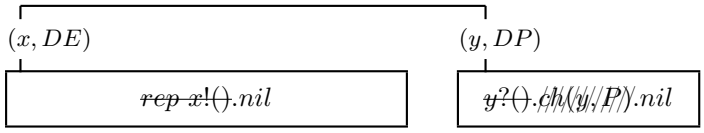
This very simple and short program is able to reproduce the desired dynamic behaviour. Let us consider an enzyme E and a substrate S in their initial configuration:



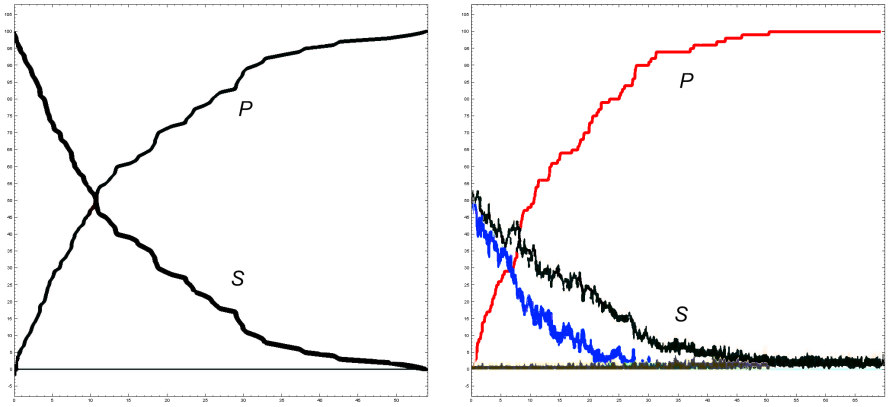
The enzyme E and the substrate S can complex together with rate K_{ES}:



and consume an *inter-communication* through the dedicated connection. After the communication, the substrate S consume immediately the action ch because its rate is infinite[3] and the pi-process of the enzyme E is replicated. The resulting system is:

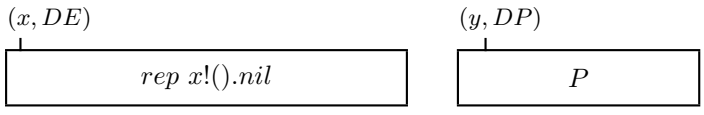


Now the two entities will decomplex with rate K_P, because of the affinity between DE and DP, by producing the two boxes:



(a) Michaelis-Menten with inhibitor (b) Detailed model

Figure 13: The observed dynamic behaviour of the Enzyme-Substrate-Inhibitor system.



The substrate S has been converted to the product P in the resulting system. This representation of the enzymatic reaction mechanism is pretty accurate, as

[3]Since no rate is associated to the change operation, we consider the BASERATE.

we do not make any assumption on the relative speed of each reaction. Furthermore, it is trivial to modify the system to introduce competitive inhibition. Assume we have a bio-chemical representation of this competitive inhibition mechanism:

$$EI + S \rightleftharpoons_{K_{EI}}^{K_{EI}^{-1}} I + E + S \rightleftharpoons_{K_{ES}^{-1}}^{K_{ES}} ES + I \multimap^{K_{EP}} EP + I \multimap^{K_P} E + P + I$$

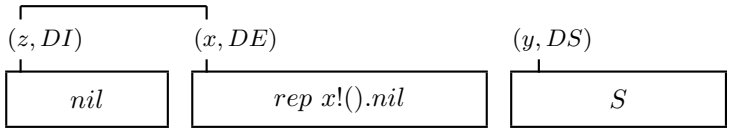
this mechanism can be obtained by adding to the previous BlenX program a box representing the inhibitor, putting it in parallel with the existing enzymes and substrates.

```
let I : bproc = #(z,DI)[ nil ];

run 10 E || 100 S || 10 I
```

We also have to update the *type* file with affinity information:

```
{ ... , DI }
%%
{ ... ,
(DE,DI,1.,1.,0.) }
```

As the affinity between DI and DE is equal to $(K_{EI}, K_{EI}^{-1}, 0)$, we have that the enzyme E can bind with the substrate S or with the inhibitor I:



Since a binder can be complexed with only another binder at a time, the resulting behaviour is exactly the one of the competitive inhibition. The dynamics of this system are reported in Fig. 13.

