MTCoord 2005 Preliminary Version

Biomolecular agents as multi-behavioural concurrent objects

Denys Duchier [1] Céline Kuttler [2]

Interdisciplinary Research Institute (CNRS), Lille, France

Abstract

In recent years, there has been increasing interest in computational models of biological systems based on various calculi of communicating processes, such as the stochastic pi-calculus. These models make it possible to simulate and eventually visualize the dynamic evolution of complex biosystems in time and under varying environmental conditions.

While the elegance of the pi-calculus lies in its minimality, this is also a drawback when it comes to modeling because much effort must be devoted to encoding high-level ideas into the low-level means that the language affords us.

In this paper, we describe an on-going effort to design a new higher-level programming language that provides direct ontological support for the concepts which are used to formulate, organize and structure models of biomolecular systems.

Our language has an object-oriented flavour where we view molecular components as agents with finite sets of behaviours (states). Reactions are modeled as exchanges over connected ports that may cause agents to switch states.

Key words: biological systems, stochastic pi-calculus, communicating agents, systems biology, simulation, programming

1 Introduction

The interdisciplinary study of coordination [12] investigates processes of managing dependencies among activities. On the surface, researchers from different fields use the same terminology for the description of complex systems: *model, control, regulation, amplification, feedback.*

Each living cell is a inherently complex systems [6]. It is densely populated by ten thousands molecules with highly specialized function and interaction capabilities. For example, a receptor molecule on a bacterium's surface may be

[1] Email: `denys.duchier@lifl.fr`

[2] Email: `celine.kuttler@lifl.fr`

responsible for sensing the availability of food in its environment. It would let the inside of the cell know that, for instance, its favourite sugar is available. The cell would produce a range of substances, some for opening entrance points for the sugar on the cell membrane. Other groups of molecules would be responsible for the exploitation of the sugar, i.e. its decomposition into smaller parts that deliver the energy needed by the cell to maintain its vital functions.

All cellular processes involve interaction between a possibly large number of its constituents: all molecules have specific functions, they can do virtually nothing in isolation, but need to work with other molecules in a well controlled and coordinated manner. Vital functions enabled by properly coordinated biomolecular actors include cell growth, replication, reactions to manyfold events in the outside world, etc. Many diseases are associated with a failure of intra-cellular interaction or signalling networks, e.g. cancer or neurodegenerative diseases. As control and regulation in biology [4,22] are implicit, it is not possible to identify a control unit as a separate process as in the classical fields of control theory.

System biology investigates the behavior and relationships of all the elements in a particular biological system while it is functioning [7]. It aims at formulating mathematical models that reflect biological knowledge. These can be used to address relevant questions to biologists: in a model one can systematically perturb individual components, monitor the system as a whole and observe its response to the perturbation. By shifting the emphasis to the description of the dynamic interactions underlying cellular functions, it considerably differs from traditional biology: this deals with the identification of the cell's components, their molecular characterization and cataloging.

Pi and systems biology: The stochastic π-calculus [15] is a refinement of Milner's π-calculus [13], designed for abstracting concurrent nondeterministic processes. Its operational semantics is subject to stochastic control. Regev and Shapiro proposed to apply the stochastic π-calculus for simulations in systems biology in 2002 [20,21]. Since then it has been applied in a number of case studies [8,10,11]. These models were formulated directly in stochastic π, and executed in the BioSpi [17] or similar systems [14]. The available models can be used to simulate biological systems, and allow for the tests of hypotheses.

Albeit that the stochastic π calculus allows to reflect the inherent concurrency, communication and stochasticity of cellular systems, each new case study reveals drawbacks of biological modeling directly in π. The origin of this is to be found in the calculus' minimality, which makes it cumbersome to hand-craft models with complex coordination and dependencies. In order to become accessible to a wider group of modelers, it seems desirable to increase the ease of modeling by extending the language used. Suggestions have been made both from the side of enriching the variants of π used, see beta binders [16], adapting the ambient calculus with the aim to more directly

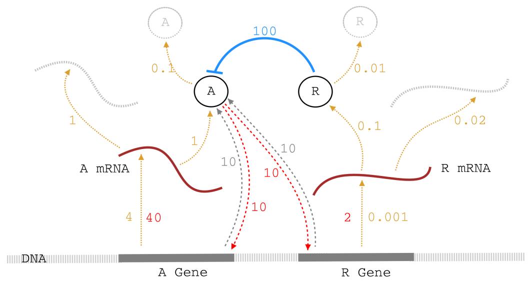

Fig. 1. Model of core circadian oscillator (as suggested by [1])

support specific aspects as compartmentalization ([19]), or designing de novo calculi for interaction *on* membranes [2], and protein-protein interaction [5]. Another early suggestion was to aim at an higher-level programming language for biological applications based on π, and down-compilable to it [3].

