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

Technical Report CoSBi 17/2007

A Parallel Beta-binders Simulator

Stefan Leye
University of Rostock
`sl031@informatik.uni-rostock.de`

Adelinde M. Uhrmacher
University of Rostock
`lin@informatik.uni-rostock.de`

Corrado Priami
CoSBi
and
DISI, University of Trento
`priami@cosbi.eu`

Technical Report
"A Parallel Beta-binders Simulator"

Stefan Leye[*], Corrado Priami[+], Adelinde M. Uhrmacher[*]
[*]University of Rostock · Department for Computer Sciences &
[+]The Microsoft Research - University of Trento Centre for Computational and Systems Biology
$\{sl031, lin\}$@informatik.uni-rostock.de & priami@cosbi.eu

Abstract

Beta-binders is a comparatively new modeling formalism introduced for systems biology. To execute Beta-binders models, suitable simulators are required which translate the operational semantics of Beta-binders into a sound and efficient execution. Efficiency can be reached by parallel and distributed simulation and by a proper representation of models to ensure a fast manipulation. Both possibilities are considered in the implementation of the described hierarchical Beta-binders simulator. The description includes a tree structure which reflects π-Calculus and Beta-binders processes and the algorithm of the simulator which enables a distributed and optimistic, parallel execution.

1 Introduction

Biological research is concerned with a huge number of very complex processes. To understand them, modeling and simulation is a very important tool. In the past, the main approach to accomplish this task has been continuous modeling and simulation, but recently new concepts have emerged. Those include discrete event approaches towards modeling and simulation, the former including process algebra. Process algebra approaches vary from very general ones, like the π-Calculus [12] and its extensions [13, 15, 10] to very specific ones, like BioAmbients [18] mimicking biological compartments or Brane Calculi [3] for modeling membrane interactions.

Beta-binders [16] is an extension of the π-Calculus specifically designed for modeling and simulation in molecular and cell biology. A π-Calculus process is wrapped into a box. Boxes can be seen as compartments, however, nesting is not allowed. Special binders form interfaces on these boxes to handle interactions between them. Possible transitions in Beta-binders allow besides the simple internal π-Calculus transition, communications between boxes, the joining of two boxes into one and the splitting of one box into two. A stochastic extension [4] allows quantitative measures and makes a discrete event view possible.

A hierarchical simulator for Beta-binders has been introduced [8]. This attempt is inspired by the abstract simulator of (P)DEVS [21] using discrete event techniques and a hierarchical structure to enable distributed simulations. An adaptation of the Gillespie algorithm [7, 14] is responsible for scheduling the events.

We implemented this approach in the simulation framework James II [9].

2 Handling the Model Structure

Since Beta-binders and the π-Calculus, are based on formal languages, executing those implies typically inefficient *string* manipulation. To avoid this, a data structure is necessary, which can be handled easily by the simulation system. The simulator described in the next sections has been realized for the simulation framework James II, which is implemented in the object-oriented programming language Java [1]. Hence, a realization of the data structure in Java classes suggests itself.

2.1 Representation of the π-Calculus

A π-calculus process can be defined by the following context-free grammar:

$$\begin{array}{lll} P & ::= & P|P \mid !\pi.P \mid M \mid \nu(x).P \mid 0 \\ M & ::= & \pi.P \mid M+M \\ \pi & ::= & x\langle y\rangle \mid x(y) \mid \tau \end{array}$$

where x and y are elements of a set of names N.

In the stochastic π-Calculus, an extension of the π-Calculus which will be used here, the grammar is slightly changed:

$$\pi \quad ::= \quad (x\langle y\rangle, r) \mid (x(y), r) \mid (\tau, r)$$

with $r \in \mathbb{R}+ \ \cup \ \infty$ as a stochastic *rate*.

Similar to [2] we use a tree structure to represent π-Calculus processes but we do not maintain an additional structure for dynamic information (This part is handled by the simulator).

The π-Calculus is based on the notion of channels or names respectively, i.e. all the actions of a π-process are processed on channels and with channels. They represent the connection between communicating entities as well as exchanged values. Hence, channels are represented by an own class - *PiChannel*, which holds the name in the form of a *string* as an attribute.

For the stochastic extension, the class *StoPiChannel* is introduced. It inherits from the class *PiChannel* and is extended by a *double* value representing the rate. The general π-process is represented by the abstract class *PiProcess*, from which all the specific process classes are derived. $\nu(x).P$ creates a new name x which is only valid for P. The class *PiProcess* holds an attribute *restrictedChannels* which is an *ArrayList* containing references to these bound names.

π is the general definition for an action. All classes realizing an action have to inherit from the abstract class *PiAction*, which has a reference to the channel on which the action takes place. $x\langle y\rangle$ is a send action. If the action happens, channel y is sent over channel x. This is realized by the class *PiSend*. We use the polyadic π-Calculus [13] where not only one channel y but a vector of channels $\vec{y}$ can be sent. Therefore, *PiSend* contains the reference to the communicating channel (inherited from the *PiAction* class) and a list of references to the channels which are communicated over the channel.

$x(y)$ is a receive action. A channel is received on x and replaces y (or a vector of channels is received and replaces the vector of channels $\vec{y}$, repectively) in the following process. The

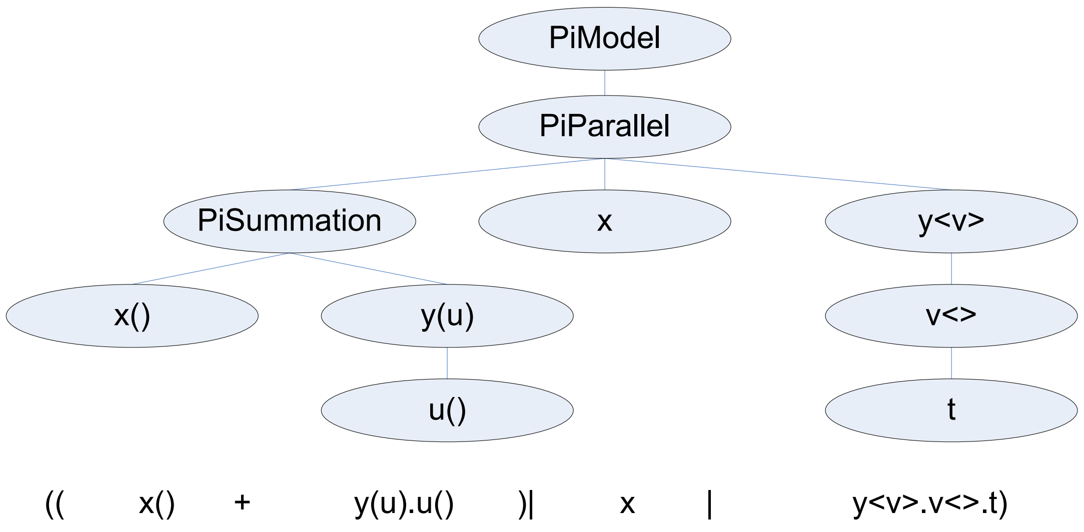

Figure 1: Example for the tree structure of a π-process.

class *PiReceive* is similar to *PiSend*, and contains a list of references to the channels that are replaced.