It is straightforward to see how it is possible to construct and modify in a compositional way complicated scenarios in which we have multi-substrate and multi-products reactions with competitive inhibition mechanisms.

5.2 Oscillatory behaviour

Many biological and ecological systems exhibit an oscillatory behaviour: the circadian rhythm, a roughly-24-hour cycle in the physiological processes of living beings; gene regulation networks; activator/inhibitor systems with feedback loops; Lotka-Volterra dynamics. Here we show with two simple examples how to codify these mechanisms in BlenX.

5.2.1 Repressilator:

the *repressilator* is a synthetic genetic regulatory network, designed specifically to exhibit a stable oscillation which is reported via the expression of a protein [7]. It acts like a clock but resembles no known natural clock. The network was implemented in Escherichia coli using standard molecular biology methods, and

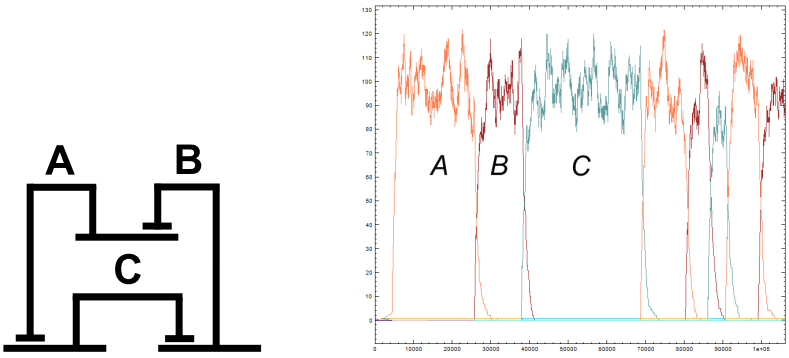


Figure 14: Left: the structure of the *Repressilator* oscillatory network. Right: the observed time course of a stochastic simulation of the network.

observations were performed that verify that the engineered colonies do indeed exhibit the desired oscillatory behavior.

The repressilator consists of three genes connected in a feedback loop, such that each gene represses the next gene in the loop, and is repressed by the previous gene (see the left part of Fig. 14).

The repressilator can be easily codified in BlenX using a process for each of the genes, a process for each of the proteins and events for transcription of a gene and the following production of a protein. Even better, as the high level behaviour of the three proteins and of the three genes is the same, it is possible to create a template for them

```
[steps = 20000]
<< BASERATE : inf >>

// Process definitions
let geneProc : pproc =
delay(0.1).nil +                //transcribe
signal?().delay(0.0001).rec!(); //a protein attaches

let proteinProc : pproc =
die(0.001) +            //decay (be degraded)
signal!().rec!() ;      //attach to a gene

// Template for a recurring pproc
template repp : pproc <<pproc P>> =
   ((rep rec?().P) | P);

let geneRep : pproc = repp<<geneProc>>;
let proteinRep : pproc = repp<<proteinProc>>;

//The process that represent a transcribed strand of DNA
let nilProc : pproc = (rep rec?().geneProc);

// Genes and Proteins templates
template Gene : bproc <<binder T,pproc P>> = #(signal:inf,T)
   [ P ];
template Protein : bproc <<binder T>> = #(signal:inf,T)
```

```
[ proteinRep ];
```

and to instantiate the appropriate code for the three copies:

```
// Genes definitions
let GeneA : bproc = Gene<<GA,geneRep>>;
let GeneB : bproc = Gene<<GB,geneRep>>;
let GeneC : bproc = Gene<<GC,geneRep>>;

// Proteins definitions
let ProteinA : bproc = Protein<<A>>;
let ProteinB : bproc = Protein<<B>>;
let ProteinC : bproc = Protein<<C>>;
```