Outline: Section 2 introduces an essential model of circadian clocks, as an example of coordinated orchestration of different components of a cell. Section 3 illustrates infelicities in the use of the π-calculus as a modeling language. We then outline the principles of our language in Sec. 4. As a use case, in Sec. 5 we apply our language for modeling circadian machinery. Section 6 summarizes and gives an outlook on the current work-in-progress of further developing and implementing our language.

2 Coordination in Circadian Clocks

As an illustration of dynamic and well-orchestrated processes that take place inside a cell, we give a description of a class of mechanisms that has been actively investigated in recent years, and is being worked on by modelers with various backgrounds.

Circadian clocks can be found in virtually all organisms. First examples are already present in some bacteria [9], and an abundance more is found among eukaryotes - these are higher organisms, starting with yeast, and including insects, plants, and animals. Circadian clocks control night-and-day rhythms that are of overwhelming importance to nearly all life on earth that has evolved in environments with cyclic environmental changes.

As a cartoonish example for a daily rhythm, one could identify two genes of an organism, *sleep* and *activity*. These unfold their activities at night and day respectively. At the cellular level, one could measure oscillations between the genes' products in rhythms of 24 hours, with time-shifted phases. The basic biomolecular mechanisms controlling circadian clocks in a wide range of organisms share common features. They consist in two interleaved feedback loops: an *activator* reinforces first its own expression, in a positive loop. At the same time, the activator enhances the production of a *repressor* element. A sufficiently high accumulation of this repressor causes the activator's production to turn off. After some time, all activator molecules have been degraded -

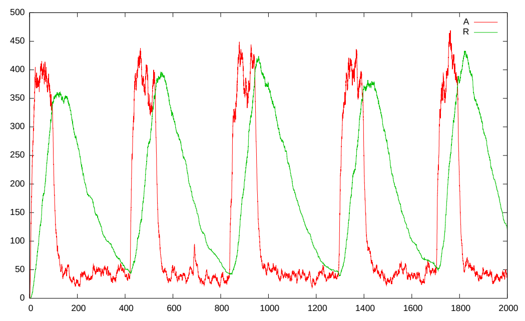

Fig. 2. Evolution of A and R protein levels over time for the core circadian oscillator. Simulation plot based on the execution of the stochastic π model from [17].

and when no more activators are available, the production of repressors ceases as well. When the repressor level has decreased by natural degradation, the activator first returns to its basal activity level, and by doing so it somewhat later starts reinforcing its own production and that of the repressor.

Figure 1 illustrates an essential model of a core circadian machinery, as proposed by Barkai and Leibler [1]. The activator A can bind to two promoters on DNA (promoters can be seen as button that activate genes), its own and that of the repressor. This leads to enhanced *transcription* of static information encoded in the genes into portable mRNA. Without activator, transcription occurs only at a lower basal level. The mRNA molecules are subject to two competing processes, *translation* into the proteins A and R, and *degradation*. The R protein can inactivate A by binding to it, as stated previously A can reversibly bind to the promoters, and both proteins are subject to degradation. All reactions are quantified by rates or speeds given on the edge labels.

Regev and co-authors have applied the stochastic π calculus to this example. Figure 2 plots the evolution of protein levels over time, as it arises when executing their model. The red and green line show absolute numbers of proteins A and R, respectively, over time given in simulated seconds on the x axis. The systems oscillates with periods of roughly 400 time units. Starting with no proteins at all in time 0, a high level of A protein rapidly accumulates. Note that the steepness of the red curve is due to A's ability to promote its own production at rate 40, whereas R's accumulation proceeds slower – both because it depends on accumulation of A, and is only promoted to a lesser degree by it (promoted rate of 2). Now we observe the inhibitive effect that R exerts on A. R's negative effect on A is well observable after a high level of R has accumulated. The complexation between A and R is by far the fastest reaction in the system, and makes A unavailable for further promotion of either gene. A undergoes effective degradation in the $A \cdot B$ complex, explaining the rapid decline of the red curve. In turn, the R pool empties more slowly, until we get to a level where both A and R are rare. At this point, the story repeats - A re-initiates both its own promotion and somewhat later that of R.

Fig. 2 also gives us a first impression the inherent *variability* of biomolecular systems. Qualitatively, the oscillatory behaviour is the same for each cycle. But still quantitative fluctuations are visible in the curves, as can best be seen in the high plateaus. The detailed evolution of protein numbers clearly fluctuates from one cycle to the next.