On the silent action τ no communication takes place. In the stochastic-π Calculus it symbolizes a delay. The τ action does not use a channel. We alterate this construct in the class *PiSilent*, which uses a channel to determine the rate of the action. Obviously this is not exactly the τ-action but we can substitue $\tau.P$ by $\nu(x)(x.P)$. This way gives us the ability to schedule the silent action as a simple communication. The class *PiSilent* simply inherits from *PiAction* without any new attributes.

In a sequence $\pi.P$ the process P is scheduled after the action π is executed. For this case the class *PiAction* has a reference to the next process. The guarded replication $!\pi.P$, symbolizes a sequential copying of the process $\pi.P$ according to the congruence rule $!\pi.P \equiv \pi.(P|!\pi.P)$. This is realized by a boolean value in the class *PiAction*. *True* means, that it is a replication, *False* means that it is a simple action. The simulator handles the action according to this property and manages the necessary transformations.

$P|P$ is the parallel execution of two processes. This is represented by the class *PiParallel*, containing the attribute *processes*, which is an *ArrayList* containing all the processes (as objects of the class *PiProcess*), that run in parallel. $M+M$ is the exclusive execution of guarded processes, which means, that only one of the processes can be processed. A guarded process is an action or a sequence. This construct is realized by the class *PiSummation*, which contains similar to the class *PiParallel* an attribute *processes*, which is in *ArrayList* containing all the processes of the summation. The processes are objects of the class *PiAction*. The empty process 0 is represented by the Java *null* object.

The class *BasicPiConstruct* is the superclass for *PiProcess*. It holds the attribute *parent* which is a reference to another *BasicPiProcess*. With it a tree representation of a π-Calculus process is possible, where the *parent* of a process is the parent node in the tree. The children of a summation and a parallel process are the composing processes, the child of an action is the following process in the sequence.

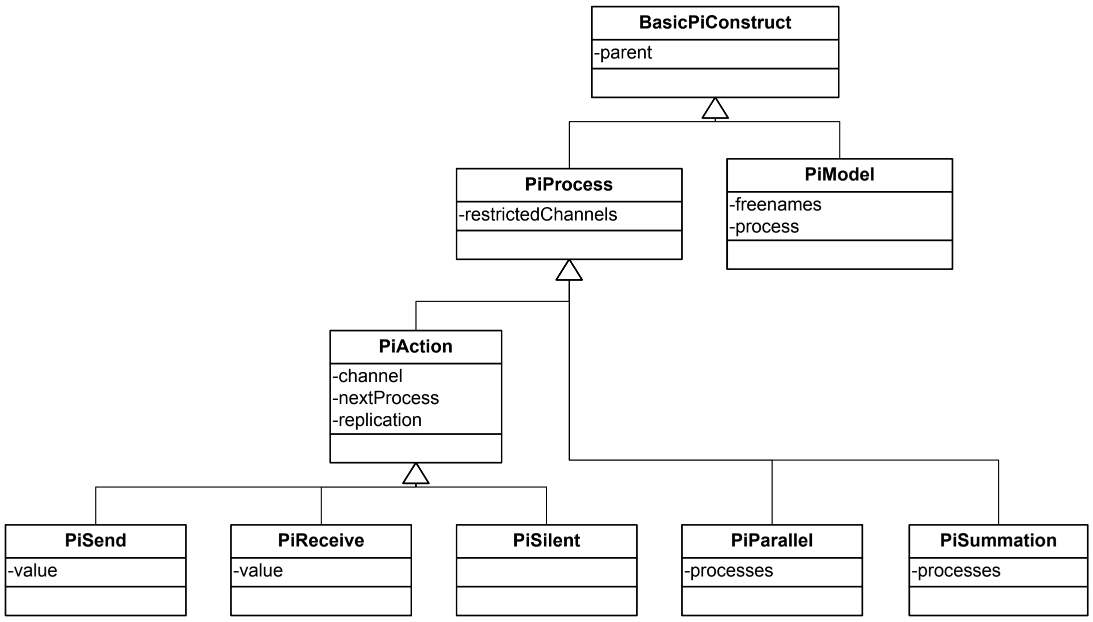

Figure 2: Classes representing π-Calculus processes.

The class *PiModel* inherits from the class *BasicPiConstruct*. The simulator works with a *PiModel* object. Additional to the reference to the process to be simulated it holds information necessary for an efficient simulation like a list of all the free names (references to the channels respectively) contained in the process.

2.2 Representation of Beta-binders

In Beta-binders an interface is wrapped around π-Calculus processes, they become *BioProcesses*:

$$\begin{array}{lll} B & ::= & \mathbf{B}^*[P] \mid B||B \\ \mathbf{B} & ::= & \beta(x,\Delta) \mid \beta^h(x,\Delta) \end{array}$$

where P is a π-Calculus process of the grammar defined in 2.1 extended in the following way:

$$\pi \quad ::= \quad x\langle y\rangle \mid x(y) \mid x \mid \mathit{expose}(x,\Delta) \mid \mathit{hide}(x) \mid \mathit{unhide}(x)$$

respectively:

$$\begin{array}{lll} \pi & ::= & (x\langle y\rangle, r) \mid (x(y), r) \mid (x, r) \mid \\ & & (\mathit{expose}(x,\Delta), r) \mid (\mathit{hide}(x), r) \mid (\mathit{unhide}(x), r) \end{array}$$

for the stochastic case. Note that in the following a $\pi Process$ is a π-Calculus process including the extensions above. The interface consists of *binders* $\beta(x, \Delta)$, where a channel x is connected to a type δ on which a communication between two *BioProcesses* can take place. The type is represented as a list of *strings*. How these *strings* are used for communications is discussed in section 3. The *binder* is realized in the class $BetaBinder$, holding a reference to the channel and a $HashSet$ of *strings* for the type. We use a $HashSet$ here for efficiency reasons. A *binder* can be hidden, which means that no communication is possible on it. This is realized by the boolean attribute *hidden*.