As a commodity, we also define three boxes $ExpressN$. These three processes define species that are structurally congruent to intermediate states of $GeneN$ boxes, representing "ready for transcription" states. The $ExpressN$ boxes are not used in the program; they unique purpose is to capture a particular state of genes and trigger an event:

```
let ExpressA : bproc = Gene<<GA,nilProc>>;
let ExpressB : bproc = Gene<<GB,nilProc>>;
let ExpressC : bproc = Gene<<GC,nilProc>>;
```

Events can now be easily defined:

```
// Genes expressions definitions
when ( ExpressA :: inf ) split ( GeneA , ProteinA ) ;
when ( ExpressB :: inf ) split ( GeneB , ProteinB ) ;
when ( ExpressC :: inf ) split ( GeneC , ProteinC ) ;
```

The *prog* file is completed with the set-up of the initial conditions:

```
// Init
run 1 GeneA || 1 GeneB || 1 GeneC
```

The behaviour of the simulated system is the cyclical behaviour depicted on the right side of Fig. 14.

5.2.2 Cell-cycle:

The cell cycle is a complex network of biochemical phenomena that controls the duplication of the cell. The cycle is usually subdivided into four phases ($G1$, S, $G2$, M). The transition between them is driven by cyclin-dependent protein kinases (Cdks) that, when bound to a cyclin partner, are able to make cells to progress along their cycle. A simple model of this mechanism can be obtained just by studing the hysteresis loop that derives from the fundamental antagonistic relationship between the APC (Anaphase Promotig Complex) and cyclin/Cdk dimers: APC (with two auxiliary proteins Cdc20 and Cdh1) extinguishes Cdk activity by destroying its cyclin partners, whereas cyclin/Cdk dimers inhibit APC activity by phosporylating Cdh1. This antagonism creates two stable steady states: a $G1$ state with low cyclin/Cdk activity and an high Cdh1/APC activity, and a S-$G2$-M state with the opposite configuration.

The following code represent a simplified model of this biochemical system:

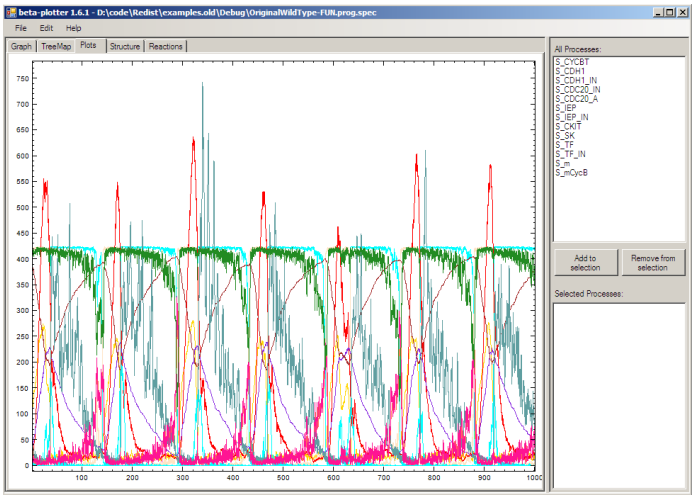


Figure 15: The observed dynamic behaviour of the Cell-Cycle system.

```
[steps = 2000, delta = 0.2]

let X : bproc = #(x:0,X)[ nil ];
when(X :: X_synthesis) new(1);
when(X :: X_self_degradation ) delete(1);
when(X :: X_degraded_by_Y ) delete(1);

let Y : bproc = #(y:0,Y)[ nil ];
let Y_IN : bproc = #(y_in:0,Y_IN) [ nil ];
when( Y_IN :: Y_self_activation ) split(Nil, Y);
when( Y_IN :: Y_activation_with_A ) split(Nil, Y);
when( Y :: Y_deactivation ) split(Nil, Y_IN);

let A : bproc = #(a:0,A)[ nil ];
when(A :: A_synthesis ) new(1);
when(A :: A_collaborate_M_X ) new(1);
when(A :: A_self_degradation ) delete(1);

when ( : mCycB -> 0.2, mCycB <- 0.1 : ) update (m, mass_div);

run 4 X || 424 Y || 424 A
```

In this code, X and Y are representing the cyclin/Cdk dimer and the active Cdh1/APC complex respectively, and A is an activator (Cdc14) that is activated indirectly by a complex pathway that involves the activation of Cdc20.

Events are used together with functions to obtain a high-level model that is the straightforward translation of the ODE model found in biological textbooks, for example in Chapter 10 of [9]. **Split** events with rate functions are used to model Michaelis-Menten reaction kinetics, while synthesis of new compounds is modelled using **new** events and degradation using **delete** events.

The *func* file holds the constant definitions:

```
let mu :  const = 0.005 ;
```

```
let k1 : const = 0.04;
let k2p : const  = 0.04;
let k2s : const = 1;
let J3 : const = 0.04;
let k3p : const = 1;
let k3s : const = 10;
let k4 : const = 35;
let J4 : const = 0.04;
let k5p : const = 0.005;
let k5s : const = 0.2;
let J5 : const = 0.3;
let k6 : const = 0.1;
let mstar : const = 10;
let alpha : const = 0.00236012;
let n : const = 4;
```

and also the definition of functions used by the events in the main *prog* file:

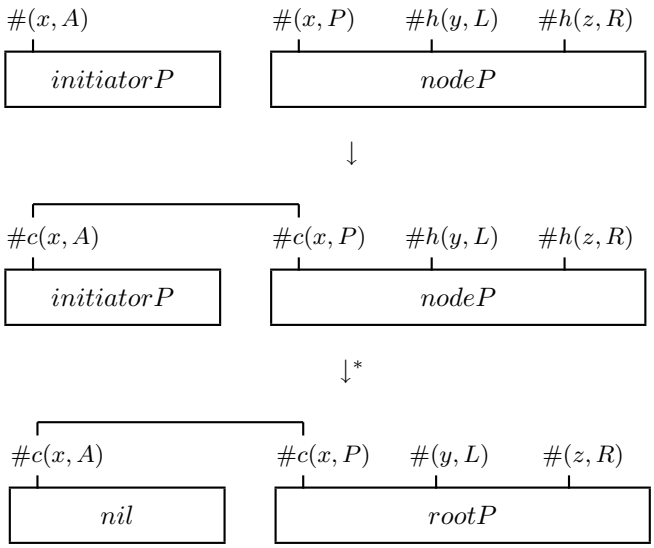
```
let m(0.1): var = mu * m * (1 - m/mstar) init 0.45;
let mass_div : function = m / 2;
let mCycB : var = m * |X| * alpha;

let X_synthesis: function = k1 / alpha ;
let X_self_degradation : function = k2p * |X|;
let X_degraded_by_Y : function = k2s * alpha * |X| * |Y|;

let Y_self_activation : function =
   (k3p * |Y_IN|) / (J3 + alpha * |Y_IN|) ;
let Y_activation_with_A : function =
   (k3s * alpha * |A| * |Y_IN|) / (J3 + alpha * |Y_IN|);
let Y_deactivation : function =
   (k4 * m * alpha * |X| * |Y|) / (J4 + alpha * |Y|);

let A_synthesis : function = k5p / alpha ;
let A_collaborate_M_X : function =
   (k5s / alpha) / (pow( (J5 / (m * alpha * |X|) ) , 4) + 1);
let A_self_degradation : function = k6 * |A|;
```

The mass is considered in our model as a time-dependent variable, which is involved in the calculations of some of the rate functions and which is driven by a specific ODE. Mass halving due to cell division is controlled by a condition on the concentration of a specific variable (e.g. mCycB, somehow related to the concentration of X); an *update* event controls the value of the mass, adjusting it whenever the concentration of mCycB cross above a threshold level and then drops below another threshold. This mechanism is described in detail in Sec. 4.7.

5.3 Self-assembly

Self-assembly is a process in which a disordered system of components forms an organized structure as a consequence of specific, local interactions among the components themselves, without an external coordination. In this example we consider a population of boxes that through complexation and decomplexation primitives and a communicating protocol organize themselves to form *binary balanced trees*. We consider a tree structure to be balanced if all the leaves

in the tree are at the same depth w.r.t the root node. An initial system is a composition of boxes called *Initiators* and boxes called *Nodes*. *Initiators* start the construction of trees; they complex to *Nodes* which, after an exchange of signals at infinite rates, become *Roots* of different trees.



When activated, a *Root* can bind with other *Nodes* on the previously hidden binders $\#(y, L)$ and $\#(z, R)$. Now, all the *Nodes* that perform a complexation with the *Root* are activated as *Child* boxes. The internal behaviour of a *Root* is defined by the process

```
let rootP : pproc = rep y?().z?().y!(node).z!(node).nil;
```

meaning that recursively a *Root* waits for signals from all his children and then propagates to them a signal with object *node*. The internal behaviour of a *Child* is defined by the process

```
let childP : pproc = rep y?().z?().x!().x?(m).y!(m).z!(m).nil;
```

meaning that recursively a *Child* waits for signals from all his children, then sends a signal to his parent, waits for a signal from his parent and finally propagates that signal to his child. The local behaviours of *Roots* and *Child* generates, in combination with the ability of boxes to bind together, a *global* behaviour which results in the creation of binary balanced trees. In general, starting with a population of *Initiators* and *Nodes* a simulation generates populations of trees as those reported in Figure 16. Notice that given a tree of depth level n, the depth level $n - 1$ is always complete. Moreover, notice that when the missing *Node* binds to the tree, then signals from all the *Child* nodes are propagated recursively to the *Root* which propagates the acknowledgment with subject *node* till to the leaves, which finally become *Child* and hence active. The complete code of the example is:

```
[ steps = 100 ]

<< BASERATE:inf >>
```

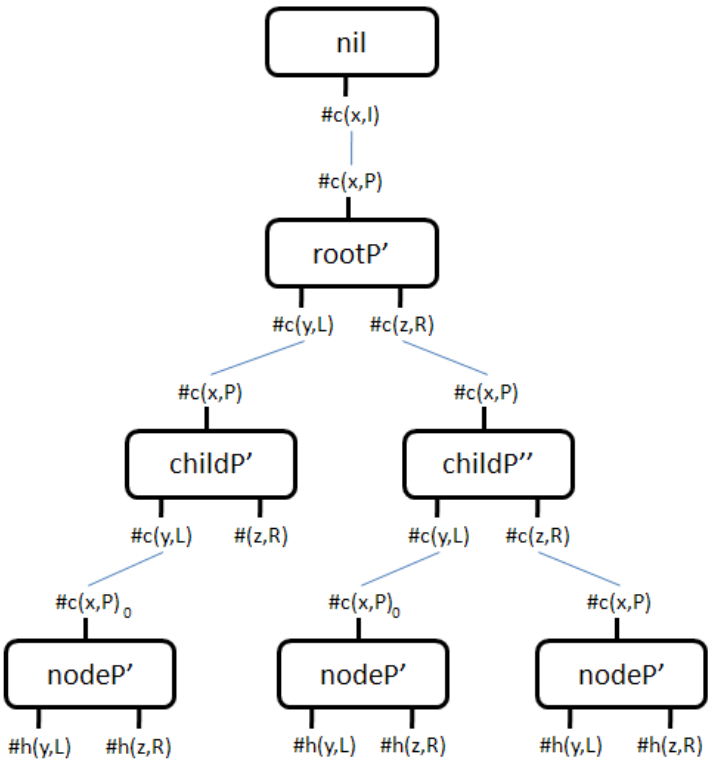


Figure 16: Example of generated tree.

```
// Initiator Definition
let I : bproc = #(x,I)
  [ x?().x!(root).nil ];

// Node Definition
let rootP : pproc =
  rep y?().z?().y!(node).z!(node).nil ;

let childP : pproc =
  rep y?().z?().x!().x?(m).y!(m).z!(m).nil ;

let nodeP : pproc =
  x!().x?(t).( t!() | (
    node?().unhide(y).unhide(z).childP +
    root?().unhide(y).unhide(z).rootP
  ) );

let Node : bproc = #(x,P),#h(y,L),#h(z,R)
  [ nodeP ];

// Init
run 2 I || 10 Node
```

where the corresponding *type* file is:

```
{P,L,R,I}
%%
{(I,P,100,0,inf),(L,P,1,0,inf),(R,P,1,0,inf)}
```

6 Conclusions

We presented the basic primitives and components of the new biology-inspired language BlenX. We then showed the usability of BlenX reporting some models of biological examples. We also briefly described the input/output supporting tools of BlenX. The BlenX environment is under further development to address relevant (biological) questions like spatial modelling and simulation as well as multi-level, multi-scale modelling of large systems.

7 Acknowledgments

We would like to thank Alida Palmisano for the insightful discussions and the Cell Cycle BlenX model as well as Roberto Larcher for the part regarding the BWB reaction generator.

References

[1] M. Bravetti and G. Zavattaro. Service oriented computing from a process algebraic perspective. *Journal of Logic and Algebraic Programming*, 70(1):3–14, 2007.

[2] L. Cardelli. Brane Calculi - Interactions of B iological Membranes. In *Proceedings of Workshop on Computational Methods in Systems Biology (CMSB'04)*, volume 3082, pages 257–278. Lecture Notes in Computer Science, Springer, 2005.

[3] V. Danos and C. Laneve. Formal molecular biology. *TCS*, 2004.

[4] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR 2004*, volume 3170 of *LNCS*, pages 292–307. Springer-Verlag, Sep 2004.

[5] P. Degano, D. Prandi, C. Priami, and P. Quaglia. Beta-binders for Biological Quantitative Experiments. In *Proc. of the 4th International Workshop on Quantitative Aspects of Programming Languages (QAPL 2006)*, volume 164 of *ENTCS*, pages 101–117. Elsevier, 2006.

[6] M. Elowitz, A. Levine, E. Siggia, and P. Swain. Stochastic gene expression in a single cell. *Science*, 297:1183–1186, 2002.

[7] Michael B. Elowitz and Stanislas Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(20):335–8, January 2000.

[8] D. Errampalli, C. Priami, and P. Quaglia. A formal language for computational systems biology. *OMICS: A Journal of Integrative Biology*, 8(4):370Ű380, 2004.

[9] C. P. Fall, E. S. Marland, J. M. Wagner, and J. J. Tyson. *Computational Cell Biology*. Springer-Verlag, 2002.

[10] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *The Journal of Computational Physics*, 22(4):403–434, 1976.

[11] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Chemical Physics*, 81:2340–2361, 1977.

[12] D.T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J. Phys. Chem.*, 22:403–434, 1976.

[13] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[14] S. Gilmore and J. Hillston. *The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling*, pages 353–368. Number 794 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[15] C.A.R. Hoare. A calculus of total correctness for communicating processes. *Science of Computer Programming*, 1(1-2):49–72, 1981.

[16] D. Hume. Probability in transcriptional regulation and its implications for leukocyte differentiation and inducible gene expression. *Blood*, 96:2323–2328, 2000.

[17] H.H. McAdams. It is a noisy business! genetic regulation at the nanomolar scale. *Trends Genet*, 15:65–69, 1999.

[18] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[19] A. Phillips and L. Cardelli. A correct abstract machine for the stochastic pi-calculus. In *Bioconcur'04*. ENTCS, August 2004.

[20] C. Priami. The stochastic π-calculus. *The Computer Journal*, (38):578–589, 1995.

[21] C. Priami and P. Quaglia. Beta binders for biological interactions. In *Proc. of Computational Methods in Systems Biology*, volume 3082 of *LNCS*, pages 20–33. Spinger, 2005.

[22] Corrado Priami and Paola Quaglia. Modeling the dynamics of bio-systems. *Briefings in Bioinformatics*, 5(3):259–269, 2004.

[23] A. Regev, E.M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, 2004.

[24] I. Ross, C. Browne, and D. Hume. Transcription of individual genes in eukaryotic cells occurs randomly and infrequently. *Immunol Cell Biol*, 1994.

[25] J. Spudich and DEJ Koshland. Non-genetic individuality: Chance in the single cell. *Nature*, 1976.

[26] Ann M. Stock, Victoria L. Robinson, and Paul N. Goudreau. Two-component signal transduction. *Annu. Rev. Biochem*, (69):183–215, 2000.