3 Modeling biosystems in the π-calculus

We are interested in elaborating stochastic models of complex biosystems that can be executed to run simulations. One type of approach is based on stochastic variants of the π-calculus where biomolecular agents are modeled as concurrent *processes* that may communicate over *channels* and where reactions are identified with communication events. While the π-calculus has proven to be a flexible and adequate semantic foundation for capturing models of biosystems, its very minimality forces the modeler to encode high-level interaction patterns into the low-level means offered by the calculus. This encoding makes models harder to write and understand, and obscures the original high-level biological model with a plethora of irrelevant computational details.

In this section, we take a critical look at the π-calculus as a modeling language for biosystems. We show how severe infelicities arise already for very simple examples. We identify the asymmetry of the send/receive communication model as one culprit for these infelicities and propose an alternative model based on simultaneous exchanges.

3.1 Complex formation in the π-calculus

Consider the reaction of complex formation involving molecules of types A and B, that occurs at rate k:

$$A + B \longrightarrow^{k} A \cdot B$$

To model this reaction in the stochastic π-calculus, we introduce a global channel `bindAB` with an appropriate stochastic rate. Molecules of types A and B may be in either of two states: free and bound. Only a molecule in its free state can participate in the reaction of complex formation. Each state is modeled by a corresponding process definition:

```
A_free   ::=   bindAB ! [] , A_bound.
B_free   ::=   bindAB ? [] , B_bound.
```

Notice how the π-calculus induces an irrelevant asymmetry as it requires one process to send on reaction channel `bindAB` and the other to receive. A consequence of this fact is that the models of molecules A and B cannot be derived from the same common base model, event hough they clearly share the same behaviour. This problem becomes even more obvious in the case of a heterodimerization reaction $A + A \longrightarrow A \cdot A$, in which two molecules of the same

```
A_free   ::=   degradeA ? [] , 0
             + new freeMe , new return ,
               bindAB ! {freeMe,return} , return ? {freeB} ,
               A_bound(freeMe,freeB) .

B_free   ::=   degradeB ? [] , 0
             + new freeMe,
               bindAB ? {freeA,return}  , return ! {freeMe},
               B_bound(freeMe,freeA).

A_bound(freeMe,freeHim)
         ::=   degradeA ? [] , freeHim ! [] , 0
             + freeMe   ? [] , A_free .

B_bound(freeMe,freeHim)
         ::=   degradeB ? [] , freeHim ! [] , 0
             + freeMe   ? [] , B_free .
```

Fig. 3. Complex formation with spontaneous degradation

type assemble.

3.2 Complex formation and degradation

Consider again the reaction of complex formation, but now suppose that molecules of types A and B can also undergo spontaneous degradation in both their free and bound states:

$$A \longrightarrow \qquad B \longrightarrow \qquad A \cdot B \longrightarrow B \qquad A \cdot B \longrightarrow B$$

The added complexity here is that when an A molecule degrades while in the bound state, it must somehow inform its companion B molecule that the latter must return to its free state. Since molecules are modeled by π-calculus processes which do not have manipulable identities, the only option for two processes to *know* each other is for them to exchange *private* channels.

In the model of Figure 3, whenever two molecules A and B bind to form a complex $A \cdot B$, they each create a private `freeMe` channel and hand it over to their partner: when one of them degrades, it signals that fact to the other by sending on the other's `freeMe` channel. The protocol for exchanging these new private channels is itself complicated by the directionality of communication: the A process much create a private `return` channel and send it to B along with A's `freeMe` channel to allow the B process to send back its own `freeMe` channel.

3.3 Replacing send/receive by simultaneous exchange

We have shown that directionality of communication in the π-calculus has a number of undesirable consequences:

- it introduces an irrelevant asymmetry in otherwise similar molecular behaviours

- which means that the respective models for two reactants with similar behaviours cannot be derived from a common base model
- furthermore, exchanging information between reactants requires extra communication over a new private channel

We can eliminate considerations of directionality by abandoning the send/receive model in favour of one based on simultaneous exchange. We will base the formulation of communication rules on the following syntax:

port `bind` [$v_1, \ldots, v_n$] `->` [$x_1, \ldots, x_k$] `...`

which represents an exchange on port `bind` whereby values $v_1, \ldots, v_n$ are sent and simultaneously bindings for $x_1, \ldots, x_k$ are received. We now explain the notion of *port*.

In models based on the π-calculus, reactants were distinguished by the fact that one had to send and the other to receive. If we remove this distinction based on directionality of communication, how can we tell them apart, and, more to the point, how can we prevent $A + A \longrightarrow A \cdot A$ from happening by accident. In order to distinguish reactants, we will no longer model a reaction by a single channel, but by a pair of *ports*: one port for each reactant.

4 Introduction to the new language

In order to overcome the limitations of the π-calculus as a modeling language, we are designing a new higher-level programming language whose primary purpose is to provide direct ontological support for the concepts which are used to organize and structure our models. It is also important to state explicit non-goals: it is not our intention to devise a new calculus, nor to design a general purpose programming language. We are solely focussed on making it easier to write and understand models by providing suitable linguistic support in the modeling language itself. To a large extent, our endeavor can be regarded as providing an improved syntactic layer over the stochastic π-calculus.