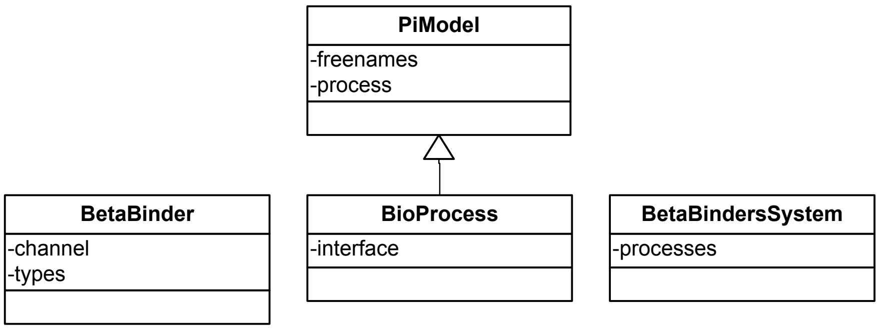

Figure 3: Additional classes to represent Beta-binders.

A *BioProcess* extends the $PiModel$ class. The $BetaBindersBioProcess$ contains a new attribute $interface$, which is a *HashMap* mapping each bound channel to its *binder*. A system or set of parallel *BioProcesses* is realized by the class *BetaBindersSystem*, which holds additionally to an *ArrayList* of single *BioProcesses*, information necessary for the simulation.

The three additional actions are responsible for *binder* manipulation.The expose action $expose(x, \Delta)$ creates a new *binder* for the *BioProcess*. The associated class *BetaBindersExpose* holds a reference to the channel which will be bound and a $HashSet$ of *strings* representing the type. $hide(x)$ and $unhide(x)$ change the hidden state of the binder connected to channel x. Each of them has been realized as an own class, extending *PiAction*. They do not need additional attributes.

3 A Hierarchical Beta-binders Simulator

The idea of [8] divides the simulator into several components according to the structure of the model. Each *BioProcess* is handled by a so called *Simulator*, which processes all internal events. The whole system is managed by a $Coordinator$, which handles the communication between the *BioProcesses*. On top of that a $RootCoordinator$ is responsible for the simulation's progress in time. It is possible to create a simulator tree according to the model tree (see figure The interaction between the components is done by messages which facilitates a distributed execution. According to the semantics of Beta-binders, the simulator has to handle four types of transitions:

- the *intra*-communication describes a communication inside of a *BioProcess*, which is in most cases a stochastic-π transition. Additionally it can be the processing of one of the new actions: *expose*, *hide* and *unhide*. These transitions are handled by the *Simulator* component. 4).

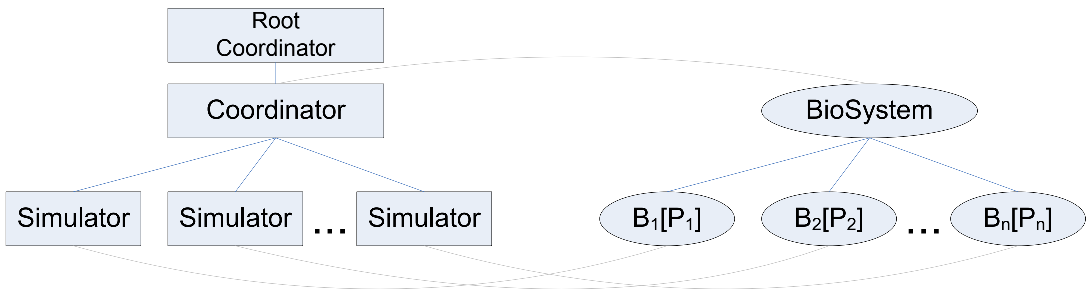

Figure 4: Scheme for the mapping of the model tree onto the hierarchical simulator.

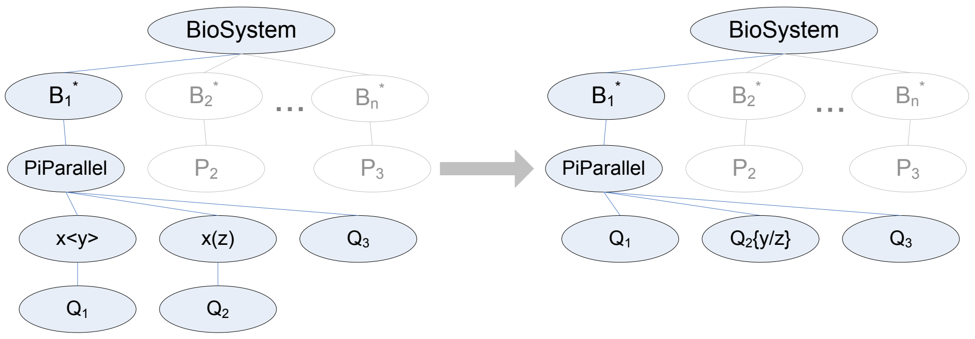

Figure 5: Example for the changes of the model tree caused by an *intra* event on channel x.

- the *inter*-communication describes a communication between two *BioProcesses*, to be more specific one *BioProcess* sends over a *binder*, while a different *BioProcess* receives over a *binder*. The connection between the *binders* is derived from their types. Therefore a special affinity (a positive, real number) is defined for each pair of types which shall be able to create a Communication. Since types in our implementation consist of sets of *strings* the affinities are defined for pairs of those *strings*. During simulation those affinities are applied similarly to the propensity calculation in basic Gillespie. The affinities are stored in the class *TypeCoupling* including as attributes the sending and receiving *string*. Although we use the stochastic variant of Beta-binders [4] we do not distinguish between a bimolecular reaction and a homodimerization. This is not necessary, because our *BioProcesses* are individual objects each simulated by an own simulation component. So whether two communicating partners are structural congruent as in homodimerization is of no interest, as the individuals are distinguished. While scheduling and forwarding messages between the involved *BioProcesses*, i.e. their *Simulators*, is done by the *Coordinator* component, the changes in the model structure is the responsibility of the *Simulators*.

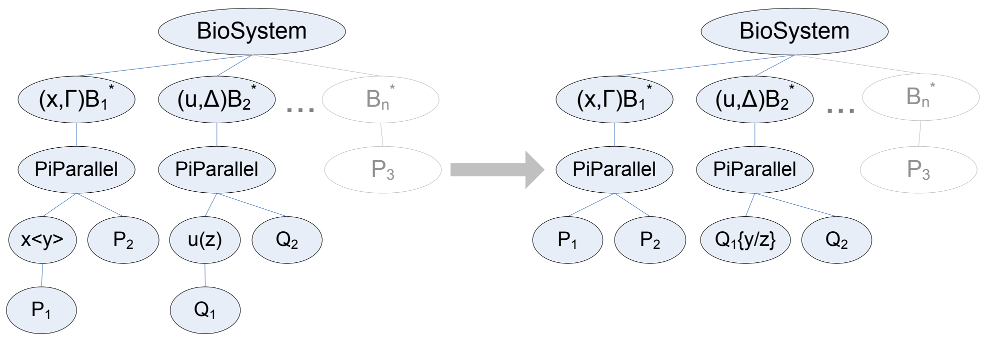

Figure 6: Example for the changes of the model tree caused by an *inter* event communicating over the types Γ and Δ.

- the *join*-transition describes the joining of two *BioProcesses* into one. Instead of the very general definition of *join* in [16] or as the idea put forward in [8] where *join* was interpreted as an invariant, we introduce a specific interpretation that is inspired by and based on the *inter*-communication. For the *join*, again an affinity between types (or *strings*, rspectively) is defined. Given that two *BioProcesses* are in the state for an *inter*-communication, two things can happen: either a normal *inter*-communication or a *join* combined with an *intra*-communication, depending on the type of the affinity. If the *BioProcesses* want to communicate over types which have a *join* affinity, they will join, i.e, the π*Process* of the 'sending' *BioProcess* is absorbed by the 'receiving' *BioProcess* (i.e., it is set parallel to the original π*Process* of the 'sending' *BioProcess*). The 'sending' channel in the engulfed π*Process* is replaced by the 'receiving' channel (an *intra*-communication is now possible between the engulfed process and the former π*Process* of the engulfing *BioProcess*).

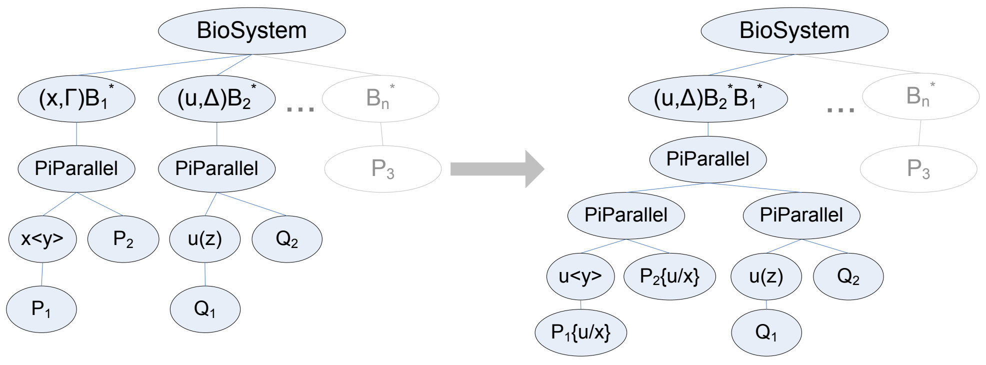

Figure 7: Example for the changes of the model tree caused by an *join* event, initiated by the types Γ and Δ.

The *join*-transition is displayed by the following rule:

$$\frac{P \equiv \nu\widetilde{u}((x(w), r_x).P_1|P_2) \qquad Q \equiv \nu\widetilde{v}((y\langle z\rangle, r_y).Q_1|Q_2)}{B \equiv \beta(x:\Gamma)\mathbf{B}_1^*[P]||\beta(y:\Delta)\mathbf{B}_2^*[Q] \overset{J;B;\alpha_j(\Gamma,\Delta);(1,1)}{\longrightarrow} \beta(x:\Gamma)\mathbf{B}_3^*[P'|Q']}$$

where $P' = \nu\widetilde{u}(x(w).P_1|P_2), Q' = \nu\widetilde{v}(x\langle z\rangle .Q_1\{x/y\}|Q_2\{x/y\})$ and $\mathbf{B}_3^* = \mathbf{B}_1^* \cap \mathbf{B}_2^*$ provided $x, z \notin \widetilde{u}$ and $x, y, z \notin \widetilde{v}$. The *join* is handled by the *Coordinator* component.

- the *split*-transition describes the splitting of one *BioProcess* into two. Again we use a specific interpretation here. For each *BioProcess* $B = \mathbf{B}^*[P]$ we introduce a set of channels $split(B)$, with $split(B) \subseteq fn(P)$. If a silent action occurs on one of these channels, the whole following sequence is put into a new created *BioProcess* holding the same interface as the *BioProcess* where the action occured.

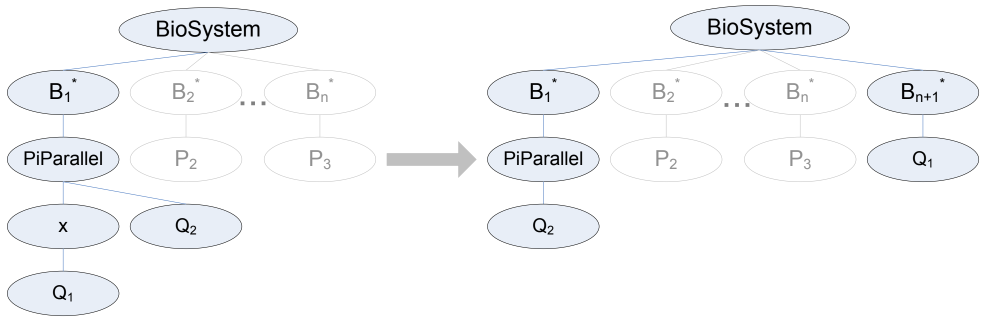

Figure 8: Example for the changes of the model tree caused by a *split* event with channel x as trigger.

The *split*-transition is displayed by the following rule:

$$\frac{P \equiv \nu\widetilde{u}((x, r_x).P_1|P_2)}{B \equiv \mathbf{B}^*[P] \overset{S;B;r_x\times(1+n_O\times n_I);(1,1)}{\longrightarrow} \mathbf{B}^*[P_1]||\mathbf{B}^*[P_2]}$$

where $n_O = 1 + Out_x(P_2)$ and $n_I = 1 + In_x(P_2)$
provided $x \in split(B)$
As a split is based on the *intra* communication and happens locally it can be handled by the *Simulator* component. The *Simulator* generates a new *BioProcess*, however, the existence of the new *BioProcess* has to be announced to the *Coordinator*, which will also generate the *Simulator* responsible for executing the new *BioProcess*.

3.1 The *Simulator* Component

The algorithm of the *Simulator* is based on the stochastic-π machine [14], extended by some additional features to handle Beta-binders. Instead of lists we work on the model structure introduced in section 2.1.

The *Simulator* holds a list of all the possible activities which can occur on each channel at the actual state. This list contains all the possible communication pairs (each with a sender and a receiver), and references to the single actions, this includes silent actions, *expose*, *hide* and *unhide* actions. Following the argumentation line in [8], the algorithm consists of three phases: the *preEvent* preparing a simulation step, the *doEvent* executing the step and the *postEvent* providing information for the next event.

In the *preEvent* a message from the *Coordinator* is received. This can be either a *StarMessage* containing only the actual simulation time or an *XYMessage* containing information necessary for an *inter* communication. The *doEvent* distinguishes between the

Algorithm 1 Pseudocode for the *preEvent* of the *Simulator* Component

```
wait until msg received
update time
```

different types of incoming messages. If it is a *StarMessage* the *Simulator* needs to execute the next scheduled *intra* communication. The algorithm uses the list of all possible activities, for the actual channel. One of these activities is picked randomly. If the received message is

Algorithm 2 Pseudocode for the *doEvent* of the *Simulator* Component

```
if msg is a StarMessge then
  get actualChannel
  get random activity on actualChannel
else if msg is a XY Message then
  if msg has a Value then
    get the type
    get the channel
    get a random receiver on the channel
  else
    get the type
    get the channel
    get a random sender on the channel
    get the value
    send XY message to parent
  fi
fi
delete actions from model tree
case
  one of the actions is a receiver:
    substitution
  one of the actions is a expose:
    create new binder
  one of the actions is a hide:
    hide binder
  one of the actions is a unhide:
    unhide binder
  one of the actions is a silent:
    if channel is a split indicator then
      split
    fi
update communication lists
create new StructureUpdateMessage
```

an *XYMessage* an *inter* communication needs to be performed. The *Simulator* can be either the sender or the receiver. The *XYMessage* holds the type on which the communication takes place and a value if the *Simulator* is a receiver. With these information it is possible to select a channel and after that a send or receive action at the level of the $\pi Processes$. In contrast to the proposed solution in [8] where the values were part of a var-struct message which wraps up the execution of a *doEvent*, here the sender of an *inter* communication needs to send its value explicitly by an additional *XYMessage* to the *Coordinator* which will forward the message to the receiver. This explicit message facilitates handling *inter* communication for the *Coordinator*. Otherwise, as the system is based on the polyadic π calculus, the *Coordinator* would be faced with handling quite complex structure updates as the values that are sent to the *Coordinator* are lists of channels.

After the *Simulator* identified the corresponding actions within the *BioProcess*, their execution starts. The first step of the execution is done by a restructuring of the model tree. In the simple case, this is done by deleting the action or actions from the tree. If a performing action is a replication the subtree of the action has to be transformed according to the rule $!\pi.P \equiv \pi.(P.!\pi.P)$ before deleting the action. If a performing action is part of a summation, the summation needs to be deleted as well.

The second step is the processing of the additional activities. Those include the substitution for a communication or the creation, hiding or unhiding of a *binder* for *expose*, *hide* or *unhide* respectively. If a silent action has been processed the *Simulator* needs to check if the actual channel is a channel demanding a *split*. If that is the case a new *BioProcess* has to be created with a copy of the interface of the actual one. The internal π*Process* of the newly generated *BioProcess* will be the sequentially following process of the silent action.

After doing the transitions, the lists of activities have to be updated. This is done by going through the tree structure and getting all the actions, that can perform. At this phase we also get the structure information necessary for the *Coordinator*. This information is sent via a *StructureUpdateMessage*. It holds the changes of the count of possible senders and receivers of the *binder's* types (their *strings* in our case)), the time of the next *intra* event which will be calculated later, and the splitted *BioProcesses*, if existing. Finally, the

Algorithm 3 Pseudocode for the *postEvent* of the *Simulator* Component

```
for all channels in the process do
  calculate next delay
get smallest delay
set associated channel as next channel
upadate tonie for StructureUpdateMessage
```

Simulator has to calculate the time of the next *intra* event and the associated channel in the *postEvent*. This is done by a variant of the Gillespie algorithm [7] using the notion of the channel activity to stochastically select the channel on which the next reaction occurs. The delay to the next *intra* event is an exponentially distributed random number taking the rates and the counts of possible activities as parameters of the channels into account. It is added to the current simulation time. The result (the time of the next *intra* event) is sent with the *StructureUpdateMessage* to the *Coordinator*.

3.2 The *Coordinator* Component

The *Coordinator* handles the interaction between the *BioProcesses*. Hence, similar to the *Simulator* it holds lists containing information about the possible communications. Since the *inter* communication and the *join* transition are based on affinities, these lists do not depend on channels but on the *TypeCoupling* objects.

We do not use the Gibson and Bruck method [6] as proposed in [8], since it is hardly possible to construct a dependency tree in our approach. Although we work with event queues to schedule some events in advance. This is possible because of the memoryless property of exponentially distributed random numbers.

As the *Coordinator* is responsible for the scheduling of all the interactions in the system (even the *intra* interactions) it maintains two event queues. The first one contains the events caused by interactions between *BioProcesses*, i.e. *inter* communications and *joins* which are represented by the *TypeCouplings*. The second one contains the events scheduled by the *StructureUpdateMessages* of the *Simulators* which covers *intra* communications and *splits*.

The algorithm of the *Coordinator* comprises again the three phases: *preEvent*, *doEvent* and *postEvent*.

In the *preEvent* a message from the *RootCoordinator* is received, which contains the time stamp of the next event. This message is always a *StarMessage* containing only the current simulation time.

Algorithm 4 Pseudocode for the *preEvent* of the *Coordinator* Component

```
wait until msg received
update time
```

The *doEvent* checks the times of the next events stored in the event queues. The event associated to the time matching the current time stamp is selected and will be executed.

If the *event* is an *intra* event or *split*, a *StarMessage* is sent to the *Simulator* responsible for its execution. If the *event* is an *inter* event or *join*, the *TypeCoupling* is taken. If it is associated to an *inter* communication, a *Simulator*, which can send on the sending type is selected randomly and an *XYMessage* is sent to it. The *Coordinator* waits for an answer *XYMessage* (see section 3.1) which contains the values of the communication. Another *Simulator* which can receive on the receiving type is selected and the *XYMessage* is forwarded to it.

If the event is a *join*, two *Simulators* able to perform on the associated types are selected randomly. The *BioProcess* which will be engulfed is sent to the *Simulator* of the engulfing *BioProcess*. The fusion of the processes happens according to the rule specified before.

After the internal changes of the *BioProcesses*, the *Coordinator* waits for the updates of the information necessary to schedule the next events. It waits for one *StructureUpdateMessage* if the actual event is an *intra*, *split*, or *join* transition and for two if it is an *inter* communication. With the received information, the *TypeCoupling* lists are updated. Each message also contains the time of next event of the *Simulator*, which is put into the appropriate event queue. If the *StructureUpdateMessage* contains a splitted process a new *Simulator* is created to handle the new *BioProcess*. They are integrated into the simulator and model structure, respectively. The *postEvent* of the *Coordinator* has a similar function like the *postEvent* of the *Simulator*. For each *TypeCoupling* the affinity and the number of possible transitions on it are taken into account to get an exponential distributed random number, which represents the delay for the next event on the types. The *TypeCoupling* is stored with its time of next event in the dedicated event queue. The minimum of the two event queues is sent to the *RootCoordinator* via a *StructureUpdateMessage*. The *RootCoordinator* starts the next simulation step.

Algorithm 5 Pseudocode for the *doEvent* of the *Coordinator* Component

```
get time from msg
if time = min of intra event queue then
  send StarMessage to Simulator
else if time = min of inter event queue then
  if event is inter then
    get TypeCoupling
    get random Simulator on sending type
    send XYMessage to Simulator
    wait for XYMessage
    get random Simulator on receiving type
    forward XYMessage to Simulator
  else
    get TypeCoupling
    get random Simulator on embedable type
    get random Simulator on embeding type
    get bio process of embedable Simulator
    join bio process into model of embeding Simulator
  fi
fi
if event is inter then
  wait for two StructureUpdateMessages
else
  wait for one StructureUpdateMessage
fi
if StructureUpdateMessage contains new process then
  create new simulator
  wait for StructureUpdateMessage
fi
update lists
```

Algorithm 6 Pseudocode for the *postEvent* of the *Coordinator* Component

```
for all TypeCouplings in the system do
  calculate next delay
update inter event queue
t1 = min of inter event queue
t2 = min of intra event queue
tonie = getMin(t1, t2)
upadate tonie for StructureUpdateMessage
```

4 An Optimistic Variant

Although the hierarchical Beta-binders simulator described in the previous section can be executed in a distributed way very easily, its parallelization opportunities are rather limited. Because of the stochastic factor there are practically no events that happen at the same time. The only event type, where more than one *Simulator* could work at the same time, would be the *inter* communication. That means that only two *Simulators* work in parallel.

We need a sophisticated concept for a parallel execution. Discrete event parallel simulation distinguishes between conservative and optimistic approaches [5]. With a conservative approach all safe events can be processed in parallel. Because events shall be executed in time stamp order, an event is safe if it is guaranteed that no other event happens prior to it. To identify these safe events guarantees are exchanged between the simulators residing at different nodes. Those are based on the local time stamp of the simulators and typically take a look ahead into consideration. The look ahead defines the time interval within which no event will be scheduled for the receiving simulator by the sending one. So the sending simulator gives a guarantee until which time stamp it is safe for the receiver to proceed. Obviously the efficiency of conservative approaches is closely related to the ability to define significant lookaheads [11]. In contrast, an optimistic approach assumes "optimistically" that no conflict will happen and processes its events. In the case it receives an event with a time stamp smaller than the ones it has already processed (a so called straggler event), it rolls back to the state before this event and undoes all the messages it has sent to other simulators by launching anti-messages.

The stochastic factor in the Beta-binders simulator hampers defining suitable lookaheads. No *Simulator* which is able to communicate via an *inter* communication is able to guarantee that no event will be scheduled for it before a specific time. Therefore, an optimistic variant of the simulator appears more suitable. However, due to the large effort required in handling the model-structure an unbounded optimistic simulation does not appear very promising. Therefore, we decided to combine conservative and optimistic features in our parallel simulator. Whereas the *intra* events are processed in an optimistic manner, the *inter* events form a kind of barrier and as such are only processed if they are safe. The later is guaranteed by letting all *BioProcesses* advance up to this barrier.

Within the *Simulators* we need mechanisms to rollback executed events, this includes saving the state of a *BioProcess*. Storing exchanged messages is not required, as those will only be processed if they are safe. At the level of the *Coordinator* the parallel execution including initiating roll backs is realized.

4.1 Realizing Rollbacks

To be able to roll an event back it is necessary to save the states which existed before the event. This is possible by using the tree structure of a π*Process*. It is only necessary to roll back *intra* events (see section 4.2). Since these events are transitions of the internal π*Processes* of *BioProcesses*, it is required to save the states of them. One possibility to save previous states, is to keep the information about the changes leading from these old states to the actual one.

The transitions of π*Processes* base on the deletion of the actions which participate on it and the replacement of them with the sequentially following process. The optimistic *Simulator* component holds a list containing each time stamp on which an action has been removed from the model tree, a reference to the action (or to the summation if the action belonged to one) and a reference to the process which replaced the action. Additional information, dependent on the transition are saved. This includes a list of the substituted channels, if the action is a receive action or a reference to the created binder, if the action is an *expose* action.

If a *Simulator* receives a rollback message it extracts the time stamp until which the π*Process* should be rolled back. Now the state of this time can be reconstructed step by step. The time of the last change is taken from the list of changes. The reference to the parent node saved in the class *PiProcess* and the reference to the process, which replaced the action, saved in the list of changes define the exact position of the action (or summation) in the model tree. The action needs to be linked at the parent node on the position of the process which replaced it before. The former replacing process gets son node of the action again. If there was a substitution it is made undone, if the action was a *hide*, *unhide* or *expose* the binder is made unhidden, hidden or deleted, respectively. After the change is cancelled the *Simulator* takes the time of the next change and compares it to the rollback time. If it

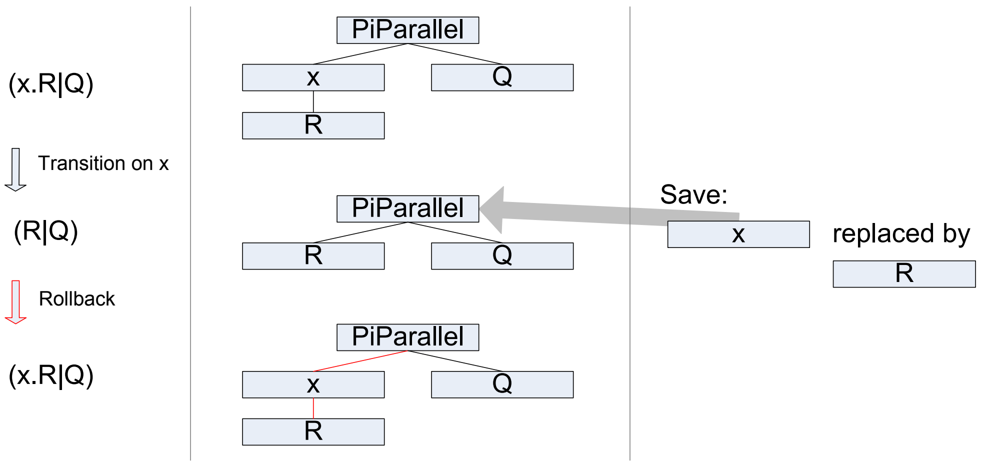

Figure 9: Example for a transition and its rollback, shown with the π-Calculus syntax (right), the tree structure (middle) and the information needed to get the last state (left).

is smaller the *Simulator* stops, the right state has been rebuilt, if it is greater or equal the change will be cancelled too. Since the *Coordinator* drops the *StructureUpdateMessages* of *Simulators* which need a rollback (see section 4.2), it is not necessary to send an update of the structure to the *Coordinator* after a rollback.

4.2 A Moving Time Windows *Coordinator*

In section 3.2 the *Coordinator* is responsible for the execution of one event at each simulation step. The optimistic *Coordinator* handles multiple events at one step. We use a moving time windows approach [20], where all the events which are scheduled before a time barrier (which is reset in each simulation step) are processed. This is a hybrid concept of parallel simulation. There is an optimistic part because the barrier is no guarantee here, it only indicates a time frame where conflicts are unlikely. In our case the barrier is the next *inter* or *join* event.

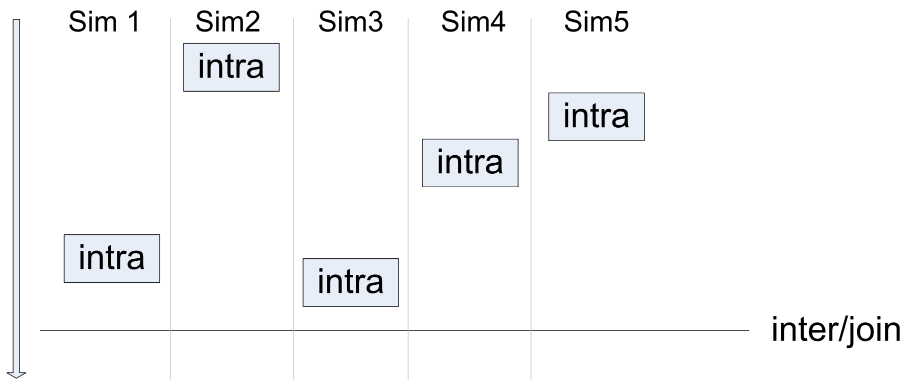

Figure 10: Phase 1: Executing *intra* events.

The algorithm is devided into four phases. In phase one a $StarMessage$ is sent to each $Simulator$ which can execute an $intra$ event, before the next $inter$ or $join$ event (the time barrier) in the system. In phase two the $Coordinator$ waits for a $TimeWarpUpdateMessages$

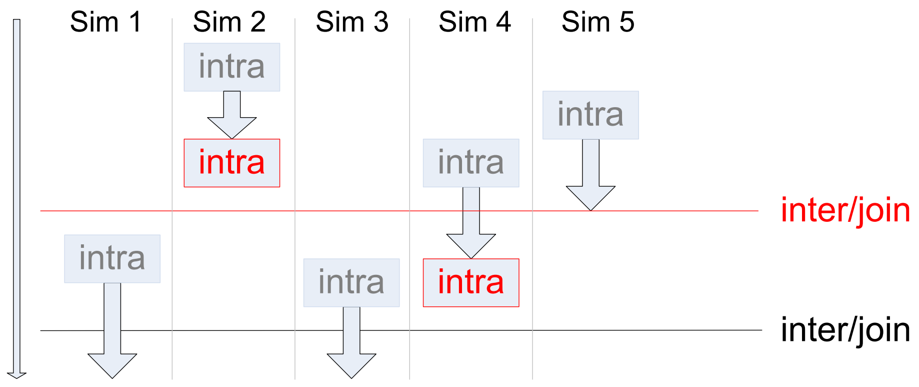

Figure 11: Phase 2: Updates and scheduling of new events.

from each $Simulator$ which executed an event. The $TimeWarpUpdateMessages$ is an extension of the $StructureUpdateMessages$ holding the time stamp of the event. This is necessary because the updates have to be done in the order of the execution of their events. This is a bottleneck in the algorithm because the updates can only be done sequentially. With the

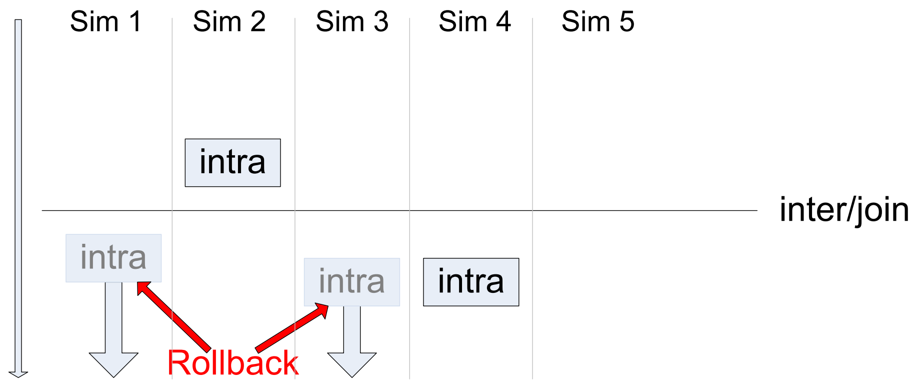

Figure 12: Phase 3: Rollback of executed events scheduled after the new barrier.

updates a new $intra$ or $join$ event could be scheduled which lies before the original one. The barrier moves backward. This could conflict the events executed in phase one, which have a time stamp after the moved barrier. The $TimeWarpUpdateMessages$ from the $Simulators$ which executed those events are dropped, and the $Coordinator$ sends $RollbckMessages$, containing the time stamp of the new barrier, to them. The $Simulators$ do the rollbacks and recover the states at the time of the barrier. In phase four the $Coordinator$ checks whether there are $intra$ events left, scheduled before the barrier. If there is no one left, the $intra$ event or $join$ representing the barrier is executed. If there are $intra$ events before, the bar-

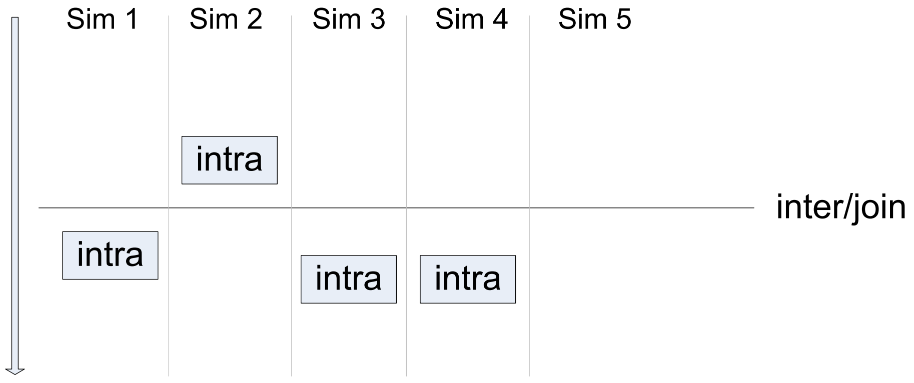

Figure 13: Phase 4: Executing *intra* or *join* if possible.

rier remains for the next simulation step, which means that no *intra* or *join* event will be executed at the actual step.

5 Related Work

The *Simulator* component presented in section 3.1 is based on a stochastic-π simulator inspired by the stchastic-π machine (spim) [14]. Spim translates a stochastic-π Calculus process into a list of summations, for an efficient handling. For each channel x the count of possible communications is calculated, according to $(in_x * out_x) - mix_x$, where in_x is the count of possible inputs on x, out_x is the count of possible outputs on x and mix_x is the count of communications on x where sender and receiver lie in the same summation (i.e. they can not communicate). The channels on which a transition occurs is calculated by a variant of the Gillespie method, using the notion of channel activity. For each channel x the propensity (the product of a channels *rate* and the number of possible communications on it) p_x is calculated. Non-zero values of p_x are stored in a list (x_μ, p_μ), with $\mu \in \{1, ..n\}$. The sum of the propensities is calculated and a random number n_1 between 0 and 1 taken. The time delay to the next event τ is calculated according to $\tau = (1/s) * ln(1/n_1)$. A second random number n_2 is taken. The next reaction channel x_μ is achieved by $\sum_{v=1}^{\mu-1} p_v < n_2 * s \leq \sum_{v=1}^{\mu} p_v$. With the channel, the next transition can be choosen by randomly selecting a communication pair of the channel. We use this concept for an own implementation of a stochastic-π simulator, which is the base of the Beta-binders *Simulator* component.

[19] presents a Beta-binders simulator following a different approach than the one described here. *BioProcesses* are seen as instances of species, multiple *BioProcesses* belong to one species if they are structural congruent [17]. The simulator holds a list of these species together with the count of the *BioProcesses* of each species. A transition does not lead to a structural change of a process but to a decrease or respectively increase of the species counts. The possibility of composing and decomposing complexes of *BioProcesses* by introducing dynamic communication connections, replaces the *split* and *join* transitions. This way gives the opportunity to represent molecular structures but drops the original idea of a simple modeling of absorbing and excluding processes. This attempt is very efficient when

handling large amounts of similar *BioProcesses*. Although, it is impossible to trace a single *BioProcess* and its behaviour during a simulation. We focus on individual *BioProcesses* to make a tracing of specific entities possible. We get this ability through our tree structure. Each process of a system is represented by a node, the structure of a process is represented by the structure of the sub tree on the associated node. Observation can be done by following the structure of the sub tree during simulation.

6 Conclusion

We presented a parallel Beta-binders simulator. We extended the ideas of [8], which provided the possibility of a distributed simulation and created an optimistic variant to enable a real parallel execution. A model structure has been introduced to represent π-Calculus and Beta-binders processes in an entity based way.

Future work needs to evaluate the time warp simulator with an appropriate model. We have to study the influences of networks and different types of models on the performance of our approach. In this context a full optimistic attempt without a barrier could be developed and considered in the evaluation as well.

7 Acknowledgements

The research was done while the first author was visiting Microsoft Research - University of Trento Centre for Computational and Systems Biology. The research was supported by the DFG project dIEM oSiRiS and the DFG project DiErMoSiS (at the University of Rostock) and the Microsoft Research - University of Trento Centre for Computational and Systems Biology. We would like to thank Allessandro Romanel, Lorenzo Dematté, Celine Kuttler, Andrew Phillips, Roland Ewald and Jan Himmelspach for constructive suggestions in realizing the simulator and for fruitful discussions. In particular, the first author would like to thank the entire team in Trento for the inspiring and productive atmosphere.

References

[1] Java JDK 6.0. http://java.sun.com/javase/downloads/index.jsp.

[2] A. Bloch, B Haagensen, M. K. Hoyer, and S. U. Knudson. The stopi-calculus and simulator - a stochastic pi-calculus and the implementation of a simulator. Technical report, Aalborg University, Department of Computer Science, 2003.

[3] Luca Cardelli. Brane calculi - interactions of biological membranes. *Computational Methods in Systems Biology: International Conference CMSB 2004*, pages 257–278, 2005.

[4] Pierpaolo Degano, Davide Prandi, Corrado Priami, and Paola Quaglia. Beta-binders for biological quantitative experiments. *Electronic Notes in Theoretical Computer Science*, 2005.

[5] Richard Fujimoto. *Parallel and Distributed Simulation Systems.* John Wiley & Sons, 2000.

[6] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry*, 104(9):1876–1889, 2000.

[7] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81, 1977.

[8] Jan Himmelspach, Paola Lecca, Davide Prandi, Corrado Priami, Paola Quaglia, and Adelinde M. Uhrmacher. Developing an hierarchical simulator for beta-binders. In *Workshop on Parallel and Distributed Simulation (PADS)*, 2006.

[9] Jan Himmelspach and Adelinde M. Uhrmacher. