The Microsoft Research - University of Trento
Centre for Computational and Systems Biology

Technical Report CoSBi 03/2007

The Beta Workbench

Alessandro Romanel
The Microsoft Research - University of Trento
Centre for Computational and Systems Biology
DISI - Univeristà di Trento
romanel@cosbi.eu

Lorenzo Dematté
The Microsoft Research - University of Trento
Centre for Computational and Systems Biology
DISI - Univeristà di Trento
dematte@cosbi.eu

Corrado Priami
The Microsoft Research - University of Trento
Centre for Computational and Systems Biology
priami@cosbi.eu

The Beta Workbench

Alessandro Romanel, Lorenzo Dematté and Corrado Priami

The Microsoft Research - University of Trento
Centre for Computational and Systems Biology
{*romanel,dematte,priami*}*@cosbi.eu*

Abstract

This paper presents a system to model and simulate biological processes. It is based on process calculi theory and incorporates a language, a compiler, the execution environment and some graphical interface components. The language is based on β-binders, a recently introduced process algebra bio-inspired and developed to be suitable for the biological applicative domain. The runtime environment is based on a stochastic abstract machine that extends and improve the classical Gillespie's approach. The quantitative aspects included in the stochastic information associated with the language allow to simulate and plot quantitative parameters of the system under investigation. We define the syntax, semantics and implementation of the language comparing our design choices with the most common features of process calculi applied to biology. A relevant part of this work is the description of design patterns for the most common biological features in molecular interactions. This is an important aspect in exploiting the expressive power of the language and in providing a preliminary guide to the use of the compositional properties of process calculi.

Contents

1 Introduction

Biology is rapidly producing a huge number of experimental results and it is becoming impossible to coherently organize them using only human power. Abstract models to reason about biological systems is becoming an indispensable conceptual and computational tool for biologists. An abstraction has to capture the essential properties of the phenomenon under consideration, and, at the same time, it has to be computable, to allow automatic analysis, and extensible, to permit the addition of further details [1]. Computer science modeling is specifically designed to meet the above requirements, but it heavily uses mathematical symbolism that is not easy to read for a neophyte. Therefore we need an approach that hides as many technical details as possible from users.

In recent times, a paradigmatic shift occurred in biology. Researchers started trying to build system visions rather than component visions, and the focus is now rapidly moving from structure to function. This process leads to the so-called *Systems Biology* [2] that is mostly interested in the behavior of cellular processes and in the description of the interactions among components. Seen from a computer science point of view, the methods and the techniques that could be best suited to face the challenge of systems biology are those related to the description and simulation of interacting distributed systems.

The process calculi approach to the formal modeling of biological systems has gained more and more attention over the last few years, particularly since the publication on Nature of the landmark paper by Regev and Shapiro [1].

Starting from the forerunner CCS, the ‘Calculus of Communicating Systems’ [3], process calculi have been defined with the primary goal of providing formal specifications of concurrent processes, namely of computational entities executing their tasks in parallel and able to synchronize over certain kinds of activities. The model of a system is typically given as a program or a term that defines the possible behaviors of the various components of the system. Calculi are then equipped with syntax-driven rules, the so-called *operational semantics* [4]. These rules, that can automatically allow the inference of the possible future of the system under analysis. For instance, they can specify that a certain process P evolves into process Q, written $P \longrightarrow Q$.

The basic entities of process calculi are *actions* and *co-actions* (complementary actions). In the most basic view, like e.g. in CCS, an action is seen as an input or an output over a channel. Input is complementary to output and vice-versa. Actions and co-actions can also transmit/receive names over the channel (e.g. the IP address of the Internet) on which input and output are supposed to take place. This is, indeed, the underlying assumption taken in the π-calculus [5]. As it will be clear in the rest of the paper, the actual interpretation of complementarity varies from one calculus to the other. The relevant fact to be pointed out here is that complementary actions are those that parallel processes can perform together to synchronize their (otherwise) independent behaviors.

Process calculi are typically very simple, yet contain all the ingredients for the description of concurrent systems in terms of *what they can do* rather than of *what they are*.

Two main properties of process calculi are worth mentioning. First, the meaning (behavior) of a complex system is expressed in terms of the meaning of its components. A model can be designed following a bottom-up approach: one defines the basic operations that a system can perform, then the whole behavior is obtained by composition of these basic building blocks. This property is called *compositionality*. Second, the mathematical rules defining the operational semantics of process calculi allow us to implement a simulator of the runs of the system. So process calculi are specification languages that can be directly implemented and executed.

The main contribution of this paper is the definition of a process-calculi based programming language designed to model biological systems. It is hence bio-inspired in the definition of the basic primitives. Furthermore we implemented the language providing quantitative tools that allow the user to model and simulate real case studies. We define first the syntax and the operational semantics of the calculus. Then we report some templates to model most of the common biological phenomena and we describe the software architecture of or implementation. Finally, we provide some hints on how to use the software components we developed.

2 Related work

In the last few years a number of process calculi have been adapted or newly developed for applications in systems biology. Some of the most important are:

- *Biochemical stochastic π-calculus*: is a stochastic extension of the π-calculus where biochemical interactions are represented as communications of processes. Simulators for the Biochemical stochastic π-calculus [6] have been implemented, i.e. BioSpi[1] and SPiM[2] [7]. They are based on Gillespie's assumption [8] and make in-silico experiments possible;

[1]`http://www.wisdom.weizmann.ac.il/ biospi/`

[2]`http://research.microsoft.com/ aphillip/`

- *BioAmbients*: is an extension of the Biochemical stochastic π-calculus where an explicit notion of compartments is introduced [9]. The language is provided with a stochastic extension and a simulator, based on Gillespies algorithm [10], has been implemented as part of the BioSpi project;

- *Brane Calculus*: is a calculus which is centered on membranes [11]. Brane calculus is inspired by BioAmbients, but it gives membranes an active role;

- *Performance Evaluation Process Algebra (PEPA)*: is a formal language for describing Markov processes [12]. PEPA allows to quantitatively model and analyze large pathway systems. PEPA is supported by a large community and a lot of software tool for analysis and stochastic simulations are available.

- *κ-calculus*: is a formal calculus of proteins interaction [13, 14]. It was designed to represent complexation and decomplexation of proteins. The κ-calculus comes equipped with a very clear visual notation, and uses the concept of shared names to represents bonds.

For a more detailed introduction of process calculi and their application in biology, we refer the reader to [15].

3 The Language

The BetaSIM language is based on β-binders [16, 17], a process calculus developed for better representing the interactions between biological entities, and its stochastic extension. The main idea of β-binders is to encapsulate π-calculus processes into *boxes* with interaction capabilities. Like the π-calculus also β-binders is based on the notion of *naming*. Thus, we assume the existence of a countably infinite set $\mathcal{N}$ of names (ranged over by lower-case letter) and a countably infinite set of interactions capabilities $\mathcal{T}$ (ranged over by indexed δ). We also assume a special name $\tau \notin \mathcal{N} \cup \mathcal{T}$ to express internal activities of processes or delays. BetaSIM has several modifications with respect to the original syntax and all of them will be discussed throughout the paper.

A BetaSIM program, called also β-system, is a tuple $Z{=}\langle B, E, \xi \rangle$ which is a composition of a *bio-process* B, a list of *events* E and *environment* ξ. The bio-process B intuitively represents the structure of the system, that is a set of entities interacting in the same context, E represents the list of possible events enabled on the system and the environment ξ contains information like the set T of considered *types* (ranged over by Δ, Γ_0, Σ', $\cdots$).

3.1 The syntax

In this section we describe the syntax of the bio-processes, of the events and the structure of the environment. We will also discuss the pi-processes P.

The bio-process B and the list of events E are defined according to the following context-free grammar:

$$
\begin{array}{lcl}
B & ::= & \mathsf{Nil} \mid \vec{B}[P] \mid B||B \\
\vec{B} & ::= & \widehat{\beta}(x,r,\Delta) \mid \widehat{\beta}(x,r,\Delta)\vec{B} \\
\widehat{\beta} & ::= & \beta \mid \beta^h \mid \beta^c \\
P & ::= & \mathsf{nil} \mid P|P \mid\ !\pi.P \mid M \\
M & ::= & \pi.P \mid M+M \\
\pi & ::= & x(y) \mid \overline{x}\langle y\rangle \mid (\tau,r) \mid (\mathsf{die},r) \mid (\mathsf{ch}(x,\Delta),r) \mid \\
 & & (\mathsf{hide}(x),r) \mid (\mathsf{unhide}(x),r) \mid (\mathsf{expose}(x,s,\Delta),r) \\
\mathit{cond} & ::= & \vec{B}[P]:r \mid |\vec{B}[P]|=n \mid \vec{B}[P],\vec{B}[P]:r \\
\mathit{verb} & ::= & \mathsf{new}(\vec{B}[P],n) \mid \mathsf{split}(\vec{B}[P],\vec{B}[P]) \mid \mathsf{join}(\vec{B}[P]) \mid \mathsf{delete} \\
\mathit{event} & ::= & \bullet \mid (\mathit{cond})\ \mathit{verb} \\
E & ::= & \mathit{event} \mid \mathit{event} :: E
\end{array}
$$

where $x, y \in \mathcal{N}$, $n \in \mathbb{N}$ and $r \in \mathbb{R}^+ \cup \infty$ is a stochastic rate[3].

3.1.1 Bio-processes ad pi-processes

Bio-processes (or *boxes*) generated by the non terminal symbol B can be either *elementary bio-processes* (the first two productions) or a parallel composition of elementary bio-processes, i.e. bio-processes running concurrently. The special process Nil does nothing; i.e. it is the deadlocked bio-process. The bio-process $\vec{B}[P]$ is a pi-process (see below) prefixed by a specialized *beta binder* $\vec{B}$ that represents the interaction capabilities of the bio-process. The intuition is that a bio-process represent a biological entity that has its own control mechanism (the pi-process P) and some interaction capabilities expressed by the beta binder.

A beta binder $\vec{B}$ is made up of a non empty list of *elementary beta binders* of the form $\beta(x,r,\Gamma)$ (*active*), $\beta^h(x,r,\Gamma)$ (*hidden*) or $\beta^c(x,r,\Gamma)$ (*complexed*),

[3]A stochastic rate is the single parameter defining an exponential distribution that drives the stochastic behaviour of an action. The rate ∞ is used to denote immediate actions, i.e., actions that are executed as soon as they become enabled.

where the name x is the *subject* of the beta binder, r is the stochastic parameter that quantitatively drives the activities involving the binder and Γ represents the type of x. The subject x of an elementary beta binder is a binding occurence that binds all the free occurrences of x in 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 molecules interacts with/binds to other molecules. The complexed binders states that the corresponding box is physically bound through that interaction site to another box. We define three auxiliary functions to extract the set of subjects, the set of all types and the set of the complexed elementary beta binders from a beta binder.

With $\mathcal{P}$, $\mathcal{BB}$ and $\vec{\mathcal{BB}}$ we denote respectively the set of all the possible pi-processes, bio-processes and beta binder.

Definition 1. *Let $\vec{\mathcal{BB}}$ be the set of beta binders. Then, $sub : \vec{\mathcal{BB}} \to 2^{\mathcal{N}}$; $types : \vec{\mathcal{BB}} \to 2^{T}$ and $bc : \vec{\mathcal{BB}} \to 2^{T}$ are defined as follows*

$$\begin{array}{l}
sub(\beta(x,r,\Gamma)) = sub(\beta^h(x,r,\Gamma)) = sub(\beta^c(x,r,\Gamma)) = \{x\} \\
sub(\widehat{\beta}(x,r,\Gamma)\vec{B}) = \{x\} \cup sub(\vec{B}) \\
\\
types(\beta(x,r,\Gamma)) = types(\beta^h(x,r,\Gamma)) = types(\beta^c(x,r,\Gamma)) = \{\Gamma\} \\
types(\widehat{\beta}(x,r,\Gamma)\vec{B}) = \{\Gamma\} \cup types(\vec{B}) \\
\\
bc(\beta(x,r,\Gamma)) = bc(\beta^h(x,r,\Gamma)) = \emptyset \\
bc(\beta^c(x,r,\Gamma)) = \{\Gamma\} \\
bc(\widehat{\beta}(x,r,\Gamma)\vec{B}) = bc(\widehat{\beta}(x,r,\Gamma)) \cup bc(\vec{B})
\end{array}$$

We now define *well-formed* bio-processes (assuming hereafter that the operator $| - |$ denotes the cardinality of the argument, i.e. the length of a string, the length of a list or the number of elements of a set, letting the context disambiguate the overloaded symbol).

Definition 2. *Let $B = \vec{B}_1[P_1] \parallel \cdots \parallel \vec{B}_n[P_n]$ be a bio-process. We say that B is well-formed if $\forall i \in \{1, ..., n\}. \vec{B}_i$ is well-formed.*
A beta binder is well-formed if $|\vec{B}_i| = |sub(\vec{B}_i)| = |types(\vec{B}_i)| > 0$.

The condition on beta binders states that a well-formed beta binder is a non-empty string of elementary beta binders where subjects and types are all distinct.

Processes generated by the non terminal symbol P are referred as *pi-processes.* The nil process does nothing; it is a deadlocked process. The binary operator $|$ composes two processes that can run concurrently. The bang operator ! is used to replicate copies of the pi-process passed as argument. Note that we use only guarded replication, i.e. the process argument of the ! must have a prefix π that forbids any other action of the process until it has been consumed. The last non-terminal symbol M of the productions of P is used to introduce guarded choices. In fact M generates summations of guarded prefixes of the form $\pi.P$.

The actions that a pi-process can perform are described by the syntactic category π. The first two actions are common to most process calculi. They represent respectively the input/reception of something that will instantiate the placeholder y over a channel named x ($x(y)$) and the output/send of a value y over a channel named x ($\overline{x}\langle y\rangle$). The placeholder y in the input is a binding occurrence that binds all the free occurrences of y in the scope of the prefix $x(y)$. Sometimes the channel name x is called subject and the placeholder/value y is called object of the prefix. The action (τ, r) denotes internal activities not involved in interactions of the pi-process that execute the prefix. Quantitatively, it is a delay driven by the stochastic parameter r.

Parallel pi-processes that perform complementary actions on the same channel inside the same box (a process perform and input $x(z)$ and the other one an output $\overline{x}\langle y\rangle$) can synchronize and exchange a message. The value y flows from the process performing the output to the one performing the input. The flow of information affects the future behavior of the system because all the free occurrences of z bound by the input placeholder are replaced in the receiving process by the actual value sent y. Pi-processes in different boxes can perform an *inter-communication* (distinct from the *intra-communication* described above) if one send out of the box a value y over a link x that is bound to an active binder of the box $\beta(x, r, \Delta)$ and a pi-process in another box is willing to receive a value from a *compatible* binder $\beta(y, r, \Gamma)$ through the action $y(z)$. The two corresponding binders are compatible if a *compatibility* function α applied to the types returns a real in $\mathbb{R}^+ \cup \infty$. Note that intra-communications occur on perfectly symmetric input/output pairs that share the same subject, while inter-communication can occur between primitives that has different subjects provided that their types are compatible. This new notion of communication is particularly relevant in biology interactions occur on the basis of affinity which can never be exact complementary of molecular structures. The same substance can interact with many other in the same context, although with different levels of affinity.

The remain actions are peculiar of the BetaSIM language. The action (die, r) destroy the box enclosing the pi-process that executes the prefix. The action $(\mathsf{ch}(x, \Delta), r)$ change the type of the binder with subject x to Δ. The actions $(\mathsf{hide}(x), r)$ and $(\mathsf{unhide}(x), r)$ are complementary and they change the state of an active binder to hidden and vice versa. Finally, the action $(\mathsf{expose}(x, s, \Delta), r)$ creates a new binder for the current box with subject x, rate s and type Δ. The subject x of the newly created binder is a binding occurrence that binds all the free occurrence of x in the prefixed pi-process.

3.1.2 Events

The non terminal symbol E generates a list of events. A list of events is always related to a bio-process B and each single event occurs only if its condition is satisfied on a set of one or more boxes composing B. A single *event* is the composition of a condition *cond* and an action *verb* and is a riformulation of the f_{join} and f_{split} axioms of the original β-binders definition.

To gain efficiency with respect to β-binders, in the present version of the language conditions are limited to structural congruence of bio-processes and cardinality of the equivalence classes originated by the structural congruence (more detail will be available in Subsect. ??). In particular, given a bio-process B, the condition $\vec{B}[P] : r$ ($|\vec{B}[P]| = n$) is satisfied if there is at least one (exactly n) box in B that is equivalent[4] to $\vec{B}[P]$. The condition $\vec{B}[P], \vec{B}[P] : r$ is satisfied if B contains at least a bio-process equivalent to the first element of the pair and at least a bio-process equivalent to the second element of the pair. The stochastic information associated to conditions will be described in the next subsection.

The syntactic category *verb* denotes the actions that are associated with conditions. The action $\mathsf{new}(\vec{B}[P], n)$ creates n new instances of the bio-process specified as argument. Hereafter when the created bio-process coincides with the one specified in the condition we will write for short $\mathsf{new}(n)$. The action $\mathsf{split}(\vec{B}[P], \vec{B}[P])$ remove a copy of the box in the condition and introduces the two processes arguments of the split operation. The action $\mathsf{join}(\vec{B}[P])$ remove a copy of each of the bio-processes in the condition and introduce a copy of its argument. Finally, the action delete remove a copy of the bio-process in the condition.

The action new can be used to model, for example, the translation of new proteins and enzymes in the cell at a given rate, or the introduction of a new bio-process from the external environment (entrance of hormones, nutrients or other entities in the cell) without necessarily modelling the whole transport or synthesis pathway. The actions split and join can represent

[4]We are using the word equivalent here to mean structurally congruent. Since this concept is introduced later in the paper we remain vague deliberately relying on the intuition.

classical bind/unbind reactions in molecular environments. Finally $\textsf{delete}$ is useful to model the decay or degradation of entities (such as molecules and proteins).

We now define *well-formed* events.

Definition 3. *Let* (cond) verb *be an event. We say that the event is well-formed if it satisfies one of the following forms and conditions:*

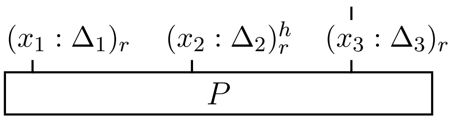
- $(\vec{B}_1[P_1] : r)$ $\textsf{split}(\vec{B}_2[P_2], \vec{B}_3[P_3])$
  with $(bc(\vec{B}_2) \cap bc(\vec{B}_3) = \emptyset)$ *and* $(bc(\vec{B}_2) \cup bc(\vec{B}_3) = bc(\vec{B}_1))$;
- $(\vec{B}_1[P_1], \vec{B}_2[P_2] : r)$ $\textsf{join}(\vec{B}_3[P_3])$
  with $((bc(\vec{B}_1) \cap bc(\vec{B}_2) = \emptyset)$ *and* $(bc(\vec{B}_1) \cup bc(\vec{B}_2) = bc(\vec{B}_3))$;
- $(\vec{B}[P] : r)$ $\textsf{new}(\vec{B}'[P'], n)$ *with* $bc(\vec{B}) = \emptyset$;
- $(|\vec{B}[P]| = m)$ $\textsf{new}(\vec{B}'[P'], n)$ *with* $bc(\vec{B}) = \emptyset$ *and* $\vec{B}'[P'] \equiv_b \vec{B}[P]$;
- $(\vec{B}[P] : r)$ $\textsf{delete}$ *with* $bc(\vec{B}) = \emptyset$.

A list E of events is well-formed if all its events are well-formed.

The intuition underlying the above definition is that manipulation of boxes must take care of complexes. In particular, it is forbidden to create new copies or to destroy boxes that are part of complexes (last three items).

3.1.3 Graphical syntax

The BetaSIM language is also provided with a graphical representation of boxes:



The pairs $x_i : \Delta_i$ represent the sites through which the box may interact with other boxes. Types Δ_i express the interaction capabilities at x_i. The value r represent the stochastic rate, h represents the hidden status and the black line over the last beta binder represents the complexed status. P is the pi-process defining the internal structure of the box. Hereafter, we assume that all the boxes around are composed in a β-system through the $||$ operator.

3.1.4 The environment

The environment ξ contains the set T of types, the affinity function $\alpha : T^2 \to \mathbb{R}^3$, the function $\rho : \mathcal{N} \to \mathbb{R}$ and a symmetric binary relation $\odot$, called complexation relation.

In the original paper of β-binders, types are defined over sets of names in $\mathcal{N}$. However, we refer to a more general definition of type. In particular, we assume that the elements composing T are defined over algebric structures with decidable equality relation.

The function α is the affinity function. Given two types Δ and Γ, the application $\alpha(\Delta,\Gamma)=(r,s,t)$ returns a measure of the compatibility of the two types. The value r represents the *complexation* stochastic rate, the value s represents the *decomplexation* stochastic rate and t represents the *inter-communication* stochastic rate. In the next sections the meant of complexation and decomplexation will be clearily explained.

We define also three auxiliary $\alpha_c(\Delta,\Gamma) = r$, $\alpha_d(\Delta,\Gamma) = s$ and $\alpha_i(\Delta,\Gamma) = t$ that project the components of the result of $\alpha(\Delta,\Gamma)$.

The function ρ associates stochastic rates to names in $\mathcal{N}$. Given $\rho(x)=r$, the value r drives the stochastic behaviour of the communications enabled on the channel x.

Definition 4. *Let $\mathcal{L} = \{\|_0, \|_1\}$ and $\vartheta \in \mathcal{L}^*$. The set of labels L (with metavariable γ) is defined as $\gamma ::= \vartheta\Delta$, where $\Delta \in T$.*

Labels ϑ are also called localities and are used to provide interaction sites of boxes with unique names. The complexation relation $\odot \subseteq L \times L$ is a symmetric binary relation defined over the set of labels Θ. Intuitively, $\odot$ states that two interaction sites are joined in a complex. We say that the relation $\odot$ is well-formed if for each pair $(\gamma,\gamma') \in \odot$ there does not exist another pair in $\odot$ that contains γ or γ'.

Definition 5. *Let $\odot$ be a complexation relation. The relation $\odot$ is well-formed if $\gamma \odot \gamma' \wedge \gamma \odot \gamma'' \Rightarrow \gamma' = \gamma''$.*

Two labels are connected if there exists a path of relations built by $\odot$ that relates the two labels, i.e., the corresponding interaction sites are part of the same complex.

Definition 6. *Let $\odot$ be a well-formed complexation relation and let $\gamma,\gamma' \in \odot$. Then γ and γ' are connected, denoted with $\gamma\overline{\odot}\gamma'$, if there exists labels $\gamma_1 = \vartheta_1\Delta_1, \cdots, \gamma_n = \vartheta_n\Delta_n$ such that*

$$(\gamma = \gamma_1) \wedge (\gamma' = \gamma_n) \wedge (\forall i \in \{1,...,n/2\}\gamma_{2i-1} \odot \gamma_{2i}) \wedge$$
$$(\forall i \in \{1,...,(n/2)-1\}(\vartheta_{2i} = \vartheta_{2i+1} \wedge \Delta_{2i} \neq \Delta_{2i+1}))$$

Since $\odot$ is well-formed, then we are sure that the value n is even. We also introduce a notion of substitution of environment elements. In particular, given an environment $\xi = (T,\alpha,\rho,\odot)$ and element T', α', ρ' and $\odot'$, a substitution is defined in the following way:

$$\xi[T'] = (T', \alpha, \rho, \odot) \qquad \xi[\alpha'] = (T, \alpha', \rho, \odot)$$
$$\xi[\rho'] = (T, \alpha, \rho', \odot) \qquad \xi[\odot'] = (T, \alpha, \rho, \odot')$$

Sequential substitution are possible,

$$\xi[T'][\alpha'][\rho'][\odot'] = (T', \alpha', \rho', \odot')$$

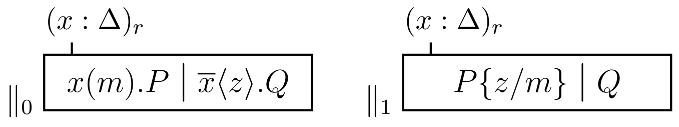
Moreover, with T_ξ, α_ξ, ρ_ξ and $\odot_\xi$ we indicate the elements of the environment ξ.

3.2 The operational semantics

The evolution of the system is formally specified through the *operational semantics* of the language, which is defined with a limited number of operations. In order to define the semantics, we first enrich the language with labels that allow us to identify in a unique way all the boxes composing the considered system.

We use the localities ϑ, previously introduced, and we use them to label boxes. We then replace each box $\vec{B}[P]$ with a labeled box $\vartheta\vec{B}[P]$ (where ϑ provides a linear encoding of the syntactical location of the box $\vec{B}[P]$ in the syntax tree of the whole initial system).

For instance, the bio-process $\beta^h(x, r, \Delta)[P] \parallel \beta(z, s, \Sigma)[Q]$ is mapped to the bio-process $(\parallel_0 \beta^h(x, r, \Delta)[P]) \parallel (\parallel_1 \beta(z, s, \Sigma)[Q])$. Graphically, such a parallel composition of bio-processes can be represented in the following way:



where the labels that precede the boxes are the localities associated to them. For semplicity, if parallel boxes $\vartheta B[P] \parallel \vartheta'' B'[P']$ have labels $\vartheta = \vartheta_0\vartheta_1$ $\vartheta' = \vartheta_0\vartheta_2$ that share subparts, than it is possible to represent the bio-process also with $\vartheta_0(\vartheta_1 B[P] \parallel \vartheta_2 B'[P'])$.

In the reminder of the technical report, when not necessary, we will omit the localities, either in the formal and graphical representation. Moreover, the operational semantics of the language is defined up to structural congruence. The structural congruence for the BetaSIM β-systems is defined through a structural congruence over pi-processes, a structural congruence over beta-processes and a structural congruence over events.

Definition 7. *The structural congruence over pi-processes, denoted $\equiv_p$, is the smallest relation which satisfies the laws in Fig. 1 (group a), the structural congruence over beta-processes, denoted $\equiv_b$, is the smallest relation which satisfies the laws in Fig. 1 (group b) and the structural congruence over events, denoted $\equiv_e$, is the smallest relation which satisfies the laws in Fig. 1 (group c).*
Hence, two β-systems $Z = \langle B, E, \xi \rangle$ and $Z' = \langle B', E', \xi' \rangle$ are structurally congruent, indicated with $Z \equiv Z'$, only if $B \equiv_b B'$, $E \equiv_e E'$ and $\xi = \xi'$.

Notice that the structural congruence do not consider the presence of locations associated to boxes. In general, considering also the locations, we have to guarantee that they do not change under structural congruence. Formally, assuming B and B' bio-processes, we have that:

$$\vartheta B \equiv_b \vartheta' B' \Leftrightarrow \vartheta = \vartheta' \wedge B \equiv_b B'$$

The structural congruence is computable and hence we can effectively implement it.

Theorem 1. *The relation $\equiv$ is decidable.*

Proof. We report here a sketch of the proof. The set of bio-processes $\mathcal{BB}$ match with the set $\mathcal{BB}^e$, introduced in [18]. Therefore the structural congruence $\equiv_b$ is decidable and efficently solvable. As a consequence, also the structural congruece $\equiv_e$ is efficently solvable and therefore the structural congruence $\equiv$ is decidable and efficently solvable. More details are in [18]. $\square$

The operational semantics of the language is defined using a reduction relation $\rightarrow_s$, which uses a labeled reduction relation $\xrightarrow{\theta}$.

Definition 8. *The set of labels Θ, with metavariable θ, is defined in the following way:*

$$\theta ::= r, type, data$$

where $r \in \mathbb{R}$, $type \in \{die, new\}$ and data is a generic string. The function $rate : \Theta \rightarrow \mathbb{R}$ returns the value r of the triple.

The main concept of β-binders is to encapsulate of processes into boxes with interaction capabilities. This encapsulation allows us to distinguish between three types of operations: monomolecular, bimolecular and events. Monomolecular operations describe the evolution of *single entities* and therefore we define them *intra* actions; the other two kinds of operations describe interactions that involves *two or more entities*, and so they are defined *inter* actions.

group a

- $P_1 \equiv_p P_2$ if P_1 and P_2 α-equivalent
- $P_1 \mid (P_2 \mid P_3) \equiv_p (P_1 \mid P_2) \mid P_3$
- $P_1 \mid P_2 \equiv_p P_2 \mid P_1$
- $P \mid \mathsf{nil} \equiv_p P$
- $M_1 + (M_2 + M_3) \equiv_p (M_1 + M_2) + M_3$
- $M_1 + M_2 \equiv_p M_2 + M_1$
- $!\pi.P \equiv_p \pi.(P \mid !\pi.P)$

group b

- $\vec{B}[P_1] \equiv_b \vec{B}[P_2]$ if $P_1 \equiv P_2$
- $B_1 \mid\mid (B_2 \mid\mid B_3) \equiv_b (B_1 \mid\mid B_2) \mid\mid B_3$
- $B_1 \mid\mid B_2 \equiv_b B_2 \mid\mid B_1$
- $B \mid\mid \mathsf{Nil} \equiv_b B$
- $\vec{B}_1\vec{B}_2[P] \equiv_b \vec{B}_2\vec{B}_1[P]$
- $\vec{B}^*\widehat{\beta}(x,r,\Gamma)[P] \equiv_b \vec{B}^*\widehat{\beta}(y,r,\Gamma)[P\{y/x\}]$
  with y fresh in P and $y \notin sub(\vec{B}^*)$

group c

- $(B_0 : r)\ split(B_1, B_2) \equiv_e (B'_0 : r)\ split(B'_1, B'_2)$
  if $B_0 \equiv_b B'_0$, $B_1 \equiv_b B'_1$ and $B_2 \equiv_b B'_2$
- $(B : r)\ delete \equiv_e (B' : r)\ delete$, if $B \equiv_b B'$
- $(B : r)\ new(B_1, n) \equiv_e (B' : r)\ new(B'_1, n)$
  if $B \equiv_b B'$ and $B_1 \equiv_b B'_1$
- $(|B| = m)\ new(B_1, n) \equiv_e (|B'| = m)\ new(B'_1, n)$
  if $B \equiv_b B'$ and $B_1 \equiv_b B'_1$
- $(B_0, B_1 : r)\ join(B_2) \equiv_e (B'_0, B'_1 : r)\ join(B'_2)$
  if $B_0 \equiv_b B'_0$, $B_1 \equiv_b B'_1$ and $B_2 \equiv_b B'_2$
- $E_0{::}E_1 \equiv_e E_1{::}E_0$

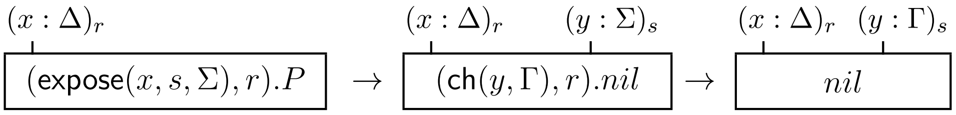
Figure 1: Structural laws for BetaSIM language.

3.2.1 Monomolecular operations

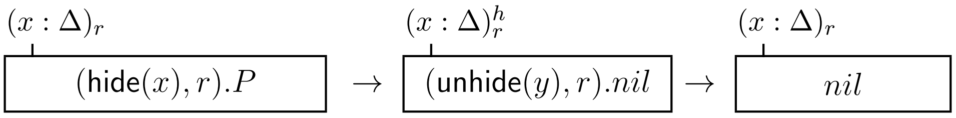
The formal semanctics of monomoecular operations is reported in Table 1. Hereafter substitutions are typed as $\{-/-\} : \mathcal{N} \to \mathcal{N}$. In general, *intra-boxes communication* allows components within the same box to interact,

$$\overset{(x:\Delta)_r}{\boxed{x(m).P \mid \overline{x}\langle z\rangle.Q}} \rightarrow \overset{(x:\Delta)_r}{\boxed{P\{z/m\} \mid Q}}$$

The *expose* action adds a new site of interaction to the interface, the *change* action modifies the type of an interaction site,



the *hide* and *unhide* actions makes respectively invisible and visible an interaction site.



and the *die* action eliminates the box that performs the action and, by propagating the proper information with the label $\theta = r, die, \vartheta$ through the derivation tree, cause the elimination of all the boxes directly or indirectly complexed with them. In the Sect. 5, this mechanism will be clearily explained. Notice that the environment is modified to delete all the bindings in which the eliminated box is involved.

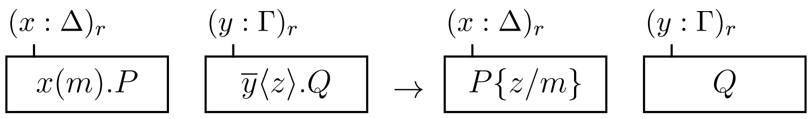
(intra)	$\langle \vartheta\vec{B}[\overline{x}\langle z\rangle.\, P_1 + M_1 \mid x(w).\, P_2 + M_2 \mid P_3], E, \xi\rangle \xrightarrow{\rho(x),\bullet,\epsilon} \langle \vartheta\vec{B}[P_1 \mid P_2\{z/w\} \mid P_3], E, \xi\rangle$
(tau)	$\langle \vartheta\vec{B}[(\tau, r).\, P_1 + M_1 \mid P_2], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta\vec{B}[P_1 \mid P_2], E, \xi\rangle$
(expose)	$\langle \vartheta\vec{B}[(\mathsf{expose}(x,s,\Gamma), r).\, R + M \mid Q], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta\vec{B}\,\beta(y,s,\Gamma)[R\{y/x\} \mid Q], E, \xi\rangle$ $y \notin sub(\vec{B})$ *and* $\Gamma \notin types(\vec{B})$
(change)	$\langle \vartheta\vec{B}^*\,\beta(x,s,\Delta)[(\mathsf{ch}(x,\Gamma), r).\, R + M \mid Q], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta\vec{B}^*\,\beta(x,s,\Gamma)[R \mid Q], E, \xi\rangle$
(hide)	$\langle \vartheta\vec{B}^*\,\beta(x,s,\Delta)[(\mathsf{hide}(x), r).\, R + M \mid Q], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta\vec{B}^*\,\beta(x,s,\Delta)[R \mid Q], E, \xi\rangle$
(unhide)	$\langle \vartheta\vec{B}^*\,\beta^h(x,s,\Delta)[(\mathsf{unhide}(x), r).\, R + M \mid Q], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta\vec{B}^*\,\beta(x,s,\Delta)[R \mid Q], E, \xi\rangle$
(die)	$\langle \vartheta\vec{B}[(die, r).\, R + M \mid Q], E, \xi\rangle \xrightarrow{r,die,\vartheta} \langle \mathsf{Nil}, E, \xi[\odot']\rangle$ where $\odot' = \odot_\xi \setminus \{(\vartheta_0\Delta, \vartheta_1\Gamma) : \vartheta_0 = \vartheta \vee \vartheta_1 = \vartheta\}$

Table 1: Monomolecular reduction rules

3.2.2 Bimolecular operations

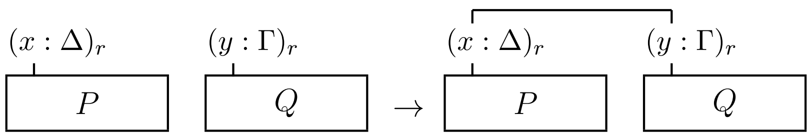
Bimolecular operations describe interactions that involves two boxes. The formal semantics of these operations is reported in Table 2.

Inter-communication represent the classical notion of communication between boxes:

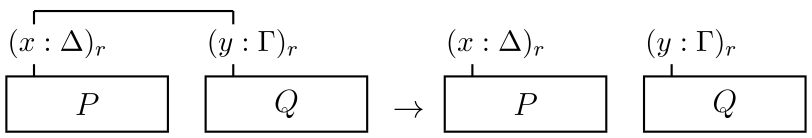


In particular, the communication is enabled only if the affinity of the types of the involved elementary beta binders $\alpha(\Delta,\Gamma)$ is a triple $(0,0,n)$ with $n>0$. This means that the complexation and decomplexation feature is not enabled and hence only a notion of communication is permitted. This also allows to maintain a compatibility with respect to the original version of β-binders, where complexation and decomplexation are not available.

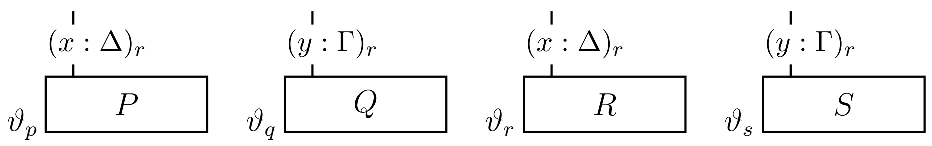
Complex and *decomplex* operations create and delete dedicated communication binding between boxes. The biological counterpart of this construct is the binding of a ligand to a receptor, or of an enzyme to a substrate through an active domain. Assume $\alpha(\Delta,\Gamma)=(r,s,t)$ with $r,s,t>0$. The *complex* operation creates, with rate $\alpha_c(\Delta,\Gamma)$, a dedicated communication binding:



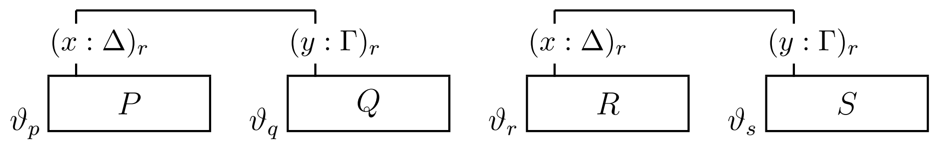
while the *decomplex* operation deletes, with rate $\alpha_d(\Delta,\Gamma)$, an already existing binding:

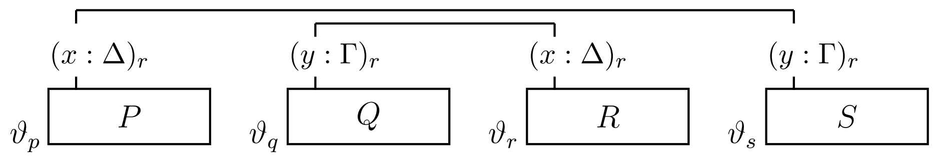


Notice that the information of the bindings is not present in the formal description of the boxes. What we know by the bio-process is that the elementary beta binders are in the complex status. For example, considering the graphical representation of the following bio-process:



with $\vartheta_p\neq\vartheta_q\neq\vartheta_r\neq\vartheta_s$, we do not know which box is complexed with which other box. The information about the complexation is maintained by the symmetric binary relation $\odot_\xi$ of the environment ξ. Several configurations are in fact possible:





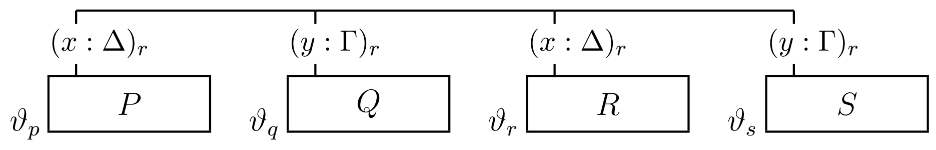
For the first example it is

$$\odot_\xi = \{(\vartheta_p\Delta, \vartheta_q\Gamma), (\vartheta_r\Delta, \vartheta_s\Gamma), (\vartheta_q\Gamma, \vartheta_p\Delta), (\vartheta_s\Gamma, \vartheta_r\Delta)\}$$

while for the second example it is

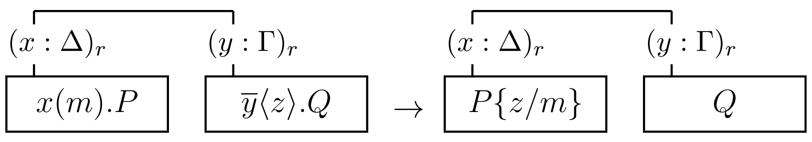
$$\odot_\xi = \{(\vartheta_p\Delta, \vartheta_s\Gamma), (\vartheta_q\Gamma, \vartheta_r\Delta), (\vartheta_s\Gamma, \vartheta_p\Delta), (\vartheta_r\Delta, \vartheta_q\Gamma)\}$$

The well-formedness of the relation $\odot_\xi$ preserves by situations in which elementary beta binders are complexed with more than one other elementary beta binders. For example, in the following configuration:



$\odot_\xi = \{(\vartheta_p\Delta, \vartheta_q\Gamma), (\vartheta_r\Delta, \vartheta_s\Gamma), (\vartheta_q\Gamma, \vartheta_p\Delta), (\vartheta_s\Gamma, \vartheta_r\Delta), (\vartheta_q\Gamma, \vartheta_r\Delta),$
$(\vartheta_r\Delta, \vartheta_q\Gamma)\}$ and the well-formedness condition is not preserved because there exists pairs $(\vartheta_q\Gamma, \vartheta_p\Delta)$ and $(\vartheta_q\Gamma, \vartheta_r\Delta)$ such that $\vartheta_r\Delta \neq \vartheta_p\Delta$.

Finally, the *inter-complex* operation enables, with rate $\alpha_i(\Delta, \Gamma)$, a communication between complexed boxes through the complexed elementary beta-binders:



A more intuitive discussion of *complex*, *decomplex* and *inter-complex* operations is reported in Sect. 5.

$$\text{(inter\_c)}\quad \frac{P_1 \equiv \overline{x}\langle z\rangle.\, R_1 + M_1 \mid Q_1 \qquad P_2 \equiv y(w).\, R_2 + M_2 \mid Q_2}{\langle \vartheta_1 \vec{B}_1[P_1] \parallel \vartheta_2 \vec{B}_2[P_2], E, \xi\rangle \xrightarrow{\alpha_i(\Gamma,\Delta),\bullet,\epsilon} \langle \vartheta_1 \vec{B}_1[R_1 \mid Q_1] \parallel \vartheta_2 \vec{B}_2[R_2\{z/w\} \mid Q_2], E, \xi\rangle}$$
$$\text{where } \vec{B}_1 = \beta^c(x,r,\Gamma)\,\vec{B}_1^* \text{ and } \vec{B}_2 = \beta^c(y,s,\Gamma)\,\vec{B}_2^*$$

$$\text{(inter)}\quad \frac{P_1 \equiv \overline{x}\langle z\rangle.\, R_1 + M_1 \mid Q_1 \qquad P_2 \equiv y(w).\, R_2 + M_2 \mid Q_2}{\langle \vartheta_1 \vec{B}_1[P_1] \parallel \vartheta_2 \vec{B}_2[P_2], E, \xi\rangle \xrightarrow{\alpha_i(\Gamma,\Delta),\bullet,\epsilon} \langle \vartheta_1 \vec{B}_1[R_1 \mid Q_1] \parallel \vartheta_2 \vec{B}_2[R_2\{z/w\} \mid Q_2], E, \xi\rangle}$$
$$\text{provided } \alpha_c(\Gamma,\Delta) = 0 \text{ and where } \vec{B}_1 = \beta(x,r,\Gamma)\,\vec{B}_1^* \text{ and } \vec{B}_2 = \beta(y,s,\Gamma)\,\vec{B}_2^*$$

$$\text{(comp)}\quad \langle \vartheta_1\, \beta(x,r,\Delta)\,\vec{B}_1^*[P_1] \parallel \vartheta_2\, \beta(y,s,\Gamma)\,\vec{B}_2^*[P_2], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta_1 \vec{B}_1[P_1] \parallel \vartheta_2 \vec{B}_2[P_2], E, \xi[\odot]\rangle$$
$$\text{where } \vec{B}_1 = \beta^c(x,r,\Delta)\,\vec{B}_1^*,\ \vec{B}_2 = \beta^c(y,s,\Gamma)\,\vec{B}_2^*\ ,\ r = \alpha_c(\Gamma,\Delta)$$
$$\text{and } \odot = \odot_\xi \cup \{(\vartheta_1\Delta, \vartheta_2\Gamma), (\vartheta_2\Gamma, \vartheta_1\Delta)\}$$

$$\text{(dcomp)}\quad \langle \vartheta_1\, \beta^c(x,r,\Delta)\,\vec{B}_1^*[P_1] \parallel \vartheta_2\, \beta^c(y,s,\Gamma)\,\vec{B}_2^*[P_2], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta_1 \vec{B}_1[P_1] \parallel \vartheta_2 \vec{B}_2[P_2], E, \xi[\odot]\rangle$$
$$\text{where } \vec{B}_1 = \beta(x,r,\Delta)\,\vec{B}_1^*,\ \vec{B}_2 = \beta(y,s,\Gamma)\,\vec{B}_2^*\ ,\ r = \alpha_d(\Gamma,\Delta)$$
$$\text{and } \odot = \odot_\xi \setminus \{(\vartheta_1\Delta, \vartheta_2\Gamma), (\vartheta_2\Gamma, \vartheta_1\Delta)\}$$

Table 2: Bimolecular reduction rules

3.2.3 Events

Events can be considered as global rules of the environment, triggered only when the conditions associated with them are satisfied. The original beta-binders specification [16] had operations for joining boxes together and for splitting them defined as mathematical functions through λ-terms. A join operation can model the bind of two boxes to form an active complex. Then, after some internal transformations, the complex can be broken with a split operation releasing a product.

The functions f_{join} and f_{split} determine the actual interface of the bio-process resulting from the aggregation or separation of boxes, as well as possible renamings of the enclosed pi-processes (see Table 3 for their semantics).

Functions f_{join} and f_{split} are evaluated and, if the function is defined (i.e. if for that imput it does not evaluate to $\perp$) boxes are joined or splitted. The expressive power of computable functions as f_{join} and f_{split} are introduces two drawbacks: it limits performance, since all the f_{join}'s and f_{split}'s functions defined in the system have to be evaluated at each time step, and it makes static analysis of the processes difficult. Without join and split functions the beta-binders process algebra do not have a way to modify the number of bio-processes. A bio-process can change its internal pi-process to respond to state changes, and the pi-process can execute actions to change the bio-process interface (the number and type of the subject it exposes), but the bio-process

$$\begin{array}{ll}
\text{(join)} & \vec{B}_1[P_1] \parallel \vec{B}_2[P_2] \longrightarrow \vec{B}[P_1\sigma_1|P_2\sigma_2] \\
& \text{provided that } f_{join} \text{ is defined in } (\vec{B}_1, \vec{B}_2, P_1, P_2) \\
& \text{and with } f_{join}(\vec{B}_1, \vec{B}_2, P_1, P_2) = (\vec{B}, \sigma_1, \sigma_2) \\
\text{(split)} & \vec{B}[P_1|P_2] \longrightarrow \vec{B}_1[P_1\sigma_1] \parallel \vec{B}_2[P_2\sigma_2] \\
& \text{provided that } f_{split} \text{ is defined in } (\vec{B}, P_1, P_2) \\
& \text{and with } f_{split}(\vec{B}, P_1, P_2) = (\vec{B}_1, \vec{B}_2, \sigma_1, \sigma_2)
\end{array}$$

Table 3: Join and split axioms of the original formalism of β-binders.

itself cannot be incorporated in another bio-process, nor it can be divided in two distint bio-processes. Join and split axioms can be explained in natural language with a time clause: when the function f_{split} is defined in $(\vec{B}, P_1, P_2)$ (i.e. $f_{split}(\vec{B}, P_1, P_2) \neq \perp$), the operation is carried out, otherwise the system remain unchanged. So the join and split axioms can be reformulated as *events*: when some given conditions are fulfilled on a set of one or more boxes, an action is triggered.

The formal semanctics of events is reported in Table 4. Conditions are defined over structural congruence of one or two boxes.

Let hereafter $Z = \langle B, E, \xi \rangle$ be a β-system. The meaning of the event conditions is the following:

- $(\vec{B}[P] : r)$verb: The action *verb* is enabled, with rate r, only if the bio-process B of the β-system Z is structurally congruent to the bio-process $\vartheta\vec{B}[P] \parallel B'$;

- $(\vec{B}[P], \vec{B}'[P'] : r)$verb: The action *verb* is enabled, with rate r, only if the bio-process B of the β-system Z is structurally congruent to the bio-process $\vartheta\vec{B}[P] \parallel \vartheta'\vec{B}'[P'] \parallel B'$;

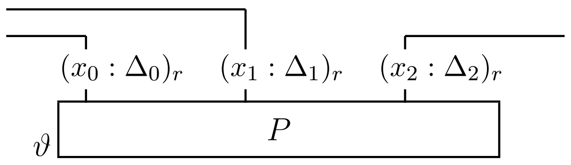
- $(|\vec{B}[P]| = m)$verb: The action *verb* is enabled, with rate ∞, only if the bio-process B of the β-system Z is structurally congruent to the bio-process $\underbrace{\vartheta_0\vec{B}[P] \parallel \cdots \parallel \vartheta_m\vec{B}[P]}_{m} \parallel B'$ and $B' \not\equiv_b \vartheta\vec{B}[P] \parallel B''$.

As far as the *verb* component is concerned, we can distinguish between *split*, *join*, *new* and *delete* actions.

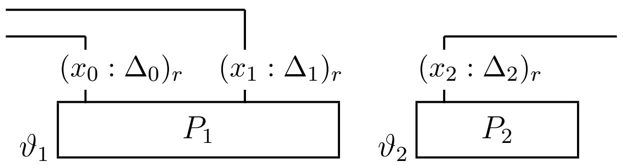
The *split* action is described by a well-formed event of the form

$$(\vec{B}[P] : r)\ \mathsf{split}(\vec{B}_1[P_1], \vec{B}_2[P_2])$$

If the condition is satisfied in the β-system Z, the execution of the *split* action, enabled with rate r, substitutes an occurrence of a box structurally congruent to $\vec{B}[P]$ in B with the parallel composition of the boxes $\vec{B}_1[P_1]$ and $\vec{B}_2[P_2]$. Moreover, modifications in the $\odot_\xi$ relation in the environment ξ can be produced. Indeed, consider the box $\vec{B}[P]$ complexed with other boxes:



and assume that the well-formed event splits the box into boxes that mantains the complexations with the other boxes:



Since the information of the bindings is mantained in the relation $\odot_\xi$, and the labels ϑ_1 and ϑ_2 associated to the created boxes are different with respect to ϑ, then the ξ environment has to be updated with a new relation $\odot'$, where the bindings associated to the consumed box $\vec{B}[P]$ are removed and the new bindings of the created boxes $\vec{B}_1[P_1]$ and $\vec{B}_2[P_2]$ are added. The relation is mantained symmetric.

The *join* action is described by a well-formed event of the form

$$(\vec{B}_1[P_1], \vec{B}_2[P_2] : r)\ \mathsf{join}(\vec{B}[P])$$

If the condition is satisfied in the β-system Z, the execution of the *join* action, enabled with rate r, substitutes an occurrence of boxes structurally congruent to $\vec{B}_1[P_1]$ and $\vec{B}_2[P_2]$ in B with the box $\vec{B}[P]$. Like the *split* action, also the *join* action produces modifications in the environment ξ.

The *delete* action is described by a well-formed event of the form

$$(\vec{B}[P] : r)\ \mathsf{delete}$$

If the condition is satisfied in the β-system Z, the execution of the *delete* action, enabled with rate r, consumes one instance of a box structurally congruent to $\vec{B}[P]$ in B.

The *new* action is described by the well-formed events

$$(\vec{B}[P] : r)\ \mathsf{new}(\vec{B}'[P'], n) \text{ and } (|\vec{B}[P]| = m)\ \mathsf{new}(\vec{B}'[P'], n)$$

These events are enabled (the first with rate r and the second with infinite rate), only if the bio-process B contains at least a box for the first event and exactly m boxes for the second event that are structurally congruent to $\vec{B}'[P']$. The execution of the event, in both cases, creates n copies of the box $\vec{B}'[P']$. However, the compositional nature of the structural operational

---

(split) $\langle \vartheta\vec{B}[P], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta((\|_0\ \vec{B}_0[P_0]) \parallel (\|_1\ \vec{B}_1[P_1])), E, \xi[\odot']\rangle$

where $E = (\vec{B}[P] : r)\ \mathsf{split}(\vec{B}_0[P_0],\vec{B}_1[P_1]) :: E'$ and

$\odot' = (\odot_\xi \setminus \odot_0) \cup (\odot_1 \cup \odot_1^{-1} \cup \odot_2 \cup \odot_2^{-1})$ with

$\odot_0 = \{(\vartheta_0\Delta_0, \vartheta_1\Delta_1) \in \odot_\xi : \vartheta = \vartheta_0 \vee \vartheta = \vartheta_1\}$,

$\odot_1 = \{(\vartheta_0\Delta_0, \vartheta\ \|_0\ \Delta) : \Delta \in \mathit{types}(\vec{B}_0) \wedge \vartheta_0\Delta_0 \odot_\xi \vartheta\Delta\}$,

$\odot_2 = \{(\vartheta_1\Delta_1, \vartheta\ \|_1\ \Delta) : \Delta \in \mathit{types}(\vec{B}_1) \wedge \vartheta_1\Delta_1 \odot_\xi \vartheta\Delta\}$

(join) $\langle \vartheta_0\vec{B}_0[P_0] \parallel \vartheta_1\vec{B}_1[P_1], E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta_0\vec{B}[P] \parallel \mathsf{Nil}, E, \xi[\odot']\rangle$

where $E = (\vec{B}_0[P_0], \vec{B}_1[P_1] : r)\ \mathsf{join}(\vec{B}[P]) :: E'$ and

$\odot' = (\odot_\xi \setminus \odot_0) \cup (\odot_1 \cup \odot_1^{-1})$ with

$\odot_0 = \{(\vartheta\Delta, \vartheta'\Delta') \in \odot_\xi : \vartheta_1 = \vartheta \vee \vartheta_1 = \vartheta'\}$,

$\odot_1 = \{(\vartheta\Delta, \vartheta_0\ \|_0\ \Delta') : \Delta' \in \mathit{types}(\vec{B}_1) \wedge \vartheta\Delta \odot_\xi \vartheta_1\Delta'\}$

(delete) $\langle \vartheta\vec{B}[P], (\vec{B}[P] : r)\ \mathsf{delete}() :: E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \mathsf{Nil}, (\vec{B}[P] : r)\ \mathsf{delete} :: E, \xi\rangle$

(new) $\langle \vartheta\vec{B}[P], (|\vec{B}[P]| = m)\ \mathsf{new}(\vec{B}'[P'], n) :: E, \xi\rangle \xrightarrow{\theta} \langle \vartheta\vec{B}[P], (|\vec{B}[P]| = m)\ \mathsf{new}(\vec{B}'[P'], n) :: E, \xi\rangle$

where $\theta = r, \mathit{new}, (\vec{B}'[P'], m, n)$

(new_c) $\langle \vartheta\vec{B}[P], (\vec{B}[P] : r)\ \mathsf{new}(\vec{B}'[P'], n) :: E, \xi\rangle \xrightarrow{r,\bullet,\epsilon} \langle \vartheta B, (\vec{B}[P] : r)\ \mathsf{new}(\vec{B}'[P'], n) :: E, \xi\rangle$

where $B = \underbrace{(\|_0\ \vec{B}'[P']) \parallel (\|_1\ (\cdots \parallel (\|_1\ \vec{B}'[P'])))}_{n}$

Table 4: Events reduction rules

semantics does not permit to evaluate the actual number of boxes $\vec{B}[P]$ in the whole system only by applying the new axiom. This problem is solved by using a labeled semantics and the θ labels are used to propagate information though the derivation tree. In particular, the new axiom propagate a label $\theta = r, \mathit{new}, (\vec{B}'[P'], m, n)$ which contains the information about the *new* action. Notice that, because of the well-formedness property of events, in this case we have that $\vec{B}'[P'] \equiv_b \vec{B}[P]$ and hence the information present in the label is enough. In the next section we will show how this information is used.

3.3 The stochastic transition system

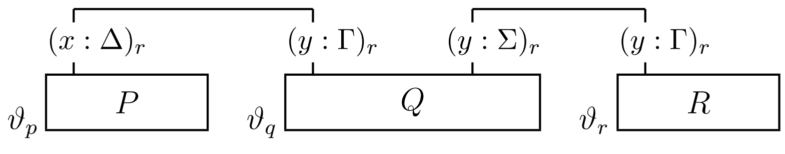
In order to complete the set of rules composing the operational semantics of the BetaSIM language, other three reduction rules has to be presented. The last three rules are reported in Table 5.

Before explaining the behaviour of that rules, we need to introduces some definitions. First, we need to establish what a well-formed β-system is. In the previous sections several well-formedness definitions have been introduced. Now, we want to use all this definitions for providing a notion of well-formedness for a general β-system. Formally,

Definition 9. *Let $Z = \langle B, E, \xi \rangle$ be a β-system with $\xi = (T, \alpha, \rho, \odot)$. Then Z is well-formed only if B is well-formed, E is well-formed, $\odot$ is well-formed and it holds*

$$(\vartheta\Delta, \vartheta'\Gamma) \in \odot \Leftrightarrow (B \equiv_b \vartheta\, \beta^c(x, r, \Delta)\, \vec{B}_1^*[P_1] \parallel \vartheta'\, \beta^c(y, s, \Gamma)\, \vec{B}_2^*[P_2] \parallel B')$$

Now, we need to formally define the notion of *complex*. As previosuly explained, we know that two boxes can complex together by creating a dedicated communication binding. Since the structure of a β-system is described by a parallel composition of boxes and since these boxes can be complexed in several ways, we need to formalize better the notion of complex. For example, given the bio-process:



we know that the first box is complexed with the second one and that the second box is complexed with the third one. Morover, given the relation $\odot = \{(\vartheta_p\Delta, \vartheta_q\Gamma), (\vartheta_q\Gamma, \vartheta_p\Delta), (\vartheta_q\Sigma, \vartheta_r\Gamma), (\vartheta_r\Gamma, \vartheta_q\Sigma)\}$ we infer that $\vartheta_p\Delta\overline{\odot}\vartheta_r\Gamma$.

Now, we introduce a notion of connection between boxes.

Definition 10. *Let $\odot$ be a well-formed complexation relation and let $\vartheta\vec{B}[P]$ and $\vartheta'\vec{B}'[P']$ be well-formed boxes. The box $\vartheta\vec{B}[P]$ is connected with the box $\vartheta'\vec{B}'[P']$, denoted with $\vartheta\vec{B}[P]\overline{\odot}\vartheta'\vec{B}'[P']$, if*

$$\exists\ \Delta \in bc(\vec{B}),\ \Gamma \in bc(\vec{B}') \textit{ such that } \vartheta\Delta\overline{\odot}\vartheta'\Gamma$$

A set of boxes completely connected together can be considered a complex. Formally,

Definition 11. *Let $Z = \langle B, E, \xi \rangle$ be a well formed β-system and let $B \equiv_b B' \parallel B''$ where $B' = \vartheta_1 \vec{B}_1[P_1] \parallel \cdots \parallel \vartheta_n \vec{B}_n[P_n]$. The bio-process B' is a complex in B only if,*

$$\forall i \in \{1, ..., n\}((\exists\ j \in \{1, ..., n\}(i \neq j \wedge \vartheta_i \vec{B}_i[P_i] \overline{\odot} \vartheta_j \vec{B}_j[P_j])) \wedge$$
$$(\nexists\ \vartheta \vec{B}[P]\ in\ B'' : (\vartheta_i \vec{B}_i[P_i] \overline{\odot} \vartheta \vec{B}[P])))$$

Note that the notion of complex is not explicit in our language, but it is a consequence of the presence of complex and decomplex operations. In the next sections we will show how the notion of complex is used in the implementation of the simulator.

(struct) $$\frac{Z_1 \equiv Z_1' \qquad Z_1 \xrightarrow{\theta} Z_2}{Z_1' \xrightarrow{\theta} Z_2}$$

(redex) $$\frac{\langle B, E, \xi \rangle \xrightarrow{\theta} \langle B', E, \xi' \rangle}{\langle B \parallel B_1, E, \xi \rangle \xrightarrow{\theta} \langle B' \parallel B_2, E, \xi'[\odot'] \rangle}$$

where $(B_2, \odot) = \mathfrak{C}(B_1, \vartheta, \odot_\xi)$ and $\odot' = \odot_{\xi'} \cap (\odot_\xi \setminus \odot)$ if $\theta = (r, die, \vartheta)$, while $B_2 = B_1$ and $\odot' = \odot_{\xi'}$ otherwise

(redex_s) $$\frac{\langle B, E, \xi \rangle \xrightarrow{\theta} \langle B', E, \xi' \rangle}{\langle B, E, \xi \rangle \xrightarrow{r}_s \langle B' \parallel B_1, E, \xi' \rangle}$$

where $B_1 = \overbrace{(\parallel_0 \vec{B}[P]) \parallel (\parallel_1 (\cdots \parallel (\parallel_1 \vec{B}[P]))}^{n}$
if $\theta = (r, new, (\vec{B}[P], m, n))$ and $Num(\vec{B}[P], B) = m,$ while
$B_2 =$ Nil otherwise

Table 5: BetaSIM reduction rules

Now, we can introduce the last three reduction rules.

The struct rule, which is standard in reduction semantics, equates the behaviours of structurally congruent β-systems.

The redex rule is used to collect the context and uses a function $\mathfrak{C} : \mathcal{B} \times \vartheta \times \odot \rightarrow \mathcal{B}$, defined on the structure of labeled bio-processes in the following way:

$$\mathfrak{C}(\vartheta' \vec{B}[P], \vartheta, \odot) = \begin{cases} (\mathsf{Nil}, \odot') & \text{if } \exists\ \Delta, \Gamma \in T : \vartheta \Delta \overline{\odot} \vartheta' \Gamma \text{ and} \\ & \odot' = \{(\vartheta_0 \Delta, \vartheta_1 \Gamma) \in \odot : \vartheta_0 = \vartheta' \vee \vartheta_1 = \vartheta'\} \\ (\vartheta' \vec{B}[P], \odot) & \text{otherwise} \end{cases}$$

$$\mathfrak{C}(\vartheta' \vec{B}[P] \parallel B, \vartheta, \odot) = \mathfrak{C}(\vartheta' \vec{B}[P], \vartheta, \odot)\ @\ \mathfrak{C}(B, \vartheta, \odot)$$

where the function @ is defined in the following way:

$$(\vartheta B, \odot)@(\vartheta' B', \odot') = (\vartheta B \parallel \vartheta' B', \odot \cup \odot').$$

The function $\mathfrak{C}$ takes as parameters a bio-process, a label ϑ and a relation $\odot$ and returns a bio-process, obtained from B where all the boxes connected with the box with label ϑ are eliminated, and the $\odot'$ relation, containing all the bindings associated to the eliminated boxes. The global effect of the application of the function $\mathfrak{C}$ in the derivation tree is to eliminate all the boxes belonging to the same complex to which the box that performs a *die* action is part. The redex_s rule is used for constructing the actual transition relation. We introduce this additional level of derivation because of the presence of a particular type of *new* event, which is enabled only if the global system satisfies a condition. Since the operational semantics is compositional, we decided to add a final reduction rule that performs the check for the global condition and that represents the transition relation of our stochastic reduction system. Formally,

Definition 12. *The β-binders* Stochastic Transition System *(*STS*) is referred as $\mathcal{S} = (\mathcal{Z}, \xrightarrow{r}_s, Z_0)$, where $\mathcal{Z}$ is the set of well-formed β-systems, $Z_0 \in \mathcal{Z}$ is the initial β-system and $\xrightarrow{r}_s \subseteq \mathcal{Z} \times \mathbb{R} \times \mathcal{Z}$ is the stochastic reduction relation, where r is a stochastic rate.*

The definition of *STS* is built upon the set of well-formed β-systems $\mathcal{Z}$. We now show that structural congruence and $\xrightarrow{r}_s$ reduction preserve the well-formedness of β-systems.

Properties 1. *Let $\mathcal{Z}$ be the set of well-formed β-systems and $Z = \langle B, E, \xi \rangle \in \mathcal{Z}$. For each β-system $Z' = \langle B', E', \xi \rangle$ such that $Z \equiv Z'$ it holds $Z' \in \mathcal{Z}$.*

Proof. It is straightforward to see that the rules for the structural congruence preserve the well-formedness property for bio-processes and event lists. Moreover, since the status of the all the elementary beta binders and the locations associated to boxes does not change and the environment remains unchanged, then the property reported in the Def. 9 continues to hold, and Z' preserves the well-formedness property. $\square$

Properties 2. *Let $\mathcal{Z}$ be the set of well-formed β-systems and $Z = \langle B, E, \xi \rangle \in \mathcal{Z}$. For each β-system $Z' = \langle B', E', \xi' \rangle$ such that $Z \xrightarrow{r}_s Z'$ it holds $Z' \in \mathcal{Z}$.*

Proof. We report here a sketch of the proof. The property is proved by analyzing the structure of the derivation tree for $\xrightarrow{r}_s$ transitions. In general, each derivation tree has the form:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{axiom}{Z_0 \xrightarrow{\theta} Z'_0}}{Z_1 \xrightarrow{\theta} Z'_1}}{\vdots}}{Z \equiv Z_n \qquad Z_n \xrightarrow{\theta} Z'_n}}{Z \xrightarrow{\theta} Z'_n}}{Z \xrightarrow{r}_s Z'}$$

where all the struct reduction can be applied together after the whole context has been collected. Notice that the application of the rule struct, because of the Property 1, preserves the well-formedness property. Moreover, it is straightword to see that, in the case the function $\mathfrak{C}$ is not applied, also the redex rule preserves the well-formedness property.

For monomolecular axioms intra, hide, unhide, change, expose and tau, and bimolecular axioms inter_c and inter, at the end of the derivation we have that the lists E and E' are structurally congruent and conditions on the rules preserve the well-formedness on the bio-process B'. Moreover, since the status of elementary beta binders and locations associated to boxes does not change and $\xi = \xi'$, then the property reported in the Def. 9 continues to hold, and Z' preserves the well-formedness property. In this cases, since the label θ is empty, no modifications in the resulting β-system caused by the propagation of label informations is produced.

Now consider the axioms new and delete. They cause, respectively, the creation of a number of new boxes and the elimination of a box. However, the created and eliminated boxes does not contain elementary beta binder in complexed status (because the related events are well-formed) and hence no modifications in the environment are produced. No information through labels θ is propagated along the derivation tree and the function $\mathfrak{C}$ is never called. Therefore, at the end of the derivation the lists E and E' are structurally congruent and the bio-process B' is well-formed; indeed, in case of axiom new only well-formed boxes are added, while in the case of axiom delete the bio-process remains obviously well-formed. Hence Z' preserves the well-formedness property.

The execution of the axiom new_c is similar to the execution of the axiom new, but an information through the label θ is propagated along the derivation three. However, because of the structure of θ, also inthis case the function $\mathfrak{C}$ is never called and in the application of the final reduction rule redex_s, if the condition associated to the *new* event holds, only well-formed boxes without elementary beta binders in complexed status are added. Therafter, also in this case Z' preserves the well-formedness property.

Consider the join and split axioms. Their execution produce a modification also in the structure of the complexation relation $\odot_\xi$ of the environment. In particular, each couple $(\gamma, \gamma') \in \odot_\xi$ such that γ' refers to an eliminated box, is substituted with a new couple $(\gamma, \gamma'') \in \odot_{\xi'}$ such that γ'' refers to a new created box, and no elementary beta binders in complexed status without a corresponding relation in $\odot_{\xi'}$ are created. Hence, because of the well-formedness of the events *join* and *split* and the conditions on their axioms, no bindings are lost and no inconsistent complexed beta binders are added. Moreover, no information through labels θ is propagated along the derivation tree and also in this case the function $\mathfrak{C}$ is never called. Moreover, B' is obtained from B by substituing well-formed boxes with other well-formed boxes and E' is congruent to E. Therefore, Z' preserves the well-formedness property.

Finally, consider the axiom die. In the case the box that performs the die action is not part of a complex the derivation is similar to the one already explained. Otherwise, modifications in the resulting β-system are generated also along the derivation tree and in particular, the function $\mathfrak{C}$ is called for each application of the redex reduction rule. At the end of the derivation, in the resulting β-system, all the boxes belonging to the same complex to which the box that performs the die action is part and the related bindings in the environment are eliminated. Notice that the Def. 11 guarantees that the elimination of a complex and its bindings does not corrupt the consistency of the remaining β-system, and hence the property reported in the Def. 9 continues to hold. Moreover, since B' is obtained from B by only eliminating boxes and E' is obviously congruent to E, then B' and E' are well-formed and also in this case Z' preserves the well-formedness property. □

3.4 Other language constructs

In the β-simulator (Section 7) a *join* event of the form:

$$(\vec{B}_0[P_0], \vec{B}_1[P_1] : r) \ \mathsf{join}$$

is provided. This event produces a box $\vec{B}_2[P_2]$ where $\vec{B}_2$ is obtained from $\vec{B}_0$ and $\vec{B}_2$ as the union of the two lists, while P_2 is the parallel composition (with the proper substitutions) of P_0 and P_1. For example, the event:

$$(\beta(x, r, \Delta)\, \beta(y, s, \Gamma)[P_0], \beta(z, t, \Delta)[P_1] : r) \ \mathsf{join}$$

produce the box:

$$B' = \beta(x, r, \Delta)\, \beta(y, s, \Gamma)[P_0 | P_1\{x/z\}]$$

We prefer to avoid the explicit introduction of this event in the BetaSIM language because it is only syntactic sugar. Indeeed, the same result can be obtained by defining the event:

$$(\beta(x, r, \Delta)\, \beta(y, s, \Gamma)[P_0], \beta(z, t, \Delta)[P_1] : r)\ \mathsf{join}(B')$$

4 System architecture

The BetaSIM stochastic simulator (hereafter β-simulator) is the core part of the system. In this section we describe the logical structure of the simulator, the algorithm and data structures it uses as well as the time evolution of a simulation.

4.1 BetaSIM 's logical blocks

The simulator is built as a composition of three logical blocks (see Fig. 2): the *compiler*, the *environment* and the *stochastic engine.*

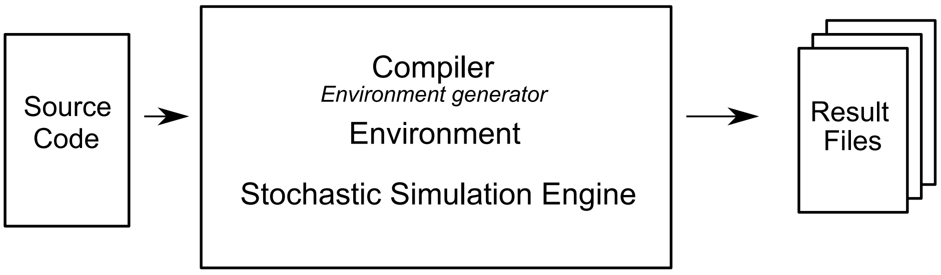


Figure 2: The logical structure of the simulator

The *compiler* translates the source code (a bio-process with a description of its types and a list of events) into a runtime representation that is then stored into the *environment.* The *environment* provides the *stochastic simulation engine* with primitives for checking the state of each entity in the model, create new entities and modify them. 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.

4.1.1 The compiler

The compiler parses the syntactic definition of bio-processes, complexes, events and semantic rules are codified into the data structures *Entities*, *Complex-graphs* and *Elements.*

4.1.2 The environment

The *environment* stores the data structures produced by the compiler This representation is dynamic, as opposed to the the source code which is static. At each time step, the environment holds the current state of the system:

- which entities (species) are present, and their cardinality;
- which complexes are present, and their cardinality;
- the active actions, e.g. actions that can be executed to make the system evolve to the next state, with their next "execution" time;

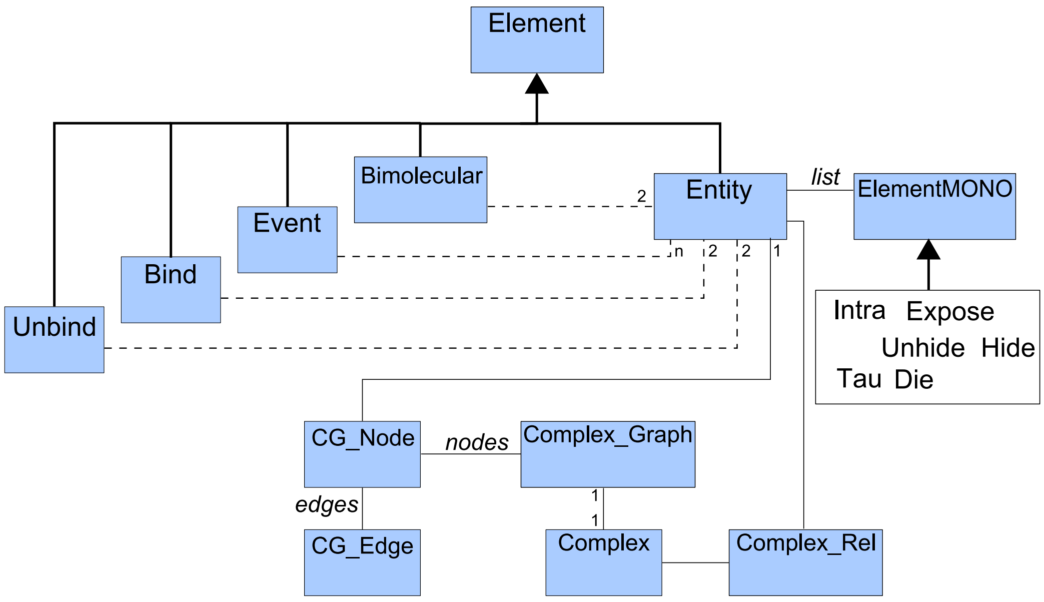


Figure 3: Main classes for environment management

Entities are the representation in the environment of equivalence classes of structural congruent bio-processes. The algorithm to decide whether two bio-processes are congruent is polynomial in time and it is described in details in [18]. The environment assigns to each *entity* in the system an unique identifier ID. This ID is used to identify them in an efficient and unique way. Our concept of entity maps directly to the concept of molecular or biological *species* used in the definition of the Gillespie's stochastic simulation algorithm [10].

The main classes for environment management are depicted in Fig. 3[5]. An object of type *element* is an object with a *timed event*, an action that will

[5]Bind/Unbind classes in the figure codify Complex/Decomplex actions

modify the environment at a given time by acting on entities. For example, a *bimolecular* element interprets a bimolecular reaction, involving exactly two source entities and two target entities:

$$A + B \rightharpoonup^{K_A B} A' + B'$$

For other elements, like *events*, the number of affected entities is variable, but always finite.

Entities are a special case of *element* too: it is not only the target of external (or environment) actions, but also both the *source* and the *target* of internal (or intra) actions. Delays, internal communications, binder modifications are all examples of actions originated by an entity that affect the entity itself (and, possibly, another entity). Therefore, the timed event of an *entity* is the fastest intra action. Intra actions are modelled by objects of type *elementMono*. Their name is due to the fact that their execution interprets a monomolecular reaction:

$$A \rightharpoonup^{K_A} A'$$

In the simulator architecture, *bind* and *unbind* elements are used to simulate complexation actions, while complexed entities are stored in a *Complex* structure. *Complex* is a facade class to hold the internal representation of complexes as graphs of entities (*Complex_Graph*, *CG_Node* and *CG_Edge* classes).

Entities, complexes and elements are held by the system in associative arrays (in Figure 3 the map for complexes *Complex_Rel* is shown) to provide a convenient access to them and to make possible the implementation of an efficient algorithm for stochastic simulation.

4.1.3 The simulation engine

The *simulation engine* relies on a stochastic selection algorithm. A simulation represents a trajectory in the STS generated by the initial β-system. Our *simulation engine* implements an efficient variant of the Gillespie's algorithms described in [8, 10].

The *two level* nature of the β-binders language, with its *intra* and *inter* actions, requires special care for a correct and efficient implementation.

We implemented a new algorithm, called *Next Action Method*, that uses efficient data structures that complement the structures of the environment and a new selection procedure to achieve a correct and performant stochastic simulation.

The data structures used by the *Next Action method* are three: the *Env List*, the *Env Map* and the *Action Queue* (see Fig. 4).

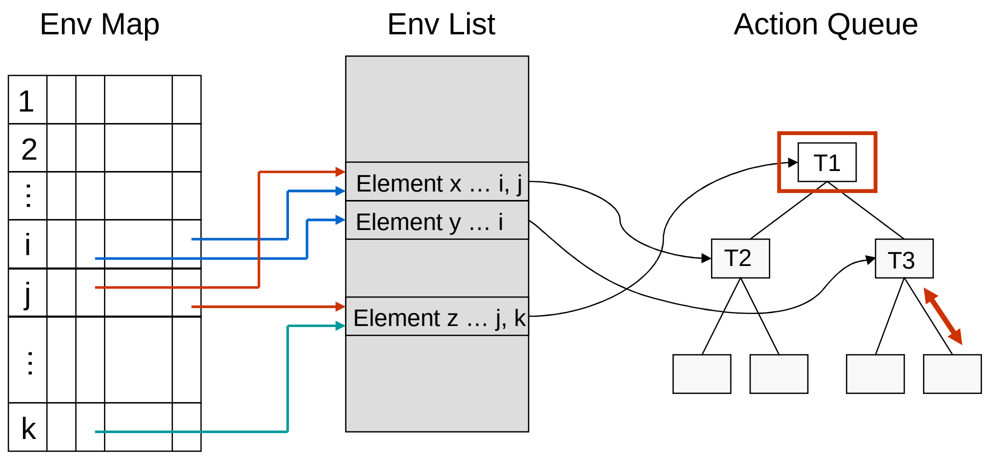


Figure 4: The data structures used by the Next Action Method

The *Env list* holds the list of all the *elements* (i.e. of all the possible actions) of the system. The stochastic selection algorithm chooses the fastest action from this list and makes the system evolve accordingly.

The *Env map* is an associative array that holds the dependency relations between entities and elements in the system. Consider as an example a bio-processes B_1, involved in two reactions:

- it can perform an inter-communication, represented by a *Bimolecular* element *bim* in the environment, so that $bim ::= B_1 + B_2 \rightharpoonup B_1' + B_2'$;

- it can perform an intra-communication, represented by the *Entity* element B_1 itself, so that $B_1 \rightharpoonup B_3$.

The Env map entry for the B_1 entity will be $EnvMap[B_1] = \{B_1, bim\}$. Intuitively, for every entity there is an entry in the *Env map* that holds a list of the elements whose timing can be affected by the entity considered.

The *action queue* is an indexed priority queue implemented as an heap [19]. The queue holds elements ordered by their next reaction time. This semi-ordered data structure allows us to implement the min operation in constant time, overcoming the principal limitation of the environment list. Updating the position of an element in the queue, an operation that must be performed when the next execution time for that element is re-computed, requires $O(\log n)$ time.

4.2 The stochastic algorithm

A simple implementation of the Gillespie's algorithm can use the Env list only; however at each time step the algorithm have to scan directly the list to search for the fastest action. Being it a $O(n)$ operation, this is one of the major bottleneck in the algorithm.

Our algorithm is similar to the efficient variant of the Gillespie's algorithm proposed by Gibson and Bruck [20]. The variant algorithm uses an indexed priority queue and a dependency graph to reduce the complexity of one execution step to $O(\log n)$. Our action queue uses the same data structure. However, since our reactions are modelled as interaction between bio-processes in place of chemical or rewriting rules, it is not efficient for us to build a dependency graph like the one in [20].

A data structure that holds the dependency relations between elements in the system is necessary for obtaining an efficient algorithm. Using a priority queue it is possible to select the fastest action in constant time; but then, if we recompute the next action time for all the entities in the system, we end up with an even worse asymptotic time of $O(n \log n)$ (n update operations). On the other side, using the *Env map* allows us to keep track of the entities that are affected by an element action. The count of these entities, like in the Gibson algorithm, is constant, and so a whole simulation steps will cost $O(\log n)$. The *Env map*, differently from the reaction graph, is built dynamically, so it is possible to use it with process algebras.

More than one *intra* action can be enabled at each time, so it is necessary to select one among the set. The Gillespie's Direct method [8] is used to compute the next reaction time of all the enabled *intra* (monomolecular) actions on a single entity, and then to select the fastest one. The First reaction method [10] is then used to choose the next Element (bimolecular, event, bind or single entity action) that will fire and to compute when it will happen.

The pseudo-code for the three main functions of the Simulator Engine is given in Tables 6 and 7.
The Init function initialises the Environment, setting for each element its next execution time and its position in the *Action Queue*. The next execution time is computed using the ComputeTime function. The time is computed in a different way according to the type of the element, as already explained. The Update function performs the simulation step using a Reduce helper function to adjust element quantities and to compute the set of modified entities. These entities are than used to find all the affected elements using the *Env Map*, as already described. Finally, these elements are updated

```
INIT ():
  for each $Element$ in $Environment$ do
    $Time := \mathsf{ComputeTime}(Element)$;
    $ActionQueue.\mathsf{Insert}(Element.\mathsf{ID}, Time)$;

COMPUTETIME ($Element$):
  switch $Element.\mathsf{Type}$
    case Bimolecular :
      if $Element.\mathsf{Entity1} = Element.\mathsf{Entity2}$ then
        $comb := \frac{Element.\mathsf{Entity1.Quantity} \times Element.\mathsf{Entity2.Quantity}}{2}$;
      else
        $comb := Element.\mathsf{Entity1.Quantity} \times Element.\mathsf{Entity2.Quantity}$;
      $rn := \mathsf{Random}[0..1]$;
      $time := \frac{1}{comb \times Element.\mathsf{Rate}} \times \log \frac{1}{rn}$;
    case Entity :
      $SumRate := \sum_{Entity.\mathsf{Actions}} Action.\mathsf{Rate}$;
      $rn1 := \mathsf{Random}[0..1]$;
      $time := \frac{1}{comb \times SumRate} \times \log \frac{1}{rn1}$;
      $rn2 := \mathsf{Random}[0..SumRate]$;
      $Entity.\mathsf{NextAction} := \mathsf{Select}(Entity.\mathsf{Actions}, SumRate)$;
    case ... :
      ...
    return $time$;
```

Table 6: INITALIZATION and COMPUTETIME.

using COMPUTETIME and their position into the *Action Queue* is adjusted accordingly.

Figure 5 describes one step of the Next Action Method.

The enabled (i.e., ready to fire) actions are stored in a priority queue. The fastest action, the one on the top of the queue, is chosen as the next one that will fire (1). The action is executed by letting the corresponding Element evolve (2). In Fig. 5 a monomolecular reaction is carried out, so its corresponding *intra* action will be handled by an Element of type Entity. The Entity in turn locates and executes its fastest ElementMONO action. The Entity will evolve, resulting in a possibly different species. If this species is already present in the system (3Y), its cardinality is updated and it is added to the list of modified Entities. If the species is new, the simulator creates a

```
UPDATE ():
    $NextElement$ := $ActionQueue$.Top;
    $ModifiedSet$ := Reduce($NextElem$);
    $UpdateList$ := $\emptyset$;
    for each  $Entity$  in  $ModifiedSet$  do
        $UpdateList$ := $UpdateList \cup$ EnvMap[$Entity$];

    for each  $Element$  in  $UpdateList$  do
        $Time$ := ComputeTime($Element$);
        $ActionQueue$.Insert($Element$.ID, $Time$);
```

Table 7: UPDATE.

new Entity that represents it, and adds it to the list of modified Entities. This list is then used by the simulator to locate all the entities for which we need to recompute the next reaction time (5). Due to the memoryless property of the exponential probability distribution, there is no need to recompute the next reaction times for entities that were not involved in the action just executed. Using an efficient hash table we have in constant time a minimal list of the entities affected by the last reaction and we use it to compute the new action times of as few entities as possible (6). The priority queue is then updated (7 - a logaritmic time operation) and the environment is then ready for the next execution step.

5 Complexes

In this section we further exploit the notion of *complex* because it is the most relevant extension to the the basic calculus of beta-binders. For example, consider the bio-process B (Fig. 6) of a well-formed β-system $Z = \langle B, E, \xi \rangle$ with $\odot_\xi = \{(\vartheta_1\Delta_0, \vartheta_2\Delta_1), (\vartheta_2\Delta_1, \vartheta_1\Delta_0), (\vartheta_4\Delta_1, \vartheta_5\Delta_0), (\vartheta_5\Delta_0, \vartheta_4\Delta_1)\}$. The first two and the last two boxes are complexes in B.

In general, non trivial configurations can be obtained, like for example the *chain*, *ring* and *tree* configurations reported in Fig. 7, where:

$$b_0 = (x : \Delta_0)_{r_0}$$
$$b_1 = (y : \Delta_1)_{r_1}$$
$$b_2 = (z : \Delta_2)_{r_2}$$

and $\forall i \in \{1, ..., 4\}\ P_i \not\equiv P_{i+1}$.

A complex of boxes can intuitively be represented as a graph, where box are nodes and complexation bindings are edges. This representation is

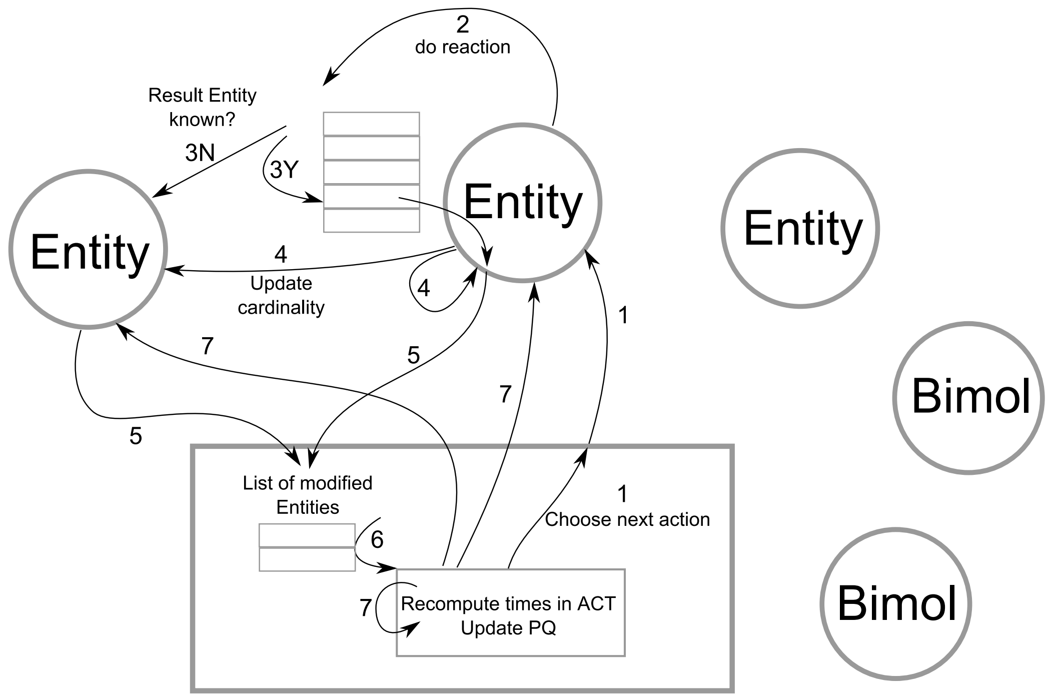


Figure 5: One step of execution in the Next Action Method. In this case, a monomolecular reaction

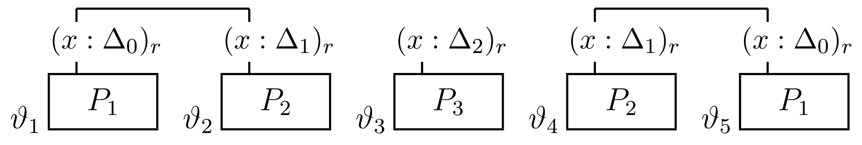


Figure 6: A well-formed β-system.

also justified by the fact that in the simulator a complex is implemented using a graph data structure. Moreover, since the simulator environment (Sect. 4) holds information about which complexes are present and their cardinality, we have to explain when two complexes are considered equal. The graph representation gives us an intuitive tool for reasoning about equality of complexes that results to be a graph isomorphism problem.

5.1 Graph representation

A box represents a biological interacting entity and it is composed by an internal structure and an interface. Two boxes belong to the same species if they are structurally congruent. According to the context-free grammar defining

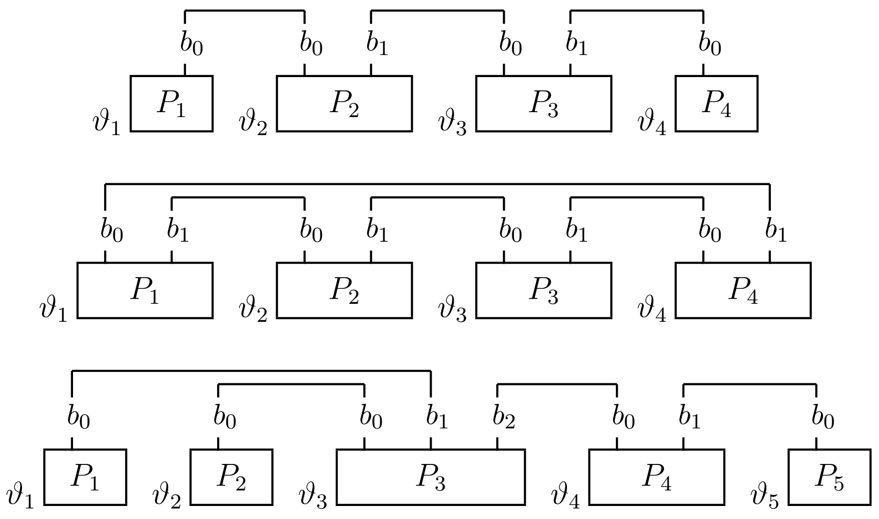


Figure 7: Example of *chain*, *ring* and *tree* complex configurations.

the syntax of the language, we can only define a countable set of boxes, and according to laws of the structural congruence the set of equivalence classes induced by the relation $\equiv_b$ is also countable. Moreover, the number of boxes belonging to the same congruence class is infinite (e.g. $\vec{B}[P|nil|nil|\cdots])$.

For the BetaSIM language (see [18]), the problem of deciding whether two boxes are structurally congruent is equivalent to a tree isomorphism problem. We can define a normal form NF for boxes belonging to the same equivalence class such that:

$$B[P] \equiv_b B'[P'] \Leftrightarrow NF(B[P]) = NF(B'[P'])$$

Each normal form identifies in a unique way a congruence class and the set of all the possible normal forms is countable. Hence, denoting with $\mathcal{NF}$ the set of all the normal forms, it is possible to define a Gödel numbering $cl : \mathcal{NF} \to \mathbb{N}$ with both cl and cl^{-1} being computable functions.

We now introduce the graph represention of a bio-process B with respect to a complexation relation $\odot$. We refer to labeled directed graphs (LDAGs) $G = (E, V)$ with:

$$V \subseteq \mathcal{L}^* \times \mathbb{N} \text{ and } E \subseteq V \times T^2 \times V$$

Therefore, the considered graphs have labeled nodes and labeled edges. Labels on nodes are couples (ϑ, n), with $\vartheta \in \mathcal{L}^*$ and $n \in \mathbb{N}$; ϑ is the location associated to the box and $n \in \mathbb{N}$ is the natural number representing the congruence class to which the box belongs. Labels on edges are couples

(Δ, Γ), with $\Delta, \Gamma \in T$; Δ and Γ are the types of the elementary beta binders on which the box represented by the source and target nodes of the edge are complexed.

Let $B = \vartheta_1 \vec{B}_1[P_1] \parallel \vartheta_2 \vec{B}_2[P_2] \parallel \cdots \parallel \vartheta_n \vec{B}_n[P_n]$ and $\odot$ be a complexation realation. The boxes composing the bio-process B represents the nodes of the graph (Fig. 8), where $\forall i \in \{1, ..., n\}, m_i = cl(NF(B_i[P_i]))$.

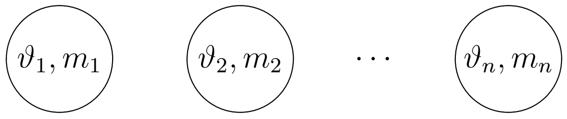


Figure 8: Nodes of the graph representaion of the bio-process $B = \vartheta_1 \vec{B}_1[P_1] \parallel \vartheta_2 \vec{B}_2[P_2] \parallel \cdots \parallel \vartheta_n \vec{B}_n[P_n]$.

The pairs $(\gamma, \gamma') \in \odot$ contain the information for obtaining the edges of the graph. For example, supposing $(\vartheta_1 \Delta, \vartheta_2 \Gamma) \in \odot$, then an edge between the nodes of the boxes with locations ϑ_1 and ϑ_2 exists and it is labeled (Δ, Γ) (Fig. 9). Obviously, since the reaction $\odot$ is symmetric, also the inverse pair

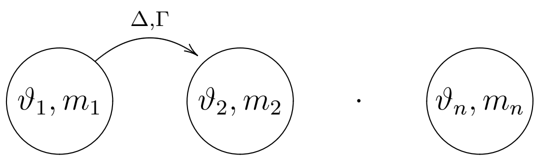


Figure 9: Example of an edge.

$(\vartheta_2 \Gamma, \vartheta_1 \Delta)$ is in $\odot$ and hence each complexation binding in the bio-process is represented with two edges in the graph (Fig. 10). Moreover, since the locations identify in a unique way all the boxes, then there is no ambiguity in the association of edges between nodes. For a more detailed example,

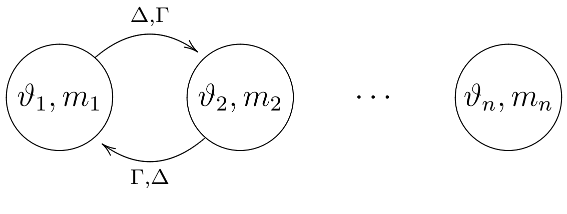


Figure 10: Example of the edges representing a complexation binding.

consider the bio-processes reported in Fig. 7. Their graph representations are reported in Fig. 12.

We now provide the formal definition of the graph representation of complexes.

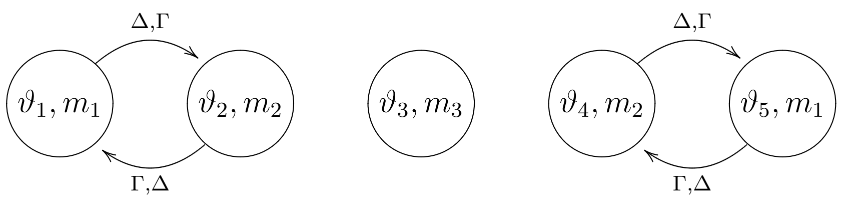


Figure 11: Graph representation of a well-formed β-system.

Definition 13. *Let $B = \vartheta_1\vec{B}_1[P_1] \parallel \vartheta_2\vec{B}_2[P_2] \parallel \cdots \parallel \vartheta_n\vec{B}_n[P_n]$ be a bio-process and let $\odot$ be a complexation relation. We denote with $G(B, \odot) = (E, V)$ the graph representation of B with respect to $\odot$ where:*

$$V = \{(\vartheta_i, m_i) : i \in \{1, ..., n\} \wedge m_i = cl(NF(B_i[P_i]))$$
$$E = \{(v, (\Delta, \Gamma), v') : v, v' \in V \wedge v = (\vartheta, m) \wedge v' = (\vartheta', m') \wedge \vartheta\Delta \odot \vartheta'\Gamma\}$$

Let $Z = \langle B, E, \xi\rangle$ be a well-formed β-system. The graph representation of the β-system Z is the graph generated by the bio-process B with respect to the complexation relation $\odot_\xi$. For example, the graph representation of the bio-process B in Fig. 6 and its related complexation relation $\odot_\xi$ is reported in Fig. 11. Note that the connected components of the graph represents complexes, while the single node represent a not complexed box. Therefore, we can define a complex in terms of graph representation of bio-processes.

Definition 14. *Let $Z = \langle B, E, \xi\rangle$ be a well formed β-system and let $B \equiv_b B' \parallel B''$. The bio-process B' is a complex in B only if the graph representation $G(B', \odot_\xi)$ is a connected component of the graph representation $G(B, \odot_\xi)$.*

5.2 Complex equality

Let $Z = \langle B, E, \xi\rangle$ and let B' and B'' be complexes in B. To establish whether the two complexes are equal, we consider their graph representations $G(B', \odot_\xi)$ and $G(B'', \odot_\xi)$, and verify the equality of the graphs. This means that we have to establish if the two graphs are isomorph or not. The isomorphism relation we consider, denoted with $\simeq$, is the one in Def. 15.

Definition 15. *Let B and B' be bio-processes and let $\odot$ be a complexation relation. Let $G(B, \odot) = (V_1, E_1)$ and $G(B', \odot) = (V_2, E_2)$ be their graph representations. Then $G(B, \odot)$ and $G(B', \odot)$ are isomorph, denoted with $G(B, \odot) \simeq G(B', \odot)$, if there is a bijection φ between V_1 and V_2 such that $v = (\vartheta, n) \in V_1$ implies $\varphi(v) = v' = (\vartheta', n)$ and for every pair $v, v' \in V_1$, $(v, (\Delta, \Gamma), v') \in E_1$ if and only if $(\varphi(v), (\Delta, \Gamma), \varphi(v')) \in E_2$.*

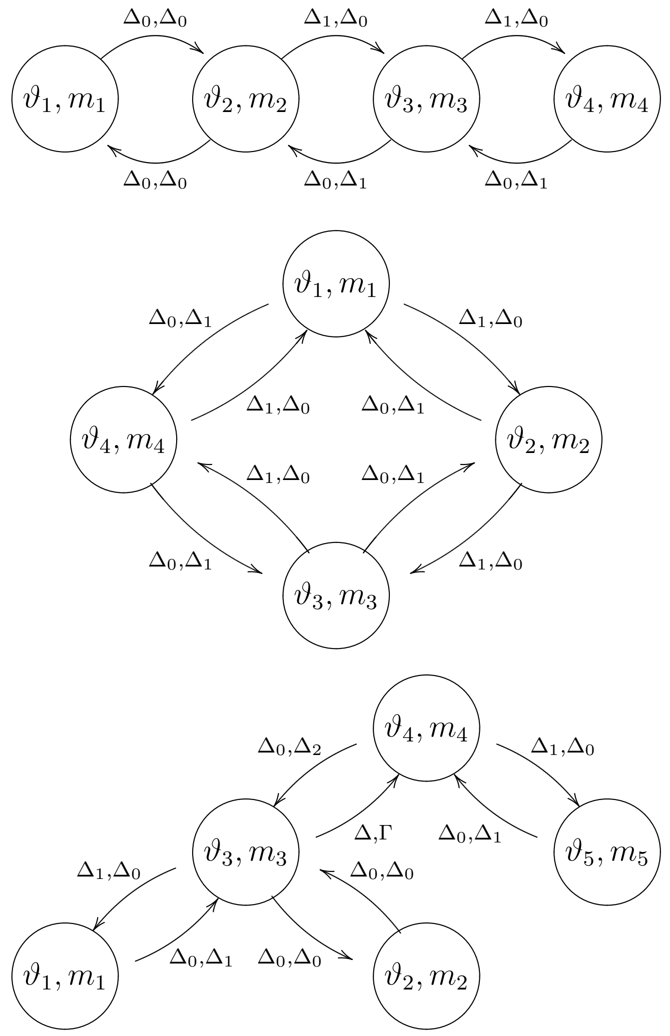


Figure 12: Graph representation of the *chain*, *ring* and *tree* complex configurations.

As an example, consider the graph representation reported in Fig. 11; there are two connected components that represent complexes. In order to establish if the two complexes are equal or not we have to verify if they are isomorph with respect to the ismorphism relation $\simeq$. It is simply to see that they are isomorph and indeed a possible bijection φ between the nodes of the graphs is $\varphi(\vartheta_1, m_1) = (\vartheta_5, m_1)$ and $\varphi(\vartheta_2, m_2) = (\vartheta_4, m_2)$.

The LDAG isomorphism problem [21] is placed in the complexity class *GI*, which contains all the problems equivalent to the general graph isomorphism problem. No polinomialy resolution algorithm for the problems in *GI* has been still found and it is not known if they are or not *NP-complete.*

However, for the class of graphs we consider, the LDAG isomorphism problem results to be of quadratic complexity in the number of graph nodes. In particular, due to the property of well-formedness of boxes (Sect. 3), it is possible to implement an algorithm that verifies if two graph representation of boxes are isomorph or not in polynomial time. The pseudocode of the algorithm is reported in Table 8.

```
ISOMORPHISM $(G_1, G_2)$:
   take $v = (\vartheta, n) \in V_1$;
   for each  $v' = (\vartheta', n')$  in  $V_2$  do
      if  $n = n'$  then
         res := DFSmatching($G_1$,$G_2$,$v$,$v'$) ;
         if  res  then
            return  true

   return  false;

DFSMATCHING $(G_1, G_2, v_1, v_2)$:
   if  IsVisited($v_1$) $\neq$ IsVisited($v_2$)  then
      return  false
   if  IsVisited($v_1$)  then
      return  true
   SetVisited($v_1$);
   SetVisited($v_2$);
   for each  $e = (v_1, (\Delta, \Gamma), v_1' = (\vartheta_1, n))$  in  $E_1$  do
      if  $\exists\ e_2 = (v_2, (\Delta, \Gamma), v_2' = (\vartheta_2, n)) \in E_2$  then
         res := DFSmatching($G_1$,$G_2$,$v_1'$,$v_2'$);
         if  $\neg$ res  then
            return  false;

      else
         return  false;

   return  true;
```

Table 8: Pseudocode of the algorithm that verifies if two graph representations of complexes are isomorph.

5.3 Inheritance

In this subsection we explore the interplay between events and complexes. Every bio-process can have some events *attached* to it; an event is said to be attached to a bio-process B if it is named in the *condition* part the event. Consider for instance a β-system $Z = \langle B, E, \xi \rangle$ such that B is the parallel composition of the boxes: [6]

$$\begin{aligned}
&A \triangleq \beta(x:\Delta)\beta(y:\Gamma)[\ P_A\ ]\\
&B \triangleq \beta(x:\Delta)[\ P_B\ ]\\
&C \triangleq \beta(x:\Delta)\beta(y:\Gamma)[\ P_A \mid P_B\ ]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{e}_1 \triangleq (A, B, K_C)\ \textit{join}\ (C)\\
&\mathsf{e}_2 \triangleq (C, K_C^{-1})\ \textit{split}\ (A, B)\\
&\mathsf{e}_3 \triangleq (A, K_{decay})\ \textit{delete}
\end{aligned}$$

The events attached to A are (e_1, e_3), while e_2 is attached to C. We have to choose a strategy to handle the case in which the bio-process A forms a complex with another one. Consider the situation in Fig. 13 (where the indexes to the names of boxes are only meant to disambiguate different occurrences of the same box): it makes no sense to allow e_3 to happen on A_1 (it will left the system in an unconsistent state because A is part of a complex). On the other hand, the splitting of C_1 will leave the system in a consistent state, i.e. a state in which complexed binders are preserved, as shown in the Figure. Therefore, it is possible to let the split event e_2 happen on C_1, while it is not possible, for example, to allow e_2 to happen on C_2: it will led the system in a non consistent state.
We define the concept of *inheritance* of events.

Definition 16. *An event e is inheritable if one of the following holds*

1. $e \triangleq (\boldsymbol{B}_1, \boldsymbol{B}_2, r)$ *join* $(\boldsymbol{B}_3)$ *with* $bc(B_1) \cap bc(B_2) = \varnothing$ *and* $bc(B_1) \cup bc(B_2) \subseteq bc(B_3)$
2. $e \triangleq (\boldsymbol{B}_1, r)$ *split* $(\boldsymbol{B}_2, \boldsymbol{B}_3)$ *with* $bc(B_2) \cap bc(B_3) = \varnothing$ *and* $bc(B_2) \cup bc(B_3) \supseteq bc(B_1)$

Inheritance is an optional feature of the language. Hence we introduce the keyword **inherit** that must prefix the definition of a split or a join event in order to let it being inheritable. The keyword works as a directive to the compiler.

[6]Hereafter we will use macros to provide pi-processes and bio-processes with names. These are not intended to be agent or constant definitions, but only a shorthand to write programs.

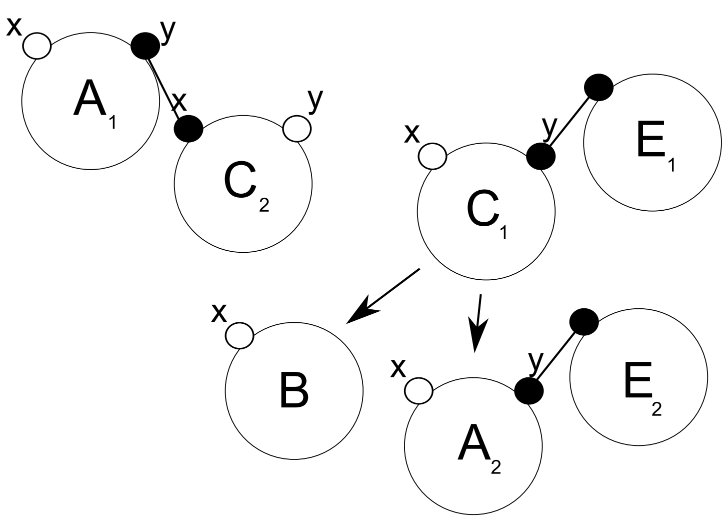


Figure 13: A system with complexes

As explained in Sect. 4, when an entity is bound on one of its binders, it becomes effectively a different entity. So for example:

$$A \triangleq \beta(x : \Delta)\beta(y : \Gamma)[\ P_A\ ]$$
$$A^i \triangleq \beta^c(x : \Delta)\beta(y : \Gamma)[\ P_A\ ]$$

are different entities. The second one is created, by the user in the system specification or by the system during the evolution, when a bio-process A is bound on $(x : \Delta)$. A^i is created using A as a *template*; when this happens, the inheritance conditions of Def. 16 are checked on each of the attached events that are declared as inheritable, and the events that can be inherited are attached to the newly created entity. So, for A^i, event e_3 cannot be inherited, while e_1 can only be inherited if $\beta(x : \Delta)$ is not bound in either A or B (Fig. 14).

In the current release of the language inheritance is the only mechanism that allows to an entity that is bound on one of its binders to have events: it is not possible to explicitly declare an event involving an entity with bound binders. This shortcoming will be addressed in the next release of the language.

6 Modeling patterns

In this section we first report some modelling examples and then we highlight the design strategy used to model biological systems providing modeling

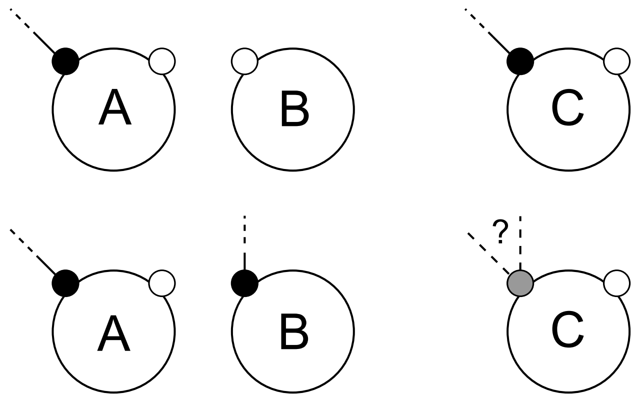


Figure 14: Event inheritance with bound subjects

patterns.

6.1 Examples

In this subsection we first report the simple Lotka-Volterra model of predators and preys. We then report a more comprehensive example by considering the NO-cGMP pathway.

6.1.1 Lotka-Volterra

Consider the simple Lotka-Volterra predator prey model, introduced as ecological model in [22]. There are two entities *Prey* and *Predator*, which represent general species of animals, and a third entity *Food*. The *Prey* species multiplies after feeding on *Food*, and the *Predator* species multiplies after feeding on *Prey*.

This scenario is represented in BetaSIM with a program $Z_{PP} = \langle B, E, \xi \rangle$, where B is the parallel composition of the boxes:

$$\overset{(eat:\Delta_{Hunt})_{r_1}}{\boxed{x(e).(Predator_p|Predator_p)|Predator_p}} \quad (B_1)$$

$$\overset{(eat:\Delta_{Hunt})_{r_2}\quad(food:\Delta_{Life})_{r_3}}{\boxed{x(e).(Prey_p|Prey_p)|Prey_p}} \quad (B_2) \qquad \overset{(feed:\Delta_{Life})_{r_4}}{\boxed{x(e).Food_p|Food_p}} \quad (B_3)$$

where:

$$\begin{aligned}
Prey_p &\triangleq \overline{food}\langle e\rangle.\overline{x}\langle e\rangle.\mathsf{nil} + eat(p).\mathsf{nil}\\
Predator_p &\triangleq (\tau, r).\mathsf{nil} + \overline{eat}\langle p\rangle.\overline{x}\langle e\rangle.\mathsf{nil}\\
Food_p &\triangleq feed(e).\overline{x}\langle e\rangle.\mathsf{nil}
\end{aligned}$$

and where the list of events E is defined in the following way:

$$\begin{aligned}
&(\beta(eat, r_1, \Delta_{Hunt}[\ Predator|Predator\ ] : r)\ \mathsf{split}\ (B_1, B_1);\\
&(\beta(eat, r_2, \Delta_{Hunt})\beta(food, r_3, \Delta_{Life})[\ Prey|Prey\ ] : r)\ \mathsf{split}\ (B_2, B_2);\\
&(\beta(eat, r_1, \Delta_{Hunt}[\ \bar{x}.(Predator|Predator)\ ] : r)\ \mathsf{delete};\\
&(\beta(eat, r_2, \Delta_{Hunt})\beta(food, r_3, \Delta_{Life})[\ \bar{x}.(Prey|Prey)\ ] : r)\ \mathsf{delete};
\end{aligned}$$

The box B_1 represents the *Predator*, the box B_2 represents the *Prey* and the box B_3 represents the *Food*. Assume $\alpha(\Delta_{Hunt}, \Delta_{Hunt}){=}(r, s, t)$ and $\alpha(\Delta_{Life}, \Delta_{Life}){=}(r', s', t')$ with all the resulting values greater than zero. A *Predator* can eat a *Prey* and this is represented as an *inter-communication*, with rate t, between the corrisponding boxes on the compatible elementary beta binders with type *Hunt*:

$$\overset{(eat : \Delta_{Hunt})_{r_1}}{\boxed{x(e).(Predator_p|Predator_p)|\overline{x}\langle e\rangle.\mathsf{nil}}}\quad (B_4)$$

$$\overset{(eat : \Delta_{Hunt})_{r_2}\quad (food : \Delta_{Life})_{r_3}}{\boxed{x(e).(Prey_p|Prey_p)|\mathsf{nil}}}\quad (B_5)$$

Then, by consuming the *intra-communication* on the channel x with rate $\rho(x)$, the *Predator* changes its species to one with more $Predator_p$ pi-processes inside, but it not increase its cardinality:

$$\overset{(c : \Delta_{Hunt})_{r_1}}{\boxed{Predator_p|Predator_p}}\quad (B_6)$$

To draw a parallel with a biological phenomena, the state of the *Predator* bio-process after reacting with a *Prey* is the same of a cell stuck in the *Telophase* stage in *mitosis*, not being able to perform *Cytokinesis*. At this point, because of the structure of the *Predator*, the event e_1 present in the list E is enabled and the *Predator* can multiply itself:

$$\overset{(eat : \Delta_{Hunt})_{r_1}}{\boxed{x(e).(Predator_p|Predator_p)|Predator_p}} \quad (B_1)$$

$$\overset{(eat : \Delta_{Hunt})_{r_1}}{\boxed{x(e).(Predator_p|Predator_p)|Predator_p}} \quad (B_1)$$

The sequence of interactions that allow *Prey* species to multiply after feeding on *Food* is similar to the one just presented for predators.

Assuming an initial abundance of preys, the predator number grows and reduces the number of prey. This results in a reduction of predators and, since the source of food is limitless, in a consequent new growth in the number of preys. The initial abundance of prey is hence red-establish, and the cycle continues. The result of the simulation is shown in Fig.26 and the complete code is provided in B.

6.1.2 The NO-cGMP pathway

Hypertension is a medical condition where the blood pressure is chronically elevated. Hypertension affects almost one third of population in developed countries; however, a lot of therapies are known and widely used. One of the approaches is to intervent on the *vascular tone*, the degree of constriction experienced by a blood vessel relative to its maximally dilated state. Vascular tone is primarily dependent on a protein called *myosin light chain kinase* (MLCK). This kinase increases the phosphorylation of *myosin light chains*, thereby increasing smooth muscle tension and causing vasoconstriction. The correspondent phosphatase dephosphorylate the myosin light chains, causing vasodilation. MLCK is activated by the *calcium-calmodulin* complex, and therefore the activity of this kinase is influenced by intracellular calcium concentration. To control the level of active MLCK in the cell it is possible to modulate several signal transduction mechanisms: 1) phosphatidylinositol pathway, 2) G-coupled-protein (cAMP) pathway, and 3) nitric oxide (NO)-cGMP pathway [23, 24].

Nitric oxide (NO) is produced by vascular endothelium, and many other cell types, by the *nitric oxide synthase* (NOS), which uses amino acid L-arginine and oxygen as substrates.

When NO is formed in an endothelial cell it readily diffuses into an adjacent smooth muscle cell. Here it binds to a heme domain on *guanylyl cyclase* and activates this enzyme, which produce cGMP from GTP. The increased level of cGMP activates a kinase that subsequently inhibits the

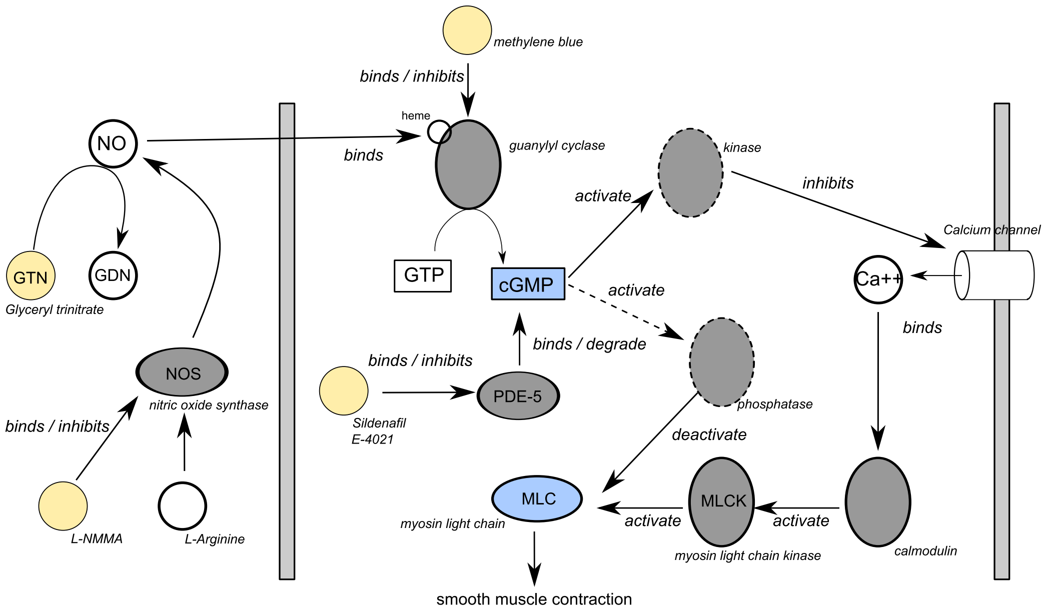


Figure 15: The NO-cGMP pathway in vessel smooth cells

flux of calcium into the vessel smooth cell, decreasing the concentration of calmodulin-calcium complexes and therefore the level of active MLCK.

There is also evidence in literature [25, 26] that increases in cGMP can also lead to myosin light chain de-phosphorylation by activating a pertinent phosphatase.

Fig. 15 shows a complete overview of the pathway. Enzymes are in grey ovals, unknown enzymes having a dotted outline, and drugs that can interact with the pathway are in light grey circles. As underlined in the figure, there are several points in which a drug can act to influence the cGMP concentration in vessel smooth muscle cells. N[ω]-monomethyl-L-arginine (L-NMMA) acts as a competitive inhibitor of nitric oxide synthase, decreasing the level of NO and therefore of cGMP. The level of NO can be increased by the introduction of *glyceryl trinitrate* (GTN), a prodrug which is denitrated, by a mechanism that is widely disputed, to produce 1,2-glyceryl dinitrate (GDN) and NO [27]. Drugs like sildenafil (Viagra) or E-4021 inhibit cGMP-specific phosphodiesterase 5, the enzyme responsible for the degradation of cGMP, and therefore compensate for reduced NO release and cGMP production [28]. In Fig. 16[7] part of the model we developed for the (NO)-cGMP pathway is

[7]In the figure there are some bio-processes defined recursively. It is not possible to have recursive definitions in the language, and this is done here only for clarity. In Section 7 it is explained how to transform them into replications, while in Appendix B the model for the whole pathway is given

$P_{NO} \triangleq \overline{act}.r.nil$ $\qquad P_{GC} \triangleq act.\overline{p}.r.nil$

$P_{GMP} \triangleq (p.unhide(d).r.nil + d.nil)$ $\qquad P_{PDE5} \triangleq \overline{d}.ch(d,\Omega).ch(d,\Delta_{coD}).r.nil$

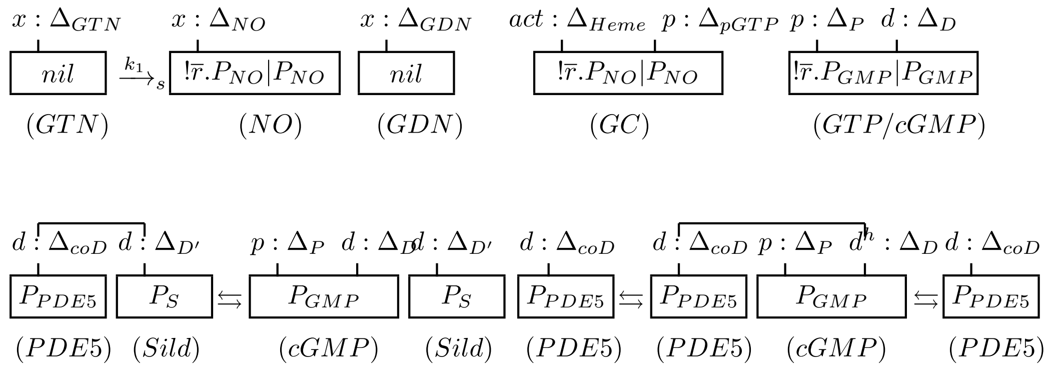


Figure 16: Part of the model for the NO-cGMP Pathway

illustrated. The first line of boxes in the figure shows the processes involved in the production of NO and in the synthesis of cGMP. It is not well established how GTN is transformed into GDN and NO, so this biochemical reaction is modelled as a *split* with rate equal the global observed rate k_1. The synthesis of cGMP is modelled as a communication on binders, as introduced in Section 3. The bio-process GC communicate with GTP through its binder of type Δ_{pGTP}. The communication changes the internal structure of the bio-process, transforming it into a process modelling $cGMP$. The second line of boxes in Fig. 16 indicate how $PDE5$ degrades $cGMP$ by binding it on the Δ_{coD} domain and sending a d message (right side of the model). The $cGMP$ bio-process reacts to the d message; it is transformed into an empty bio-process. It is also shown how competitive inhibition by $Sildenafil$ drug is modelled with a compatible domain type $\Delta_{D'}$. $Sildenafil$ binds to $PDE5$ on Δ_{coD}, but do not react to the degrade message (left side of the model).

6.2 Modeling templates

In this subsection we introduce some general templates that provide guidelines on how to model some common biological phenomena. These templates, like programming languages design patterns, have to be adapted to the specific problem the user is modeling.

6.2.1 Endless supply

Consider this simple chemical reaction:

$$X + Y \rightarrow Y1 + X$$

where the concentration of the reagents X is kept constant. This concept can be expressed in our language with a guarded replication in the definition of a bio-process B_x

$$\begin{aligned} X &\triangleq \mathit{consume}(e).\overline{r}\langle e\rangle.\mathit{nil} \\ B_x &\triangleq \beta(\mathit{consume}, s, \Delta)[\ !r(e).X \mid X\ ] \end{aligned}$$

The same result can be obtained in an alternative by defining an event

$$(\beta(\mathit{consume}, s, \Delta)[\ \mathit{nil}\ ] : s')\ \mathsf{new}(B'_x, 1)$$

that allows us to restore the concentration of X after a copy has been used. The bio-process B'_x is defined in the following way:

$$B_x \triangleq \beta(\mathit{consume}, s, \Delta)[\ \mathit{consume}(e).\mathit{nil}]$$

6.2.2 Mitosis and meiosis

The processes of cellular division (*mitosis* and *meiosis*) can be easily defined using split events. Mimicking the biological phenomenon, the π-process inside a box representing a cell is duplicated (*interphase* to *anaphase*), then the box is split in two parts (*telophase* - *cytokinesis*). The replication of

$$\begin{aligned} \mathit{Cell} &\triangleq \ldots + \ldots + \mathit{duplicate}(e).\mathit{nil} \\ \mathit{CellP} &\triangleq !\overline{\mathit{duplicate}}\langle e\rangle.(\mathit{Cell}|\mathit{Cell})|\mathit{Cell} \\ B_{\mathit{Cell}} &\triangleq \vec{B}_{\mathit{Cell}}[\ \mathit{CellP}\ ] \\ \mathit{Cell}_{\mathit{Anaphase}} &\triangleq \vec{B}_{\mathit{Cell}}[\ (\mathit{Cell}|\mathit{Cell})\ ] \end{aligned}$$

$$\mathsf{mitosis} = (\mathit{Cell}_{\mathit{Anaphase}} : s)\ \mathit{split}\ (B_{\mathit{Cell}}, B_{\mathit{Cell}});$$

Figure 17: A very simplified model of mitosis

the π-process is codified into the *Cell* and *CellP* process in Fig. 17. This pair of processes is equivalent to the single process $\mathit{Cell} \triangleq \ldots + \ldots + \mathit{duplicate}(e).(\mathit{Cell}|\mathit{Cell})$; however, using two processes allow us to write recursive programs without using recursive definition, which are not allowed in the language. This technique is explained in details in Section 7.

The generic model for cellular division in Fig. 17 can be taken as a template and modified according to the properties and level of detail of interest. For example, in a diploid organism the Genome holds two copies of each chromosome, one from each of the parent organisms:

$$\begin{aligned} \mathit{Genome} &\triangleq (\mathit{Chr}_1 | \mathit{Chr}_2) \\ \mathit{Cell} &\triangleq (\mathit{Genome} | \mathit{OtherProcesses}) + \mathit{duplicate.nil} \\ B_{\mathit{Cell}} &\triangleq \vec{B}_{\mathit{Cell}}[\ \overline{\mathit{duplicate}}.(\mathit{Cell}|\mathit{Cell})|\mathit{Cell}\ ] \end{aligned}$$

In this case, mitosis can be modelled using the following split event:

$$\begin{aligned} \mathit{Cell}_{\mathit{Anaphase}} \triangleq \vec{B}_{\mathit{Cell}} \quad & [\ (\mathit{Chr}_1|\mathit{Chr}_2|\mathit{OtherProcesses}) \\ & \ |\ (\mathit{Chr}_1|\mathit{Chr}_2|\mathit{OtherProcesses})\ ] \end{aligned}$$

$$\mathsf{Mitosis} \triangleq (\mathit{Cell}_{\mathit{Anaphase}} : r)\ \mathit{split}\ (B_{\mathit{Cell}}, B_{\mathit{Cell}});$$

In a very similar fashion, meiosis can be modelled on the same bio-processes, changing only the split events:

$$\begin{aligned} \mathit{Cell}_{\mathit{AnaphaseI}} &\triangleq \vec{B}_{\mathit{Cell}} \quad [\ (\mathit{Chr}_1|\mathit{Chr}_2|\mathit{OtherProcesses}) \\ &\qquad\quad\ |\ (\mathit{Chr}_1|\mathit{Chr}_2|\mathit{OtherProcesses})\ ] \\ \mathit{Cell}^1_{\mathit{AnaphaseII}} &\triangleq \vec{B}_{\mathit{Cell}} \quad [\ \mathit{Chr}_1|\mathit{Chr}_1|\mathit{OtherProcesses}\ ] \\ \mathit{Cell}^2_{\mathit{AnaphaseII}} &\triangleq \vec{B}_{\mathit{Cell}} \quad [\ \mathit{Chr}_2|\mathit{Chr}_2|\mathit{OtherProcesses}\ ] \end{aligned}$$

$$\begin{aligned} \mathsf{reductional} &\triangleq (\mathit{Cell}_{\mathit{AnaphaseI}} : r)\ \mathit{split}\ (&& \mathit{Cell}^1_{\mathit{AnaphaseII}}, \\ & && \mathit{Cell}^2_{\mathit{AnaphaseII}}) \\ \mathsf{equational}_1 &\triangleq (\mathit{Cell}^1_{\mathit{AnaphaseII}})\ \mathit{split}\ (&& \vec{B}_{\mathit{Cell}}[\ \mathit{Chr}_1|\mathit{OtherProcesses}\ ], \\ & && \vec{B}_{\mathit{Cell}}[\ \mathit{Chr}_1|\mathit{OtherProcesses}\ ]) \\ \mathsf{equational}_2 &\triangleq (\mathit{Cell}^2_{\mathit{AnaphaseII}})\ \mathit{split}\ (&& \vec{B}_{\mathit{Cell}}[\ \mathit{Chr}_2|\mathit{OtherProcesses}\ ], \\ & && \vec{B}_{\mathit{Cell}}[\ \mathit{Chr}_2|\mathit{OtherProcesses}\ ]) \end{aligned}$$

We need to define three split functions, as meiosis is composed by a *reductional division* (the biological process in which the homologues that form each chromosome pair separate) followed by a double *equational division* (the biological process during which sister chromatids separate; for details, see [29]). Note that, mirroring the biological reality, mitosis can be said to engage in a cell cycle, since after the split event each of the daughter cells has the same bio-process (and so the same behaviour) of the mother cell. Meiosis, on the other side, is a "one-way" process, leading to four haploid cells (the gametes).

6.2.3 Translation of chemical equations

Consider the simple reaction $A + B \rightharpoonup^{K_C} C$; a way to represent species A, B and C is to use three bio-prcesses that differ in the exposed types:

$$\begin{aligned} A &\triangleq \beta(x : \Delta_A)[\ nil\ ] \\ B &\triangleq \beta(x : \Delta_B)[\ nil\ ] \\ C &\triangleq \beta(x : \Delta_C)[\ nil\ ] \end{aligned}$$

and add one join event for the reaction

$$(A, B : K_C)\ join\ (C)$$

If we want the reaction to be revesible, i.e. $A + B \rightleftharpoons^{K_C}_{K_C^{-1}} C$, we have to add another event:

$$(C : K_C^{-1})\ join\ (A, B)$$

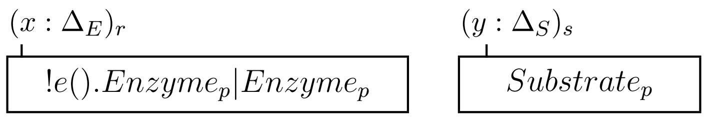
This template allows to translate chemical reactions into our language using only events; however, it is important to note that this is not the only way to do it, nor the more convenient. If it is necessary to model reversibility, affinity with more than one substrate, competitive inhibition, a model that uses the *bind* construct presented in Sect. 5 is more suitable. *Bind*, in fact, integrates in a natural way the notion of type affinity that allows us to write in a seamless way these kinds of reactions. Consider, for example, the introduction in the system of a fourth entity $D = \beta(x : \Delta_D)[\ nil\ ]$ that can react with B (e.g. Δ_B and Δ_D are *affine*). The user have to define not only D, but also to explicitly indicate its possible interactions writing new join(s) and split(s) events.

6.2.4 Enzymatic Reactions

Enzymes are proteins that catalyze chemical reactions. Molecules at the beginning of the process, called substrates S, bind with the enzymes E which convert them into different molecules, called products P. The bio-chemical representation of this kind of reactions is:

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

Almost all processes in the cell need enzymes in order to occur at significant rates. Since enzymes are extremely selective for their substrates and speed up only a few reactions from among many possibilities, the set of enzymes present in a cell determines which metabolic pathways occur in that cell. The β-system representing this model is $Z_{ER} = \langle B, \bullet, \xi \rangle$, where B is the parallel composition of the boxes:



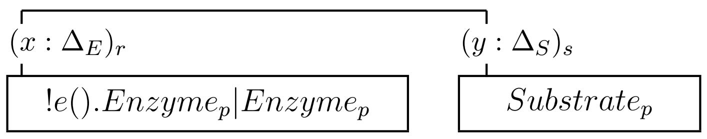
with:

$$\begin{aligned} Enzyme_p &= x().\overline{e}\langle\epsilon\rangle.\mathsf{nil} + M_e \\ Substrate_p &= \overline{y}\langle\epsilon\rangle.(\mathsf{ch}(y, \Delta_P), \infty).Product_p + M_s \end{aligned}$$

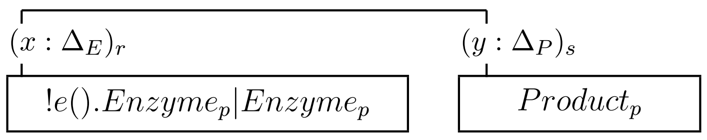
The bio-process on the left represents the *Enzyme*, while the other represents the *Substrate*. The structure of the pi-processes $Product_p$, M_e and M_s is not important for the purpose of the example.

Complementary shape of molecules domains are responsible for enzymes selectivity. In our model, domains are represented as elementary beta binders and their specificity is represented by the affinity between the types Δ_E and Δ_S. Hence, the affinity drives the ability of the *Enzyme* and of the *Substrate* to complex together.

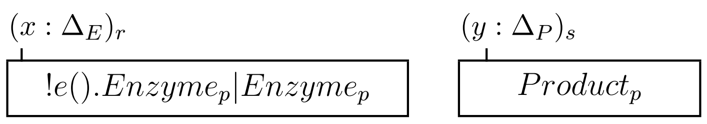
Now, suppose $\alpha(\Delta_E, \Delta_S) = (K_{ES}, K_{ES}^{-1}, K_{EP})$. The *Enzyme* and the *Substrate* can complex together with rate K_{ES}:



and consume an *inter-communication* through the created communication bound. After the communication, the *Substrate* consume immediately the action *ch* because its rate is infinite and the $Enzyme_p$ pi-process of the *Enzyme* is replicated. The resulting system is:



Now, assuming $\alpha(\Delta_E, \Delta_P) = K_P$, the two molecules can decomplex with rate K_P, producing the two boxes:



and obtaining a β-system in which the *Substrate* has been converted in *Product.*

To show how the compositionality of the language can be used to modify the system in a simple way, we consider the *competitive inhibition* mechanism. In this scenario there is an *Inhibitor* that can bind with the *Enzyme*. Therefore, the binding of the *Inhibitor* to the *Enzyme* prevents the binding of the *Substrate*. The bio-chemical representation of this reaction is:

$$EI + S \rightleftharpoons^{K_{EI}^{-1}}_{K_{EI}} I + E + S \rightleftharpoons^{K_{ES}}_{K_{ES}^{-1}} ES + I \rightharpoonup^{K_{EP}} EP + I \rightharpoonup^{K_{P}} E + P + I$$

The competitive inhibition mechanism can be introduced in our β-system of the enzymatic catalyzed reaction simply by composing in parallel a box representing the *Inhibitor*

$$\begin{array}{c}(z : \Delta_I)_i \\ \boxed{!\overline{z}(\epsilon).nil}\end{array}$$

If $\alpha(\Delta_I, \Delta_E) = (K_{EI}, K_{EI}^{-1}, t)$, then we have that the *Enzyme* can bind with the *Substrate* or with the *Inhibitor*. Since an elementary beta binder can be complexed with only one other elementary beta binder at a time, the resulting behaviour is exactly the one of the competitive inhibition. Our simulator produces the screen-shot reported in Fig. 25. The complete code of the model can be found in B.

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.

6.2.5 Multiple complexation

Consider the scenario in Fig. 18. There are molecules A, B, C and D that can create the complex $ABCD$ via the formation of intermediate complexes. The chemical equations that describe this model are the following:

$$\begin{aligned}
&A + B \rightleftharpoons AB\\
&B + C \rightleftharpoons BC\\
&C + D \rightleftharpoons CD\\
&AB + C \rightleftharpoons ABC\\
&A + BC \rightleftharpoons ABC\\
&BC + D \rightleftharpoons BCD\\
&B + CD \rightleftharpoons BCD\\
&A + BCD \rightleftharpoons ABCD\\
&ABC + D \rightleftharpoons ABCD\\
&AB + CD \rightleftharpoons ABCD
\end{aligned}$$

Note that we need to write 20 equations. In general, for n initial molecules the number of needed equations is $c * n^2$, where c is a constant value. For example, with 8 initial molecules we need to write 128 equations.

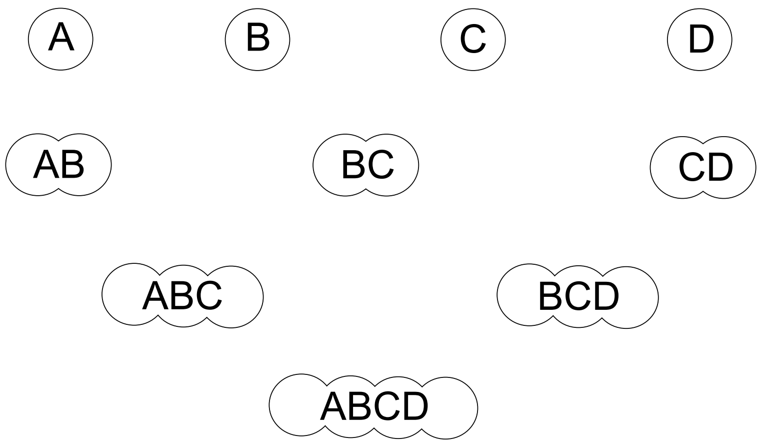
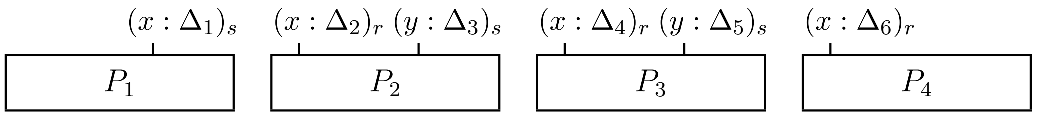


Figure 18: Complex formation via intermediate complexes

To model this scenario with the BetaSIM language, we only need to specify the initial molecules as boxes and manage all the possible intermediate complexes formation with complexation and decomplexation operations. This means that we only need to define in the proper way the affinities of the involved types. For example, the bio-process representing the structure of the scenario presented in Fig. 18 is:


$(x:\Delta_1)_s$
$(x:\Delta_2)_r$ $(y:\Delta_3)_s$
$(x:\Delta_4)_r$ $(y:\Delta_5)_s$
$(x:\Delta_6)_r$
P_1
P_2
P_3
P_4

with the affinity function defined in the following way:

$$\begin{aligned}\alpha(\Delta_1, \Delta_2) &= (r_1, s_1, 0)\\ \alpha(\Delta_3, \Delta_4) &= (r_2, s_2, 0)\\ \alpha(\Delta_5, \Delta_6) &= (r_3, s_3, 0)\end{aligned}$$

It is straightforward to see that if we consider the scenario with 8 initial molecules, we need only to compose in parallel 8 boxes and to define the affinity function on 7 couples of types. Hence, in this case the model size has linear growth.

6.2.6 Domain activation by complexation

Consider the scenario reported in Fig. 19. The molecule A can complex with the molecule B with rate r; the complexation generate a structural modification in the molecule B that activate a domain in B. The domain in B remains active until the molecule A is complexed with the molecule B. After the decomplexation of A and B, that happens with rate s, the active domain of the molecule B is deactivated with rate t.

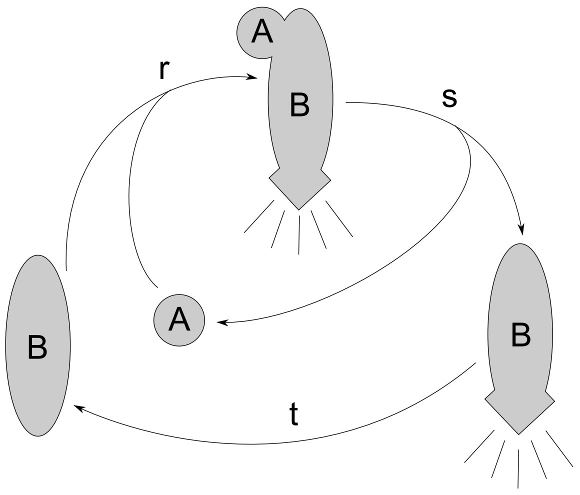
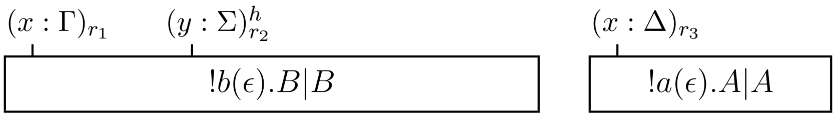


Figure 19: Indirect activation of an active site

The bio-process of the β-system that describes this scenario is the parallel composition of the boxes:



where:

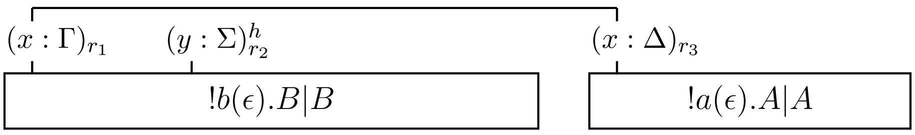
$$\begin{array}{rcl} A & = & \overline{x}\langle\epsilon\rangle.x(\epsilon).\overline{a}\langle\epsilon\rangle.nil \\ B & = & x(\epsilon).(\mathsf{unhide}(y),\infty).(\mathsf{ch}(x,\Delta_B),\infty).\overline{x}\langle\epsilon\rangle.(\mathsf{ch}(x,\Delta_U),s).C \\ C & = & (\tau,t).(\mathsf{hide}(y),\infty).(\mathsf{ch}(x,\Gamma),\infty).\overline{b}\langle\epsilon\rangle.nil \end{array}$$

and where the affinity function is defined in the following way:

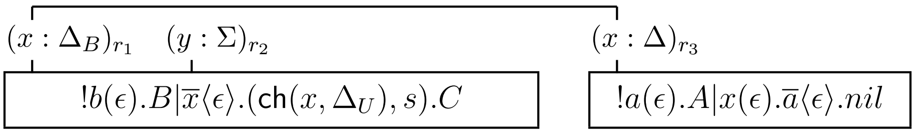
$$\begin{array}{l} \alpha(\Delta,\Gamma) = (r,0,\infty) \\ \alpha(\Delta,\Delta_B) = (0,0,s) \\ \alpha(\Delta,\Delta_U) = (0,\infty,0) \end{array}$$

The constructed model uses more than one action with infinite rate; however, no race condition between actions with infinite rate is generated. A detailed description of the evolution of the system is reported below.

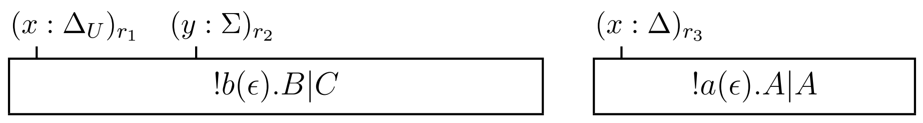
The first action enabled is the complexation, with rate r, of the two boxes between their unhidden elementary beta binders:



After the complexation a sequence of immediate actions in the molecule B is enabled; an inter-communication is immediatley consumed, the elementary beta binder with type Σ is unhidden and the type of the complexed beta binder is changed with one that can not generate a decomplexation operation:


$(x:\Delta_B)_{r_1}$
$(y:\Sigma)_{r_2}$
$!b(\epsilon).B|\overline{x}\langle\epsilon\rangle.(\mathsf{ch}(x,\Delta_U),s).C$
$(x:\Delta)_{r_3}$
$!a(\epsilon).A|x(\epsilon).\overline{a}\langle\epsilon\rangle.nil$

Now, the inter-communication is consummd with rate s and immmdiately the change action in the molecule B is executed and, because of the affinity of the types Δ_U and Δ, the decomplexation operation happens, producing the new configuration:



Now the tau action in C is executed with rate t and a new sequence of immediate actions is enabled; the elementary beta binder with type Σ is hidden, a change action is executed and the system returns to the initial configuration.

Note that we assume that the communications consumed on the channels a and b, used for internal replication of pi-processes, have infinite rate.

7 Implementation and Usage

The BetaSIM package includes three tools: a compiler/simulator (β-simulator), a graphical development editor (β-designer) and a plotting tool (β-plotter).

7.1 β-simulator

The β-simulator is a command-line application that takes two text files as input : the first file describes the model with a simple functional-like syntax, and the second one contains the list of types and the affinity between pairs of them. The β-simulator uses these two files to build the *environment* as described in Sect. 4; then the *Next Action Method* is used to drive the simulation until a final state is reached, the desired number of steps is performed or the time limit is over.

```
[time = 0.1]

<< BASERATE:inf >>

let A_p : pproc = (x{}.nil);

let A : bproc = #(x:1, DA) [ A_p ];

let B : bproc = #(x:1, DB)
[ x<e>.nil ];

run 100 A || 100 B
```

Figure 20: A simple β-binders program

```
{ DA, DB }
%%
{
(DA, DB, 1)
}
```

Figure 21: Its corresponding type file.

As an example, consider the very simple program in Fig. 20 and its matching type file in Fig. 21. Types were introduced in Sect. 3 as algebraic structures on which an equality relation is defined. In the actual implementation, types are defined by string identifiers, and their meaning is *user defined.* For example, if the language is used for describing chemical interactions, types will represent the interacting capabilities of valence shell electrons; in enzymatic reaction they will be catalytic sites; in cellular interactions the will be different kinds of receptors and so on. Placing types in a separate files allows the simulator to be agnostic with respect to the meaning of types; an external tool could be used to generate types and to compute affinities.

Types that will be used in a program are listed in the type file enclosed by {} before the double %%. After the separator, a second list of comma-separated 3-tuple or 5-tuple is used to define affinities. Therefore, the α function introduced in Sect. 3 is represented in the types file in a tabular form: if a line `(DA, DB, 1)` is present in the file, this means that $\alpha(D_A, D_B) = (0, 0, 1)$. The same holds for declarations with complexation and decomplexation rates like `(DC, DD, 1, 1.2, 1,1)`. The affinity function is meant to be reflexive, so the line `(DA, DB, 1)` stands for both $\alpha(D_A, D_B) = (0, 0, 1)$ and $\alpha(D_B, D_A) = (0, 0, 1)$. If a pair is not present in the affinity list, than the affinity between these two types is assumed to be zero.

The program starts with a line of *directives* for the simulator: it is in square brackets (line 1 in Fig. 20) and it instructs the simulator on the number of steps or on the simulation time to reach before stopping.

Next, the *rate declarations* are given. These are surrounded by double angular brackets (`<<`, `>>`) and can be thought to as the base rates for actions that do not have an explicit rate declaration (ρ function). It is possible to declare a global base rate (line 3, `BASERATE`) as well as global rates for a particular kind of actions (`CHANGE`, `HIDE`, `UNHIDE` and `EXPOSE`).

Declarations of π-processes ,β-processes, events and complexes follow the ones of rates. An example is given in lines 5–10: bio-processes are distinguished by the `bproc` keyword and by the binder declaration. Each binder is declared by a list of comma-separated pairs (subject:rate and type) starting with `#` for a regular binder and by `#h` for an hidden binder. Following the binder list, the π-process encapsulated in the bio-process must be specified (between square brackets, line 7 and 10). This π-process, as explained in Sect. 1, provide the internal structure and the hidden behaviour of the bio-process; it drives the structural conformation changes in response to external interactions with the bio-process. If this process is unknown, it is possible to specify the `nil` process.

Finally, the *run list* must be specified. The run list is used to specify the initial system that will be simulated. It is possible to declare, for each bio-process in the system, how many exemplars of that species will be present at the beginning of the simulation. If a bio-process is declared but not present in the run list it is assumed that in the initial instant it will have cardinality equal to zero; at least one bio-process must be present in the run list for the program to be valid.

Actions

Pi-processes in the β-binders languages are enriched with some actions for the manipulation of the interfaces, i.e. of the binders (see Sect. 3). So, the actions present in β-binders are:

input expressed as `x{}` without arguments and as `x{e}` in the form with the optional argument;

output expressed as `x<e>`, the argument is mandatory;

expose with rate *ar* associated to the action (`expose(ar, subject:1.2, TypeID)`) or without it, if a global base rate is specified (`expose(subject:1.2, TypeID)`;

hide with rate *ar* associated to the action (`hide(ar, subject)`) or without it, if a global base rate is specified (`hide(subject)`;

unhide with rate *ar* associated to the action (`unhide(ar, subject)`) or without it, if a global base rate is specified (`unhide(subject)`;

change with rate *ar* associated to the action (`change(ar, sbj, type)`) or without it, if a global base rate is specified (`change(sbj, type)`. The binder with subject *sbj* will have its type changed to *type*;

die with rate *ar* associated to the action (`die(ar)`) or without it, if a global base rate is specified (`die`);

tau with rate *ar* associated to the delay (`@(ar)`).

Summations and parallel composition

The sum ('+') and parallel (' | ') operators have the same syntax and semantics of the standard π-calculus, as defined in Sect. 3. Summations *must* be enclosed in square or round brackets:

```
let A_p : pproc = [ @(200.0).nil + x{}.nil ];
let B_p : pproc = ( x<e>.nil + y{}.nil );
```

Processes composed in parallel can be enclosed in round parenthesis to disambiguate their association (the ' | ' operator is right-associative):

```
x{}.(Prey_p | Prey_p) | Prey_p
```

Recursive definitions

The actual implementation of the β-binders language do not allow to directly write recursive definition of processes. For example, the following declarations:

```
let A_p : pproc = x{}.A_p;
let A : bproc = #(x:1, DA)
[ A_p ];

let E_p : pproc = [ @(200.0).nil + catalyse{}.E_p ];
let E : bproc = #(catalyse:1.0, DE)
[ E_p ];
```

are not accepted by the compiler. It is possible to rewrite these processes using the replication ('!', or *bang*) operator. β-binders permits to use *guarded replications*, i.e. to put the bang operator only before an action. Rewriting the first process is trivial: the `x{}` is repeated indefinitely:

```
let A_p : pproc = !x{}.nil;
let A : bproc = #(x:1, DA) [ A_p ];
```

Rewriting the second process is a more difficult task. We use two pi-processes with paired actions, i.e. one input and one output, on the same channel and either the input or the output has the '!' operator. Since the pi-subprocess following a guarded replication will be executed only when the action (e.g. an input) that *guards* the bang is consumed, the recursive invocation can be replaced by the complementary action (e.g. an output) on the same channel, followed by a `nil`:

```
let A_p : pproc = x{}.A_p;
```

can be translated to:

```
let A_p : pproc = x{}.r<e>.nil;
let A1_p : pproc = !r{}.A_p;
```

where the channel `r` will be used for the replication. The bio-process now will contain these two pi-processes in parallel. Intuitively, `A1_p` encloses the new instance of `A_p` that will be "released" by the old instance just before terminating.

```
let A : bproc = #(x:1, DA) [ A_p | A1_p ];
```

This can be done also using only one pi-process and writing directly the guarded replication inside the bio-process. So, our original code for `E` can be translated to:

```
let E_p : pproc = [ @(200.0).nil + catalyse{}.r<e>.nil ];
let E : bproc = #(catalyse:1.0, DE)
[ !r{}.E_p |  E_p ];
```

Events and complexes

Events are specified using the special keyword *when*. An example is given in the following fragment of code:

```
let P : pproc = x{}.r<e>.nil;
let B : bproc = #(x:1.0,Type) [ P ];
let B1 : bproc = #(x:1.0,Type) [ r<e>.nil ];

when (B1:r) split(B1,B1);
when (|B|=0) new(20);
```

Events are specified indicating in their *cond* and *verb* identifiers of bio-processes previously defined. Whereas admitted in the BetaSIM language, for semplicity here no bio-process can be defined inside the event, and hence a definition like:

```
when (#(x:1.0,Type) [ r<e>.nil ]:r) split(B1,B1);
```

is not admitted. Therefore, all the bio-processes used inside events have to be defined before the event definition. Events can be defined only on bio-processes without complexed binders. However, with the keyword *inherit*:

```
when inherit (B1:r) split(B1,B1);
```

we provide the possibility to an event to be inherited if the appropriate conditions hold (see Section 5 for details).

The simulator also provides the possibility to define and run complexes. A complex is specified as a graph, following the graph representation described in Sect. 5. For example, the first complex reported in Fig. 7 is specified in the following way:

```
let P1 : pproc = a{}.nil;
let P2 : pproc = b{}.nil;
let P3 : pproc = c{}.nil;
let P4 : pproc = d{}.nil;
let B1 : bproc = #(x:1.0,D0) [ P1 ];
let B2 : bproc = #(x:1.0,D0)#(y:1.0,D1) [ P2 ];
let B3 : bproc = #(x:1.0,D0)#(y:1.0,D1) [ P3 ];
let B3 : bproc = #(x:1.0,D0) [ P4 ];

let Complex : molecule  =
{
((Node1, x, Node2, x), (Node2, y, Node3, x), (Node3, y, Node4, x));
Node1 : B1 = (Box0;x);
Node2 : B2 = (Box1;x,y);
Node3 : B3 = (Box1;x,y);
Node4 : B4 = (Box1;x);
};
```

After the definition of the opportune pi-processes and bio-processes, the complex is defined using the keyword *molecule*. In particular, with:

```
Node1 : B1 = (Box0;x);
```

we define a node with name *Node1*, which represents an instance of a bio-process *B1* where the elementary beta binder with subject x is in complexed status. Instead with:

```
(Node2, y, Node3, x)
```

we define an edge between the nodes *Node1* and *Node2* on the elementary beta binders with subjects y and x. This edge represents a complexation between the bio-processes represented by the corresponding nodes, on the specified elementary beta binders.

The detailed description of the BNF grammar of the β-simulator is reported in Appendix A.

For a description of events and complexes, the reader should refer to Sections 3 and 5. For more information on particular constructs and for some examples we address the reader to the *Patterns* subsections of Sections 6, or to the examples included with the BetaSIM package.

Output files

Three files are generated as the output result of the simulation: one is a synthetic overview of the reactions executed, the species and complexes generated during the simulation and their internal structure; the other two are the trace of the variation of concentration of each specie or complex over time. The files have the same name of the input file, plus an additional suffix to distinguish them:

- `.spec` for the file with the description of reactions and species;

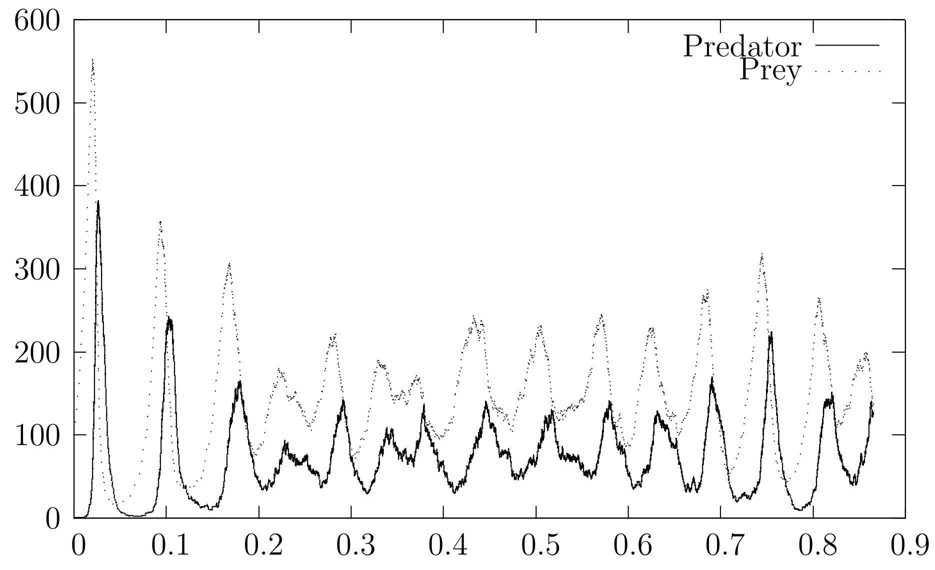


Figure 22: Charting Predator-Prey with gnuplot

- `.E.out` for the file with time variations of single entities;
- `.C.out` for the file with time variations of complexes.

The last two files lines of tabbed-separated columns; in each line, the first column is the execution time, the others are quantities of the various entities (or complexes). The order of the columns in this file is equal to the order of bio-processes definition in the `spec` file; for example, if *Predator* is the first bio-process described in the `spec` file, the first column after the time in the `E.out` file will refer to *Predator*. These two files can be easily plotted using an external program with charting functionality, such as gnuplot or Microsoft Excel.

Gnuplot

The simulation of *Predator-Prey* will led to three output files:

- `PredatorPrey.prog.spec`,
- `PredatorPrey.prog.E.out`,
- `PredatorPrey.prog.C.out`.

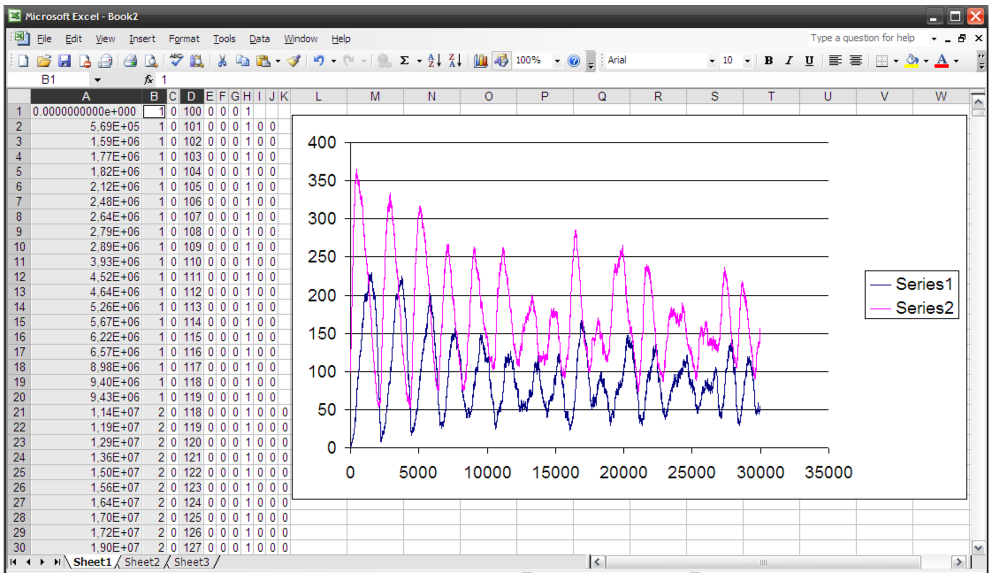
The last one will be empty, as no complexes are formed during the simulation of this program.

Examining the PredatorPrey.prog.spec file, we see that the processes we are interested in, *Predator* and *Prey*, are the first and the third respectively. To plot them with gnuplot, start the program and type the following commands at the prompt:

```
plot 'PredatorPrey.prog.E.out' using 1:2 title 'Predator' w l, \
     'PredatorPrey.prog.E.out' using 1:4 title 'Prey' w l
```

and press enter. The directive `using 1:2` means that we want to plot the time (first column) on the x axis and the first specie (second column) on the y axis. The result is shown in Figure 22.

Microsoft Excel



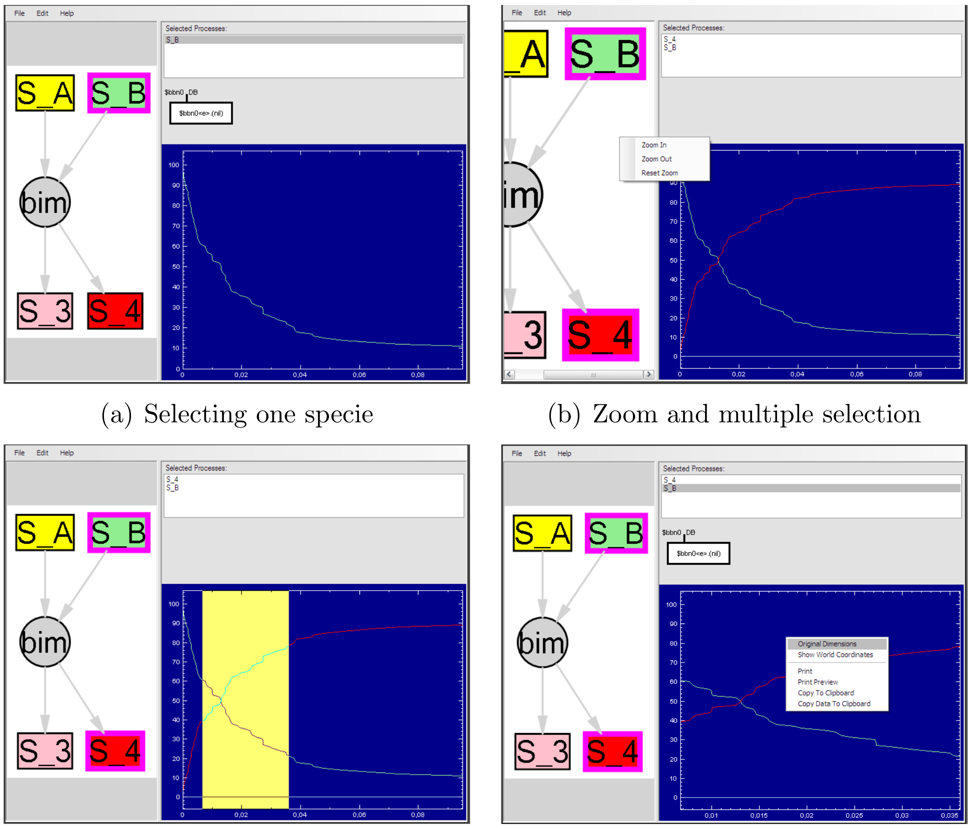
	A	B	C	D	E	F	G	H	I	J	K
1	0.0000000000e+000	1	0	100	0	0	0	1			
2	5,69E+05	1	0	101	0	0	0	1	0	0	
3	1,59E+06	1	0	102	0	0	0	1	0	0	
4	1,77E+06	1	0	103	0	0	0	1	0	0	
5	1,82E+06	1	0	104	0	0	0	1	0	0	
6	2,12E+06	1	0	105	0	0	0	1	0	0	
7	2,48E+06	1	0	106	0	0	0	1	0	0	
8	2,64E+06	1	0	107	0	0	0	1	0	0	
9	2,79E+06	1	0	108	0	0	0	1	0	0	
10	2,89E+06	1	0	109	0	0	0	1	0	0	
11	3,93E+06	1	0	110	0	0	0	1	0	0	
12	4,52E+06	1	0	111	0	0	0	1	0	0	
13	4,64E+06	1	0	112	0	0	0	1	0	0	
14	5,26E+06	1	0	113	0	0	0	1	0	0	
15	5,67E+06	1	0	114	0	0	0	1	0	0	
16	6,22E+06	1	0	115	0	0	0	1	0	0	
17	6,57E+06	1	0	116	0	0	0	1	0	0	
18	8,98E+06	1	0	117	0	0	0	1	0	0	
19	9,40E+06	1	0	118	0	0	0	1	0	0	
20	9,43E+06	1	0	119	0	0	0	1	0	0	
21	1,14E+07	2	0	118	0	0	0	1	0	0	0
22	1,19E+07	2	0	119	0	0	0	1	0	0	0
23	1,29E+07	2	0	120	0	0	0	1	0	0	0
24	1,36E+07	2	0	121	0	0	0	1	0	0	0
25	1,50E+07	2	0	122	0	0	0	1	0	0	0
26	1,56E+07	2	0	123	0	0	0	1	0	0	0
27	1,64E+07	2	0	124	0	0	0	1	0	0	0
28	1,70E+07	2	0	125	0	0	0	1	0	0	0
29	1,72E+07	2	0	126	0	0	0	1	0	0	0
30	1,90E+07	2	0	127	0	0	0	1	0	0	0

Figure 23: Charting Predator-Prey with Microsoft Excel

In Microsoft Excel, it is necessary to import the data in new Sheet. Open the "Data" menu, then choose "Import External Data" ("Get External Data" in some versions) and then click on the "Import Data..." item ("Import Text File" in some versions). In the dialog, change "Files of type:" to All Files and locate the PredatorPrey.prog.E.out file.

Accept all the default values of the wizard (column separator TAB, etc.), then select the columns you want to chart and use the Chart Wizard. It is better to choose a *XY (Scatter)* type of chart. Be sure, in the Series step of the wizard, that the first column is treated as x-axis values and not a separate Serie. The result is shown in Figure 23.

7.2 β-plotter



(a) Selecting one specie

(b) Zoom and multiple selection

(c) Zooming the plot

(d) Context menu

Figure 24: Using the plotter to view results.

β-plotter (Figures 24, 25, and 26) is a graphical tool that parses and display BetaSIM output files using a **graph view** and a **plot view**.

The **graph view** on the right panel visualises the reaction relations between entities (bio-processes or complexes): every grey circle is a reaction, incoming arcs are the reactants and outgoing arcs are the products. Possible reactions are *bim*, *bind*, *unbind*, *join* and *split*, for inter-communication, complexation and decomplexation and events. Entities are shown in colored square boxes, complexes in diamond shaped boxes. Both are clickable, so that it is possible to select them. The selected entities and complexes are added to the list of selected objects (upper right side of the window in Figure 24(a)). It is possible to click on objects in this list to examine their internal structure. This is particularly useful for complexes, as the bounds they form and the structures arisen during the simulation can be fairly complex. The complex

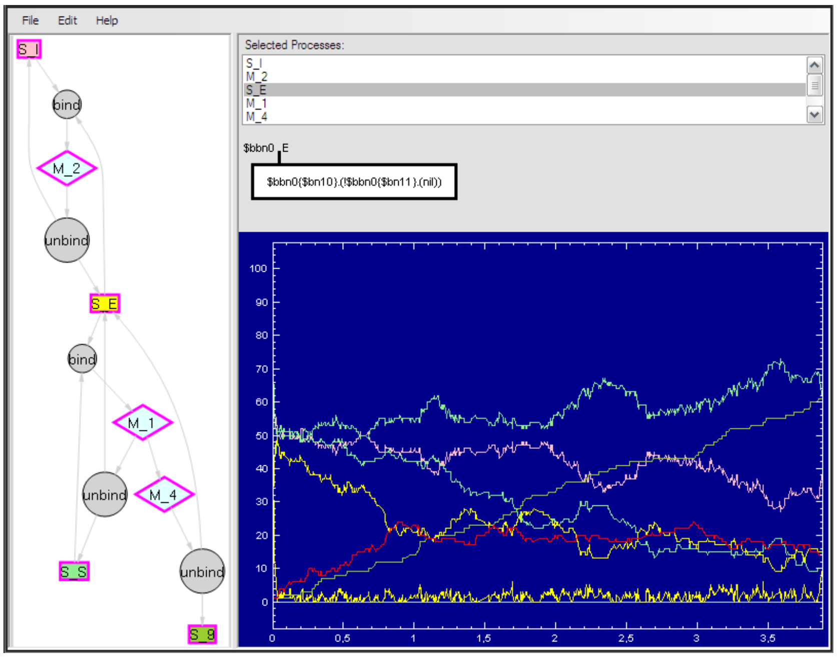


Figure 25: The result of Enzymatic Catalysed Reaction simulation in β-plotter

structure is displayed as in β-designer (Figure 30). Multiple selection is possible using ctrl+Mouse-Click; the graph is also zoom-able using the mouse and the context menu, that appears when the right mouse button is pressed in a white area (Figure 24(b)). By default, only visible (i.e. non immediate) reactions are displayed in the graph view; an option to view all the reactions is available in the *Edit* menu.

The **plot view** displays the change in concentration of entities and complexes over time. When one or more entities are selected in the graph view, the plot displays only their concentration variation. The plot is zoom-able: click and hold the left button of the mouse on a point at the beginning of the area to zoom, then drag the mouse to a point at the end of the area and release it (Figure 24(c)). To reset the zoom and for other functions related to the plot view, a click with the right mouse button on the blue area will bring up a popup menu.

Functions to print or to save on an image file both the graph and the plot are available in the file menu.

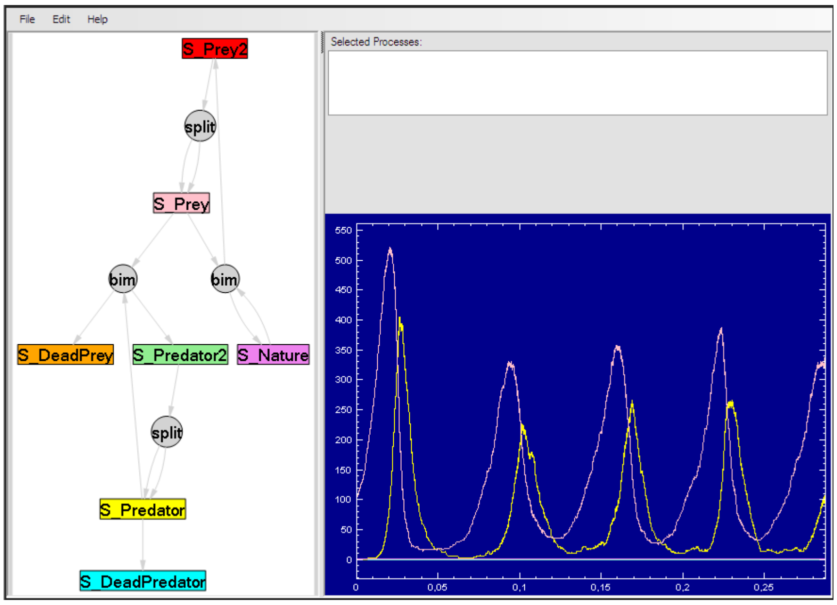


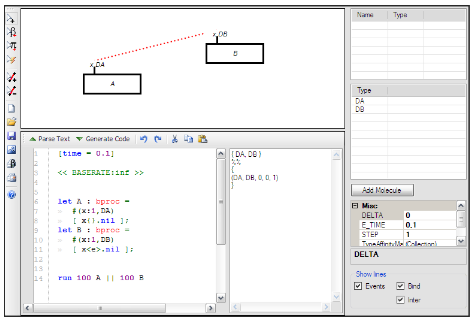
Figure 26: The result of Predator-Prey simulation in β-plotter

7.3 β-designer

β-designer is a graphical tool that allows to write β-binder programs in a graphical way. In particular, it is possible to draw boxes, pi-processes, interactions, events and to form complexes (also called *molecules*) using graphs. The graphical format and the textual description of the program are interchangeable: the tool can parse and generate the graphical representation from any valid β-binder program, and generate the textual representation from the graphical form. The textual representation can then be used as input of the BetaSIM simulator.

On the left side of the β-designer main windows there is the **Main Toolbar** from which it is possible to perform the most common operations. On the right side of Figure 29, 27 it is possible to view the **Properties panel**. The **Draw area** and the **Text editor** for the code are located in the central part, above and below another toolbar, the **Edit toolbar**.
In a separate window, a **Molecule editor** can be used to construct complexes made of more two or more β-boxes bound together, using the graph formalism introduced in Section 5.

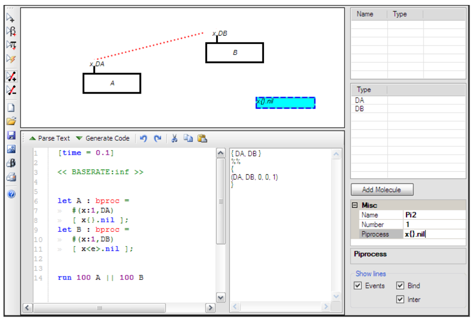
The **Main Toolbar** is used to perform operations on the program files (create a new program/model, open an existing one, save it, print the text program or the graphical model, etc) and to perform operations on the graph-



```
[time = 0.1]

<< BASERATE:inf >>


let A : bproc =
    #(x:1,DA)
    [ x{}.nil ];
let B : bproc =
    #(x:1,DB)
    [ x<e>.nil ];


run 100 A || 100 B
```

```
{ DA, DB }
%%
{
(DA, DB, 0, 0, 1)
}
```

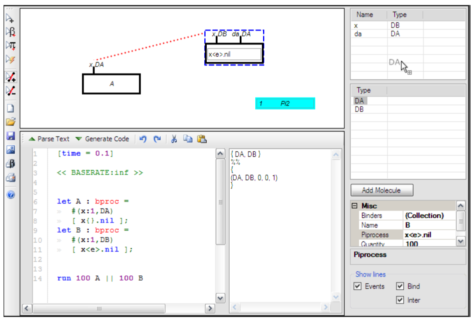
(a) Generating the graphical schema



```
[time = 0.1]

<< BASERATE:inf >>


let A : bproc =
    #(x:1,DA)
    [ x{}.nil ];
let B : bproc =
    #(x:1,DB)
    [ x<e>.nil ];


run 100 A || 100 B
```

```
{ DA, DB }
%%
{
(DA, DB, 0, 0, 1)
}
```

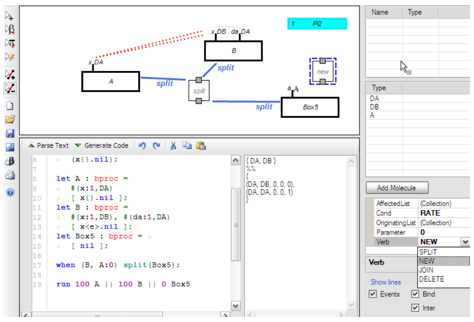
(b) Adding processes



```
[time = 0.1]

<< BASERATE:inf >>


let A : bproc =
    #(x:1,DA)
    [ x{}.nil ];
let B : bproc =
    #(x:1,DB)
    [ x<e>.nil ];


run 100 A || 100 B
```

```
{ DA, DB }
%%
{
(DA, DB, 0, 0, 1)
}
```

(c) Adding a binder



```
    (x{}.nil);

let A : bproc =
    #(x:1,DA)
    [ x{}.nil ];
let B : bproc =
    #(x:1,DB), #(da:1,DA)
    [ x<e>.nil ];
let Box5 : bproc =
    [ nil ];

when (B, A:0) split(Box5);

run 100 A || 100 B || 0 Box5
```

```
{ DA, DB }
%%
{
(DA, DB, 0, 0, 0),
(DA, DA, 0, 0, 1)
}
```

(d) Event description

Figure 27: Using the plotter to view results.

ical model. To add boxes, events, pi-processes or interactions the user need to choice an appropriate tool from this toolbar and use it on the *Draw area*. For example, to add a box the user have to select the 'Add Box' tool (second icon, see Figure 27(b)) and the click on a blank part of the Draw area. To add an interaction, select the 'Add Link' tool, click on an *hot spot* of first object involved in the interaction and then on an *hot spot* of the second one.

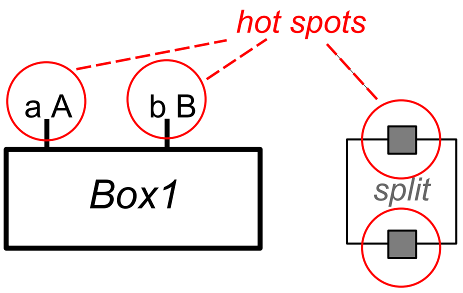
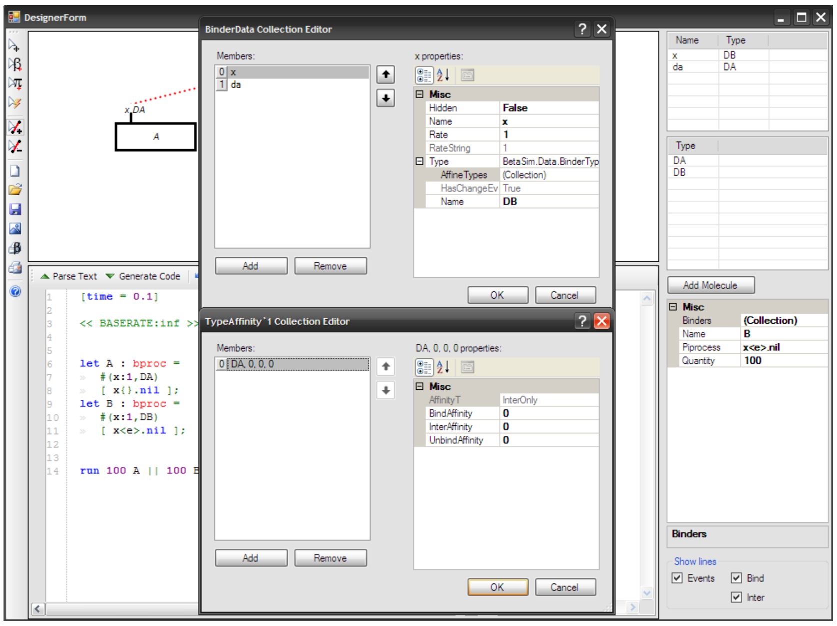


Figure 28: *Hot spots* on boxes and events.

An *hot spot* is an element in a graphical object that has interaction capabilities. These include, for example, binders on β-boxes and enter and exit point on events (see Figure 28).



```
[time = 0.1]

<< BASERATE:inf >>


let A : bproc =
    #(x:1,DA)
    [ x{}.nil ];
let B : bproc =
    #(x:1,DB)
    [ x<e>.nil ];


run 100 A || 100 B
```

BinderData Collection Editor — Members: 0 x, 1 da

x properties:

Misc	
Hidden	False
Name	x
Rate	1
RateString	1
Type	BetaSim.Data.BinderTyp
AffineTypes	(Collection)
HasChangeEv	True
Name	DB

TypeAffinity`1 Collection Editor — Members: 0 DA, 0, 0, 0

DA, 0, 0, 0 properties:

Misc	
AffinityT	InterOnly
BindAffinity	0
InterAffinity	0
UnbindAffinity	0

Name	Type
x	DB
da	DA

Type
DA
DB

Misc	
Binders	(Collection)
Name	B
Piprocess	x<e>.nil
Quantity	100

Show lines: ✓ Events ✓ Bind ✓ Inter

Figure 29: Setting binding properties and affinity

The **Edit toolbar** is located between the Text editor and the Draw area. It contains command buttons related to the Text editor, and two buttons to switch between the textual and graphical representation. The model is always saved and loaded in textual format, so once the program text it is loaded into the Text editor to obtain the correspondent graphical representation the *Parse* button must be clicked. The converse action (re-create an update program text from the graphical representation) can be done using the *Generate* button. It is important to stress that the model is saved in textual format, so every change made to the graphical representation must be reflected to the textual program by means of the *Generate* button before saving the model, or the changes will be lost.

The **Text editor** is like every other text editor in programming environments, providing syntax highlight, brackets matching, undo and redo operations, cut and paste and so on. Immediately on the right of the program editor there is a simpler editor for the type file.

The **Draw area** is were the graphical form of the model is drawn; it is possible to modify, move and delete the objects that compose the model, select them and insert new objects and interactions using the Main toolbar. As it is possible to see in the figures, interactions are drawn using lines. Lines in light azure represent the possibility of a *complexation* interaction; lines in red of an *inter-action*; lines in blue are used for *events*.

The **Properties panel** maintains a list of information about the system and the selected object. In the upper part of this panel two lists, the *binders list* and the *type list* indicate respectively the binders (subject and type) present on the selected β-box and the types available in the system. It is possible to insert new types using the right mouse button, and new binders dragging a type from the second list to the first one (see Figure 27(c)). Below the two lists, a *property grid* displays the properties of the currently selected item. If nothing is selected, the properties of the ambient are shown. Properties are editable; for example in Figure 27(d) the verb for the currently selected event is being changed. The *property grid* can contain properties, such as list of binders, that can be expanded in a new windows and examined in further details (see Figure 29 for an example).

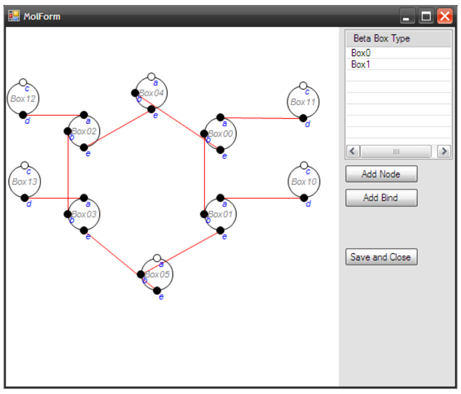
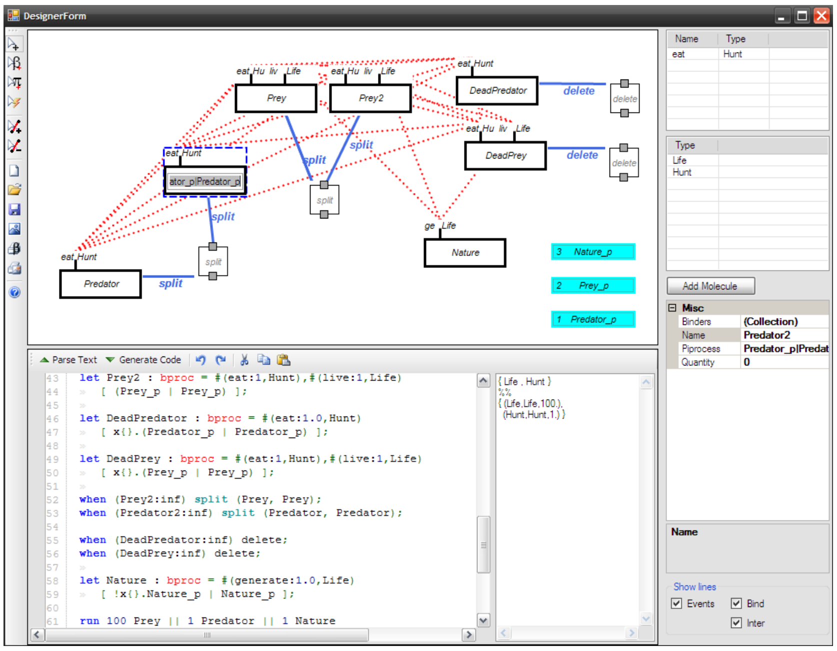


Figure 30: Viewing a complex in β-designer and β-plotter

Finally, in the lowest portion of the Properties panel, there is the possibility to filter out some or all the lines rendered in the Draw area.

The **Molecule editor**, shown in Figure 30, can be displayed using the *Add Molecule* button in the Properties panel. The usage of the Molecule editor is very similar to the usage of the Draw area: using the small toolbar in the upper left corner it is possible to add a box instance, remove it, add a bound or remove it. Boxes instances (or *nodes*) are rendered as circles, surrounded by smaller circles representing binders, as in Sections 5 and 3. Black binders are *occupied* i.e. already bound, while white binders are *free*. The available node types, i.e. the bio-process defined in the system, are listed on the right. To add a new node that is an instance of a bio-process A, click on A on the list, than select the 'Add Node' tool (first in the toolbar) and click on an empty area. To add a bound, select the third tool ('Add binder'), click on a free binder and than on another free binder in a different node.



```
let Prey2 : bproc = #(eat:1,Hunt),#(live:1,Life)
   [ (Prey_p | Prey_p) ];

let DeadPredator : bproc = #(eat:1.0,Hunt)
   [ x{}.(Predator_p | Predator_p) ];

let DeadPrey : bproc = #(eat:1,Hunt),#(live:1,Life)
   [ x{}.(Prey_p | Prey_p) ];

when (Prey2:inf) split (Prey, Prey);
when (Predator2:inf) split (Predator, Predator);

when (DeadPredator:inf) delete;
when (DeadPrey:inf) delete;

let Nature : bproc = #(generate:1.0,Life)
   [ !x{}.Nature_p | Nature_p ];

run 100 Prey || 1 Predator || 1 Nature
```

```
{ Life , Hunt }
%%
{ (Life,Life,100.),
  (Hunt,Hunt,1.) }
```

Name	Type
eat	Hunt

Type
Life
Hunt

Misc	
Binders	(Collection)
Name	Predator2
Piprocess	Predator_p|Predat
Quantity	0

Name

[✓] Events [✓] Bind [✓] Inter

Figure 31: The Predator-Prey example in β-designer

8 Conclusion

In this paper we presented BetaSIM , a stochastic simulator for biological processes. Biological processes are modelled using β-binders with bindings, complexes and event-driven join and split functions. We have developed a new algorithm to perform efficient simulation of β-binders programs, that is used to implement a fast and accurate simulator. Two companion tools make considerably easier to write models and analyse the results of the simulation. Both the tools and language are actively developed to introduce new constructs and enhancements. The complete BetaSIM package is freely available for download for academic and research users.

9 Acknowledgements

We would like to thank Davide Prandi for fruitful discussions on both theoretical and implementive aspects. Moreover, we would like to thank Maria Luisa Guerriero and Alida Palmisano for useful suggestions about usage and debugging of the tools.

References

[1] A. Regev and E. Shapiro. Cells as computation. *Nature*, 2002.

[2] H. Kitano, editor. *Foundations of Systems B iology.* The MIT Press, 2001.

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

[4] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.

[5] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[6] C. Priami, A. Regev, E. Shapiro, and W. Silvermann. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett.*, 80(1):25–31, 2001.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] V. Danos and C. Laneve. *Graphs for Core Molecular B iology*, volume 2602 of *Lecture Notes in Computer Sciences*, pages 34–46. Springer, 2003.

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

[15] Maria Luisa Guerriero, Davide Prandi, Corrado Priami, and Paola Quaglia. Process calculi abstractions for biology. Technical Report TR-13-2006, The Microsoft Research - University of Trento Centre for Computational and Systems Biology, 2006.

[16] C. Priami and P. Quaglia. Beta binders for biological interactions. In *CMSB*, pages 20–33, 2004.

[17] C. Priami and P. Quaglia. Operational patterns in beta-binders. *T. Comp. Sys. Biology*, 1:50–65, 2005.

[18] C. Priami and A. Romanel. The decidability of the structural congruence for beta-binders. In *MeCBIC 2006*, 2006.

[19] Thomas H. Cormen, E. Leiserson, Charles, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. COR th 01:1 1.Ex.

[20] M.A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem.*, 104:1876–1889, 2000.

[21] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: its structural complexity*. Birkhäuser, 1993.

[22] V. Volterra. *Variazioni e Fluttuazioni del Numero d'Individui in Specie Animali Conviventi*, pages 1–110. Memorie dela R. Academia Nationale dei Lincei. 1926.

[23] A.P. Somlyo and A.V. Somlyo. Signal transduction and regulation in smooth muscle. *Nature*, 372(6503):231236, 1994.

[24] K.E. Kamm and l J.T. Stul. Regulation of smooth muscle contractile elements by second messengers. *Annu Rev Physiol.*, 51:299313, 1989.

[25] M.R. Lee, L. Li, and T. Kitazawa. Cyclic GMP causes Ca2+ desensitization in vascular smooth muscle by activating the myosin light chain phosphatase. *J Biol Chem.*, 272(8):50635068, 1997.

[26] R. M. Rapoport, M.B. Draznin, and F. Murad. Endothelium-dependent relaxation in rat aorta may be mediated through cyclic GMP-dependent protein phosphorylation. *Nature*, 306(5939):174–176, 1983.

[27] N. Marsh and A. Marsh. A short history of nitroglycerine and nitric oxide in pharmacology and physiology. *Clin Exp Pharmacol Physiol*, 27(4):313–319, 2000.

[28] J.D. Corbin and S.H. Francis. Cyclic GMP phosphodiesterase-5: target of sildenafil. *J Biol Chem.*, 274(20):13729–32, 1999.

[29] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular biology of the cell.* Garland Science, 2002.

A BNF Grammar

Program file BNF

```
⟨ program ⟩       ::=  ⟨ info ⟩ << ⟨ global_rates ⟩ >>
                       ⟨ declarations ⟩ run ⟨ system ⟩

⟨ info ⟩          ::=  [ steps = Decimal ]
                   |   [ steps = Decimal, delta = Real ]
                   |   [ time = Real ]

⟨ rate ⟩          ::=  Real
                   |   Decimal
                   |   inf

⟨ global_rates ⟩  ::=  Name : ⟨ rate ⟩
                   |   EXPOSE : ⟨ rate ⟩
                   |   HIDE : ⟨ rate ⟩
                   |   UNHIDE : ⟨ rate ⟩
                   |   CHANGE : ⟨ rate ⟩
                   |   BASERATE : ⟨ rate ⟩
                   |   ⟨ global_rates ⟩ , ⟨ global_rates ⟩

⟨ dec_list ⟩      ::=  declaration
                   |   declaration dec_list

⟨ declaration ⟩   ::=  let Id : pproc = ⟨ piprocess ⟩;
                   |   let Id : bproc = ⟨ betaprocess ⟩;
                   |   let Id : molecule = ⟨ molecule ⟩;
                   |   when ( ⟨ cond ⟩ ) ⟨ verb ⟩;
                   |   when inherit ( ⟨ cond ⟩ ) ⟨ verb ⟩;

⟨ seq ⟩           ::=  ⟨ action ⟩ . ⟨ piprocess ⟩
                   |   ⟨ action ⟩

⟨ piprocess ⟩     ::=  nil
                   |   Id
                   |   ⟨ seq ⟩
                   |   [ ⟨ sum_list ⟩ ]
                   |   ( ⟨ sum_list ⟩ )
                   |   ! ⟨ action ⟩ . ⟨ piprocess ⟩
                   |   ⟨ piprocess ⟩ | ⟨ piprocess ⟩
                   |   ( ⟨ piprocess ⟩ )
```

```
⟨ sum_list ⟩          ::=  ⟨ seq ⟩ + ⟨ seq ⟩
                       |   ⟨ sum_list ⟩ + ⟨ seq ⟩

⟨ action ⟩            ::=  Name < Name >
                       |   Name { Name }
                       |   Name { }
                       |   expose ( Name : ⟨ rate ⟩ , Id )
                       |   expose ( ⟨ rate ⟩ , Name : ⟨ rate ⟩ , Id )
                       |   hide ( Name )
                       |   hide ( ⟨ rate ⟩ , Name )
                       |   unhide ( Name )
                       |   unhide ( ⟨ rate ⟩ , Name )
                       |   ch ( Name , Id )
                       |   ch ( ⟨ rate ⟩ , Name , Id )
                       |   die
                       |   die ( ⟨ rate ⟩ )
                       |   @( ⟨ rate ⟩ )

⟨ cond ⟩              ::=  Id : ⟨ rate ⟩
                       |   | Id | = Decimal
                       |   Id , Id : ⟨ rate ⟩

⟨ verb ⟩              ::=  delete
                       |   join
                       |   join( Id )
                       |   split( Id , Id )
                       |   new( Id , Decimal )

⟨ binder_list ⟩       ::=  # ( Name : ⟨ rate ⟩ , Id )
                       |   #h ( Name : ⟨ rate ⟩ , Id )
                       |   ⟨ binder_list ⟩,⟨ binder_list ⟩

⟨ betaprocess ⟩       ::=  ⟨ binder_list ⟩ [ ⟨ piprocess ⟩ ]

⟨ molecule ⟩          ::=  { ( ⟨ edge_list ⟩ ) ; ⟨ node_list ⟩ }

⟨ edge_list ⟩         ::=  ( NodeId , Name , NodeId , Name )
                       |   ( NodeId , Name , NodeId , Name ) , ⟨ edge_list ⟩

⟨ node_list ⟩         ::=  ( NodeId , Id , ⟨ mol_binder_list ⟩ )
                       |   ( NodeId , Id , ⟨ mol_binder_list ⟩ ) , ⟨ node_list ⟩

⟨ mol_binder_list ⟩   ::=  Name ;
                       |   Name ⟨ mol_binder_list ⟩
```

⟨ system ⟩ ::= *Decimal Id*
| ⟨ system ⟩ || ⟨ system ⟩

Type File BNF

⟨ type_file ⟩ ::= { ⟨ type_list ⟩ }
| { ⟨ type_list ⟩ } %% { ⟨ affinity_list ⟩ }

⟨ type_list ⟩ ::= *Id*
| ⟨ type_list ⟩ , *Id*

⟨ affinity_list ⟩ ::= (*Id* , *Id* , ⟨ rate ⟩)
| (*Id* , *Id* , ⟨ rate ⟩ , ⟨ rate ⟩ , ⟨ rate ⟩)
| ⟨ affinity_list ⟩ , (*Id* , *Id* , ⟨ rate ⟩)
| ⟨ affinity_list ⟩ , (*Id* , *Id* , ⟨ rate ⟩ , ⟨ rate ⟩ , ⟨ rate ⟩)

Regular Expressions

DLetter [a-z]
ULetter [A-Z]
Digit [0-9]
Exp [Ee][+\-]?{Digit}+

Id ({DLetter} | {ULetter} | _)({ULetter} | {DLetter} | {Digit} | _)*
NodeId ({DLetter} | {ULetter} | _)({ULetter} | {DLetter} | {Digit} | _)*
Name ({DLetter} | {ULetter} | _)({ULetter} | {DLetter} | {Digit} | _)*

Decimal {Digit}+
real1 {Digit}+{Exp}
real2 {Digit}*”.”{Digit}+({Exp})?
real3 {Digit}+”.”{Digit}*({Exp})?
Real {real1} | {real2} | {real3}

B Examples code

NO-cGMP pathway

Program file

```
/////////////////////////////////////////////////////
// NO-cGMP pathway
/////////////////////////////////////////////////////
// Nitric Oxide
let NO_p : pproc = act{e}.r<e>.nil;
let NO : bproc = #(act : 1.0; NOd)
   [ !r{}.NO_p | NO_p ];
// Nitric oxide synthease, the enzime that produces NO
let NOS_p : pproc = act<e>.@(2.2).r<e>.nil;
let NOS : bproc = #(act : 1.0; coLAd)
   [ !r{}.NOS_p | NOS_p ];
// Its substrate, L-arginine
let L_ARG_p : pproc = act{e}.r<e>.nil;
let L_ARG : bproc = #(act : 1.0; LAd)
    [ !r{}.L_ARG_p | L_ARG_p ];
// The active form of NOS, after interaction with L-arginine
// It stays active for a tau
let ActiveNOS : bproc = #(act : 1.0; coLAd)
   [ !r{}.NOS_p | @(2.2).r<e>.nil ];
// When NOS is active, produce some NO
when (ActiveNOS: 2.2) split (ActiveNOS, NO);
// Binding to NOS on coLAd, L-NMMA prevents activation
let L_NMMA : bproc = #(x : 1.0; LNAd)
   [ nil ];
// Another way to introduce NO: Nitro (Glycerin-tri-nitrate)
let GTN : bproc = #h(no : 1.0; NOd), #(gdn : 1.0; GDNd)
   [ NO_p ];
let GDN : bproc = #(gdn : 1.0; GDNd)
   [ NO_p ];
// the process of production of NO out of GTN is unknown,
// let's model it with a simple slipt with rate 2.2
//when (GTN: 2.2) split (GDN, NO);
// The guanylyl cyclase produces cGMP when activated by NO
let GC_p : pproc = act<e>.p{}.r<e>.nil;
let GC : bproc = #(act : 1.0; Heme), #(p : 1.0; pGTP)
  [ !r{}.GC_p | GC_p ];
// Methylene blue is a competitiva inhibitor of GC
let MBlue : bproc = #(x : 1.0; coHeme1)
      [ nil ];
// This bioprocess encodes the behaviour of both GTP and cGMP
let GMP_p : pproc = (p<e>.unhide(act).unhide(d).act{}.r<e>.nil +
                     d<e>.nil);
let cGMP : bproc = #(p: 1.0; PhosphorG), #h(d : 1.0; DegradeG),
                   #h(act : 1.0; GmpDomain)
   [ !r{}.GMP_p | GMP_p ];
// The degraded cGMP process, will be deleted
let cGMP_d : bproc = #(p : 1.0; PhosphorG), #(d : 1.0; DegradeG),
                     #(act : 1.0; GmpDomain)
   [ !r{}.GMP_p ];
when (cGMP_d: inf) delete;
// We suppose to have an unlimited amount of GTP
//when (/cGMP/ = 0) new(cGMP, 1);
// After binding on gmpDomain, send degrade message, then detach
// PDE-5 degrades cGMP
let PDE5_p : pproc = d{}.ch(d, gmpRelease).ch(d, coGmpDomain).r<e>.nil;
let PDE5 : bproc = #(d : 1.0; coGmpDomain)
   [ !r{}.PDE5_p | PDE5_p ];
// Sildenafil (or E-4021) binds to gmpDomain, but do not react to
// the degrade message
let Sildenafil : bproc = #(x : 1.0; similGmpDomain)
   [ nil ];
```

Type file

```
{
ActivateNOS,
ActiveNOS,
NOd,
GDNd,
NOSActivation,
Heme,
coHeme1,
pGTP,
PhosphorG,
DegradeG,
GmpDomain,
PKGDomain,
coGmpDomain,
similGmpDomain,
gmpRelease,
coChannel,
CACDomain,
coCalm,
CalmDomain,
pATP,
ACActivation,
BetaActivator,
BReceptorD,
EphiD,
BlockerD,
CRLRActivator,
CLReceptorD,
AMd,
PhosphorA,
DegradeA,
AmpDomain,
ActivationDomain,
DeactivationDomain,
coMLCK
}
%%
{
   ( NOSActivation, ActivateNOS, 1.0, 1.0, 0.0 ),
   ( Heme, NOd, 1.0, 1.0, 1.0 ),
   ( Heme, coHeme1, 1.0, 1.0, 1.0 ),

   ( pGTP, PhosphorG, 1.0 ),
   ( GmpDomain, PKGDomain, 1.0 ),
   ( coChannel, CACDomain, 1.0 ),
   ( coCalm, CalmDomain, 1.0, 1.0, 1.0 ),
   ( ActivationDomain, coMLCK, 1.0 ),

   // no unbinding (or very little)
   ( DegradeG, coGmpDomain, 1.0, 0.0, 1.0 ),
   // immediate unbinding
   ( DegradeG, gmpRelease, 0.0, inf, 0.0 ),

   ( similGmpDomain, coGmpDomain, 1.0, 1.0, 0.0 ),
   ( DeactivationDomain, AmpDomain, 1.0 ),
   ( PhosphorA, pATP, 1.0  ),
   ( ACActivation, BetaActivator, 1.0 ),
   ( ACActivation, CRLRActivator, 1.0 ),

   ( EphiD, BReceptorD, 1.0, 1.0, 1.0 ),
   ( BlockerD, BReceptorD, 1.0, 1.0, 0.0 ),

   ( CLReceptorD, AMd, 1.0, 1.0, 1.0 ),
   ( ActivateNOS, AMd, 1.0, 1.0, 1.0 )
}
```

Lotka Volterra

Program file

```
[steps = 10000]

<< BASERATE:inf >>

let Predator_p : pproc =
    [@(200.0).nil + eat{}.x<e>.nil];
let Prey_p : pproc =
    [live<e>.x<e>.nil + eat<e>.nil];
let Nature_p : pproc =  generate{}.x<e>.nil;

let Predator : bproc = #(eat:1.0,Hunt)
[ x{}.(Predator_p | Predator_p) |  Predator_p ];

let Predator2 : bproc = #(eat:1.0,Hunt)
[ Predator_p | Predator_p ];

let Prey : bproc = #(eat:1,Hunt),#(live:1,Life)
[ x{}.(Prey_p | Prey_p) | Prey_p ];

let Prey2 : bproc = #(eat:1,Hunt),#(live:1,Life)
[ (Prey_p | Prey_p) ];

let DeadPredator : bproc = #(eat:1.0,Hunt)
[ x{}.(Predator_p | Predator_p) ];

let DeadPrey : bproc = #(eat:1,Hunt),#(live:1,Life)
[ x{}.(Prey_p | Prey_p) ];

when (Prey2:inf) split (Prey, Prey);
when (Predator2:inf) split (Predator, Predator);

when (DeadPredator:inf) delete;
when (DeadPrey:inf) delete;

let Nature : bproc = #(generate:1.0,Life)
[ !x{}.Nature_p | Nature_p ];

run 100 Prey || 1 Predator || 1 Nature
```

Type file

```
%%
{
  (Life,Life,100.),
  (Hunt,Hunt,1.)
}
```

Enzyme Inhibition

Program file

```
[steps = 1000]

<< BASERATE:inf,
    CHANGE:inf>>
```

```
/* Definition of the enzime, the substrate
   and the inhibitor */
let E : bproc = #(x:1,E)[ !x{}.nil ];
let S : bproc = #(x:1,S)[ x<e>.ch(x,P).nil ];
let I : bproc = #(x:1,I)[ !x<e>.nil ];

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

Type file

```
{ S,P,E,I }
%%
{
 (S,E,1.,1.,1.),
 (E,I,1.,1.,0.),
 (P,E,0.,1.,0.)
}
```