As a deliberate design choice, and in contrast with the π-calculus, we choose to view agents as concurrent objects with persistent identities, and to organize their specifications into a hierarchy of classes in the object-oriented tradition. Past experience has shown that it is natural and convenient to describe biomolecular agents as finite state machines. For this reason, our notion of class directly supports the view of agents as finite state machines: a *class* is a collection of state definitions.

In the traditional object-oriented perspective, the behaviour of an object is unique and fixed by the class. In our proposal, each state defines a separate behavior bundle and an agent can adopt different behaviors at different times. A state contains definitions of communication rules: each rule corresponds to one of the reactions in which the biomolecular agent can participate in this state.

```
class Base {
   constructor init [] {} become <free>;
   state <free> {
      port bind [self] -> [other] {}
         become <bound>[other];
      port degrade [] -> [] {}
         become dead;
   }
   state <bound>[other] {
      port degrade [] -> [] {}
         become other.<free>;
         become dead;
   }
}
```

Fig. 4. Common base model for A and B

Figure 4 illustrates how we can write a common base model for biomolecular agents of types A and B. This model has two states `<free>` and `<bound>`. Let's take a look at the communication rule for port `bind` in state `<free>`:

```
port bind [self] -> [other] {}
   become <bound>[other];
```

Upon binding, the agent in `<free>` state sends its own identity to its partner while simultaneously receiving the latter's identity in variable `other`. It then enters the `<bound>` state, with `other` as a state parameter. Now let's take a look at the degradation rule in the `<bound>` state:

```
port degrade [] -> [] {}
   become other.<free>;
   become dead;
```

Statement **become** `other.<free>` causes the partner to return to its own `<free>` state, while **become dead** causes this agent to die and disappear. We now show how the common `Base` model can be specialized to derive models for both A and B. First, we define the required reactions:

```
reaction ComplexAB <a,b> [AB_COMPLEXATION_RATE];
reaction DegradeA  <a>   [A_DEGRADATION_RATE];
reaction DegradeB  <b>   [B_DEGRADATION_RATE];
```

Features given between angle brackets are arbitrary and simply allow us to conveniently access the corresponding ports for each reactant. For example, we will use `ComplexAB.a` as the port for A and `ComplexAB.b` as the port for B. A Reaction with two features is bi-molecular, while one with a single feature is mono-molecular.[3] Models for A and B are obtained by deriving from `Base` and *connecting* port names to port values:

```
class A : Base {
   port bind = ComplexAB.a;
```

[3] Note that in pi, modeling monomolecular reactions requires the introduction of artificial communication counterparts

```
   port degrade = DegradeA.a; }

class B : Base {
   port bind = ComplexAB.b;
   port degrade = DegradeB.b; }
```

4.1 Classes and multiple inheritance

In this section, we briefly and informally outline the intended semantics of multiple inheritance and the restrictions which must be respected. In our proposal, a class consists of (1) a set of state definitions and (2) a possibly partial set of connections associating port names with port values. Consider:

```
class B : A1,A2 { ... }
```

If both `A1` and `A2` provide a connection for the same port name, they must agree on this connection, i.e. connect it to the same port value.

`B`'s set of states is the union of the states defined in `A1`, `A2`, and in the body of `B`'s definition. Similarly, for each state, the set of rules is inherited from corresponding state definitions in `A1` and `A2`, and extended and/or overridden in `B`. If, for a given state, distinct rules for the same port name are inherited from `A1` and `A2`, then an overriding rule must be provided in `B` for that state.

Only classes where all port names have been connected to port values can be instantiated.

4.2 Bound states

The `Base` model of Figure 4 requires agents to exchange their respective identities. This technique has the unfortunate consequence that it requires each object to have intimate knowledge of the other's interface. For example, the statement **become** `other.<free>` requires the agent to know that the `other` has a `<free>` state and that this state is indeed the one to which it must be returned upon dissociation. What if we wanted to produce a refinement of `B` which upon dissociation should instead enter a `<widowed>` state in which complex formation is no longer possible?

We can easily address this issue by borrowing a leaf from Python's book. [4] Python has the notion of a *bound method* whereby evaluating $O.m$ returns a closure which, when applied to arguments $(x_1, ..., x_n)$, executes $O.m(x_1, ..., x_n)$. By analogy, we propose the notion of a *bound state* whereby evaluating $O.<s>$ returns a closure which, when passed as argument to **become**, executes **become** $O.<s>$.

Figure 5 illustrates the application of this idea to the common `Base` model for A and B. Now, the reactants exchange the bound states to which they must be returned upon dissociation.

[4] see www.python.org

```
class Base {
   constructor init [] {} become <free>;
   state <free> {
      port bind [<free>] -> [free_other] {}
         become <bound>[free_other];
      port degrade [] -> [] {}
         become dead;
   }
   state <bound>[free_other] {
      port degrade [] -> [] {}
         become free_other;
         become dead;
   }
}
```

Fig. 5. Revised common base model for A and B

4.3 Transitions between configurations

So far, our illustrative examples had only fairly simple rules that merely effected straightforward simultaneous state transitions of the objects directly involved in the reaction, i.e. of just the reactants. A more complex model, such the biosynthesis of glycogen (polymer of glucose), requires for each glucose element in the polymer to keep track of its distance to closest leaf, root, and/or branch node (see [21] for a model in stochastic π). Thus each reaction of cleaving, branching and elongation affects not only the two glucose elements directly involved in the reaction, but also requires updating the corresponding chains of glucose elements. For this reason, the general form of a rule is:

```
port p [$e_1, \ldots, e_n$] -> [$x_1, \ldots, x_k$]
   $PreTransitions$
   { $UpdateAlgorithm$ }
   $PostTransitions$
```

where *UpdateAlgorithm* typically invokes methods[5] of other biomolecular agents. Since concurrent systems with state are notoriously difficult to reason about, we wish the *UpdateAlgorithm* to execute atomically with respect to any state transition in the system as a whole. For this reason, for each reaction event, we allow *PreTransitions* which all take place before execution of the *UpdateAlgorithm*s of the reactants begins, and *PostTransitions* which all take place after the algorithms have finished and the system has reached computational quiescence.

What we allow the *UpdateAlgorithm* to do, is to queue state transition requests by executing statements of the form:

eventually become *BoundState*;

These queued requests are executed together with the *PostTransitions*.

[5] In addition to rules, state definitions may also contain method definitions. Different states may provide different methods, or different implementations of the same methods.

4.4 Translation into the stochastic π-calculus

While space limitations preclude a full presentation of the language's semantics, we now sketch how the first example of Section 4 can be translated into the stochastic π-calculus. The result is intended to be similar to the code of Fig 3, but also allow for *late binding* and for atomicity of execution of reaction behaviours with respect to state transitions, as described in 4.3

For simplicity, we omit a treatment of inheritance and assume that the text of class `Base` has been merged into derived classes `A` and `B` as illustrated in Figure 6.

```
class A {
   constructor init [] {} become <free>;
   state <free> {
      port bind [self] -> [other] {}
         become <bound>[other];
      port degrade [] -> [] {}
         become dead;
   }
   state <bound>[other] {
      port degrade [] -> [] {}
         become other.<free>;
         become dead;
   }
   port bind = ComplexAB.a;
   port degrade = DegradeA.a;
}
```

Fig. 6. Expanded definition of derived class *A*

Mono-molecular reactions are transformed into bi-molecular reactions by introducing timers as described by Regev in her thesis [18]. Reactions are implemented as global channels with finite stochastic rates:

```
global(complexAB(AB_COMPLEXATION_RATE),
       degradeA(A_DEGRADATION_RATE),
       degradeB(B_DEGRADATION_RATE)).
```

All other channels have infinite rates. By inspection (see Figure 6), we discover that the `bind` port name of `A` is connected to the first port value of reaction `ComplexAB` while `B` uses the second port value. Consequently, `A` will implement the *send/receive* part of the exchange protocol, while `B` will implement the corresponding *receive/send*.

Global channels are introduced to name states and methods and thus permit the implementation of *late binding*:

```
global(state_free,state_bound,state_dead).
```

Also, we need 3 additional global channels for a synchronization mechanism that delays state transitions until quiescence is reached as explained in 4.3:

```
global(sync,block,unblock).
```

Synchronization is mediated by a `Sync` agent: in its `free` state, it always accepts `sync` signals. When a reaction takes place, the reactant on the *send* side puts `Sync` into its `blocked2` state. Upon completion of the reaction behaviour, both reactants must `unblock` `Sync` to cause it to return to its `free` state again. While, in this article, we have tried to shun biospi-specific syntax, we retain one convenient notation, namely `self`, which simply reinvokes the current process definition with the same arguments:

```
Sync_free     ::= sync?[],self + block?[],Sync_blocked2.
Sync_blocked2 ::= unblock?[],Sync_blocked1.
Sync_blocked1 ::= unblock?[],Sync_free.
```

The translation of an agent's class results in one process definition per state and one process definition for the constructor. The latter creates private channels to represent the states and methods bound to this particular agent instance and starts the initial state process passing these channels as arguments. Private channel `me` is used to resolve late binding, i.e. to map global names of methods or states to corresponding private channels of the agent:

```
A ::= new me, new become_free, new become_bound, new become_dead,
       A_free(me,become_free,become_bound,become_dead).
A_free(       me,become_free,become_bound,become_dead) ::= ...
A_bound(other,me,become_free,become_bound,become_dead) ::= ...
```

Both `A_free` and `A_bound` are implemented as sums, and, in particular, share the handling of late binding. They both (textually) start with an alternative, listening on the `me` channel, that performs late binding resolution of a global name `state`, returning the result on channel `ret`. Note the use of a *test* sum of the form $Test_1, P_1 + \cdots + Test_n, P_n$:

```
        me?{state,ret},
        ((state==state_free ),ret!{become_free },self
        +(state==state_bound),ret!{become_bound},self
        +(state==state_dead ),ret!{become_dead },self)
```

This is immediately followed by alternatives listening on the private channels that effect state transitions:

```
     +  become_free?[],
        A_free(me,become_free,become_bound,become_dead)
     +  become_bound?{other},
        A_bound(other,me,become_free,become_bound,become_dead)
     +  become_dead?[],0
```

The rest of each state definition is dedicated to the reactions in which it can participate. For example, here is how `A_free` handles the `ComplexAB` reaction. This is the translation of the port rule for `bind` in state `<free>`. By inspection of class `A` (see Figure 6), we know that port name `bind` is connected to the first port value of reaction `ComplexAB`; therefore the rule will use a translation that implements the *send/receive* part of the exchange protocol on global channel `complexAB`:

```
     +  new ret,complexAB!{me,ret},block![],ret?{other},
        new done,
        ((new ret,me!{state_bound,ret},ret?{how},
          done![],sync![],how!{other},0)
        |(done?[],unblock![])
        |self)
```

A private channel `ret` is created to implement the return part of the exchange, `A_free` sends its identity (in the form of its late-binding handler `me`) and the `ret` channel; it then blocks state transitions and receives the identity from the `other`. At that point the exchange is complete and the reaction behaviour (*update* algorithm) is executed: the `A` agent should enter its bound state. This is is realized with a 3 part concurrent composition: the 3rd part (`self`) is essential to avoid deadlock: the agent must immediately become ready again to process late-binding resolutions and method invocations; the 1st part queries the `me` channel to resolve `state_bound` and obtains `become_bound` as the value of `how`;[6] at that point, it indicates that it is done (on channel `done`), waits for state transitions to be unblocked (on channel `sync`), and eventually performs the state transition by invoking `how!{other}`. The 2nd part waits until all concurrent operations are `done`, then signals with `unblock` to the `Sync` process that its part of the update algorithm has reached quiescence.

Correspondingly, again by inspection, we determine that port `bind` in class `B` is connected to the 2nd port value of reaction `ComplexAB` and therefore must implement the *receive/send* part of the exchange protocol. Thus, the prefix in the translation of the same rule needs to be the mirror image of the one for `A` above:

```
  +  complexAB?{other,ret},ret!{me},
     new done,
     ((new ret,me!{state_bound,ret},ret?{how},
       done![],sync![],how!{other},0)
     |(done?[],unblock![])
     |self)
```

The way `degradeA` is handled by `A_bound(other)` illustrates a double late-binding resolution and a corresponding double synchronization before unblocking:

```
  +  new ret,degradeA!{ret},block![],ret?[],
     new done1,new done2,
     ((new ret,other!{state_free,ret},ret?{how},
       done1![],sync![],how![],0)
     |(new ret,me!{state_dead,ret},ret?{how},
       done2![],sync![],how![],0)
     |(done1?[],done2?[],unblock![])
     |self)
```

The rest of the translation proceeds similarly. Clearly, translating our language into the stochastic π-calculus is rather technical and becomes increasingly less attractive as further high-level features have to be considered. Furthermore the overhead of this encoding leads to programs that do not perform

[6] conceivably, static analysis could optimize this particular lookup away

well in practice in BioSpi. For these reasons, we are working on formulating a direct operational semantics, and our implementation effort no longer targets the stochastic π-calculus but the high-level multiparadigm language Mozart/Oz.[7]

5 Case study: the circadian clock

In this section, we show how the biological phenomenon of the circadian clock presented in Section 2 can be modeled in our language. Previously this has been done directly in stochastic π [17].

First, Figure 7 contains the base model of a `Gene` which can be transcribed either at a slow rate, or, when promoted, at a fast rate:

```
class Gene
{
  constructor init[] {} become <slow>;
  method transcribe[] {}
  state <slow>
  {
    port slow { transcribe[]; }
    port promote[<slow>] become <fast> {}
  }
  state <fast>
  {
    port fast { transcribe[]; }
  }
}
```

Fig. 7. Common base class for genes

```
reaction A_promotion          <gene,protein> [RATE_A_PROMOTE];
reaction A_transcription_slow <gene>         [RATE_A_TRANSCRIBE_SLOW];
reaction A_transcription_fast <gene>         [RATE_A_TRANSCRIBE_FAST];

class A_Gene : Gene
{
  port slow    = A_transcription_slow.gene;
  port fast    = A_transcription_fast.gene;
  port promote = A_promotion.gene;
  method transcribe[] { new A_RNA.init[]; }
}
```

Fig. 8. Gene coding for protein A

Models for the genes coding for proteins A and R are obtained by inheritance and specialization from the common base class `Gene`. Figure 8 shows how to derive the model of the gene coding for protein A. First, we define the three reactions of promotion, slow transcription, and fast transcription when promoted, then we connect them to the corresponding port names inherited from

[7] see www.mozart-oz.org

the base model. Finally, a specialized definition of the `transcribe` method is provided which produces the version of mRNA specific to the A gene.

The model for the R gene is obtained in a similar fashion, as illustrated in Figure 9.

```
reaction R_promotion          <gene,protein> [RATE_R_PROMOTE];
reaction R_transcription_slow <gene>         [RATE_R_TRANSCRIBE_SLOW];
reaction R_transcription_fast <gene>         [RATE_R_TRANSCRIBE_FAST];

class R_Gene : Gene
{
  port slow    = R_transcription_slow.gene;
  port fast    = R_transcription_fast.gene;
  port promote = R_promotion.gene;
  method transcribe[] { new R_RNA.init[]; }
}
```

Fig. 9. Gene coding for protein R

Transcription of a gene on the DNA produces a corresponding RNA. Figure 10 contains the base model of `RNA` which can either be translated or may spontaneously degrade.

```
class RNA
{
  constructor init[] {} become <init>;
  method create[] {}
  state <init>
  {
    port translate { create[]; }
    port degrade become dead {}
  }
}
```

Fig. 10. Common base class for RNA

Models for the RNA that results from the transcription of A and R genes are obtained by inheritance and specialization from the common base class `RNA`, as shown in Figure 11.

The A protein may either be free, or it may be bound to the promoter of the A gene on the DNA, or to the promoter of the R gene on the DNA, or it can be bound to a R protein to form a complex. When the A protein is bound to a promoter site, it may fall off: this is modeled by the corresponding reaction which we dub *demotion* to emphasize that it reverses an earlier *promotion*. When A forms a complex with R, it is only subject to spontaneous degradation.

The R protein is simpler: it may either be free or form a complex with A. In either case it may spontaneously degrade. Figure 13 is basically just an instance of the model first presented in Figure 5.

```
reaction A_translation     <rna> [A_TRANSLATION_RATE];
reaction A_RNA_degradation <rna> [A_RNA_DEGRADATION_RATE];

class A_RNA : RNA
{
  port translate = A_translation.rna;
  port degrade = A_RNA_degradation.rna;
  method create[] { new A.init[]; }
}

reaction R_translation     <rna> [R_TRANSLATION_RATE];
reaction R_RNA_degradation <rna> [R_RNA_DEGRADATION_RATE];

class R_RNA : RNA
{
  port translate = R_translation.rna;
  port degrade = R_RNA_degradation.rna;
  method create[] { new R.init[]; }
}
```

Fig. 11. mRNA transcriptions of A and R genes

6 Conclusion

In this article, we have described both the appeal of the stochastic π-calculus for modeling and running dynamic simulations of molecular biosystems, as well as some inadequacies inherent to this calculus as an engineering foundation for the modular development of complex models.

To remedy the situation, we described our ongoing effort to design a new higher-level programming language that makes it possible for complex models to be written and structured more directly rather than through an encoding *tour de force*.

Our language has an object-oriented flavour that provides explicit support to model the distinct behaviours that a molecular agent adopts in different states. Crucially, to allow the development of models reusable through inheritance and refinements, we abandon the directional send/receive style of communication, and adopt instead a style based on simultaneous exchanges. For want of space, we have omitted from our exposition some interesting features of our language, such as its support for describing complex assemblies comprised of several subagents.

Preliminary experience with the language has been very positive so far. In this article we included a model of a circadian clock to illustrate an application of the language. We have also produced models of the biosynthesis of glycogen, and of the regulatory switch of the lambda phage which are considerably simpler and more intelligible than their counter parts in the stochastic π-calculus [21,8]. We are currently working on a reasonably fine and complete model of the tryptophan operon.

An implementation of the language in Mozart/Oz is in progress.

```
reaction AR_complex_formation  <a,r> [AR_COMPLEX_FORMATION_RATE];
reaction A_degration <protein> [A_DEGRADATION_RATE];
reaction A_demotion  <protein> [A_DEMOTION_RATE];
reaction R_demotion  <protein> [R_DEMOTION_RATE];

class A
{
  port promoteA  = A_promotion.protein;
  port promoteR  = R_promotion.protein;
  port complexAR = AR_complex_formation.a;
  port degrade   = A_degradation.protein;
  port demotingA = A_demotion.protein;
  port demotingR = R_demotion.protein;

  constructor init[] {} become <free>;

  state <free>
  {
    port promoteA -> [slow] become <promotingA>[slow] {}
    port promoteR -> [slow] become <promotingR>[slow] {}
    port complexAR [<free>] -> [free] become <complex>[free] {}
    port degrade become dead {}
  }
  state <promotingA>[slow]
  {
    port demotingA become slow; become <free> {}
  }
  state <promotingR>[slow]
  {
    port demotingR become slow; become <free> {}
  }
  state <complex>[free]
  {
    port degrade become free; become dead {}
  }
}
```

Fig. 12. A Protein

```
reaction R_degradation <protein> [R_DEGRADATION_RATE];

class R
{
  port complexAR = AR_complex_formation.r;
  port degrade = R_degradation.protein;
  constructor init[] {} become <free>;
  state <free>
  {
    port complexAR[<free>] -> [free] become <complex>[free] {}
    port degrade become dead {}
  }
  state <complex>[free]
  {
    port degrade become free; become dead {}
  }
}
```

Fig. 13. R Protein

References

[1] Naama Barkai and Stanislas Leibler. Biological rhythms: Circadian clocks limited by noise. *Nature*, 403:267–268, 2000.

[2] Luca Cardelli. Brane calculi: interactions of biological membranes. In *Proceedings of CMSB 2004*, volume 3082 of *Lecture Notes in Bioinformatics*, pages 257–278, 2005.

[3] Bor-Yuh Evan Chang. PML: Toward a high-level formal language for biological systems. In *Workshop on Concurrent Models in Molecular Biology*, ENTCS. Elsevier, 2003.

[4] Dave Clarke, David Costa, and Farhad Arbab. Modelling coordination in biological systems. In *1st International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, 2004.

[5] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004.

[6] B. Alberts et al. *Molecular Biology of the Cell.* Garland Science, 2002.

[7] T. Ideker, T. Galitski, and L. Hood. A New Approach to Decoding Life: Systems Biology. *Annual Review of Genomics and Human Genetics*, 2(343), 2001.

[8] Céline Kuttler, Joachim Niehren, and Ralf Blossey. Gene regulation in the pi calculus: Simulating cooperativity at the lambda switch. In *Workshop on Concurrent Models in Molecular Biology*, ENTCS. Elsevier, 2004.

[9] Patricia L. Lakin-Thomas and Stuart Brody. Circadian rrythms in microorganisms: New complexities. *Annu. Rev. Microbiol.*, 58:489–519, 2004.

[10] P. Lecca and C. Priami. Cell cycle control in eukaryotes: a BioSPI model. In *Workshop on Concurrent Models in Molecular Biology*, ENTCS. Elsevier, 2003. BioConcur workshop proceeding.

[11] Paola Lecca, Corrado Priami, Paola Quaglia, B. Rossi, C. Laudanna, and G. Constantin. A stochastic process algebra approach to simulation of autoreactive lymphocyte recruitment. *SCS Simulation*, 80(6):273–288, June 2004.

[12] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994.

[13] Robin Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.

[14] Andrew Phillips and Luca Cardelli. A correct abstract machine for the stochastic pi-calculus. *Transactions on Computational Systems Biology*, 2005. to appear.

[15] Corrado Priami. Stochastic π-calculus. *Computer Journal*, 6:578–589, 1995.

[16] Corrado Priami and Paola Quaglia. Beta binders for biological interactions. In *Proceedings of CMSB 2004*, volume 3082 of *Lecture Notes in Bioinformatics*, pages 20–33, 2005.

[17] Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, 2001.

[18] Aviv Regev. *Computational Systems Biology: A Calculus for Biomolecular Knowledge.* Tel Aviv University, 2003. PhD thesis.

[19] Aviv Regev, E. M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. BioAmbients: An abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, 2004.

[20] Aviv Regev and Ehud Shapiro. Cells as computation. *Nature*, 419:343, 2002.

[21] Aviv Regev and Ehud Shapiro. The π-calculus as an abstraction for biomolecular systems. In Gabriel Ciobanu and Grzegorz Rozenberg, editors, *Modelling in Molecular Biology.* Springer, 2004.

[22] O. Wolkenhauer, B. Ghosh, and K.H. Cho. Control and coordination in biochemical networks (editorial notes). *IEEE CSM Special Issue on Systems Biology*, 24(4):30–34, 2004.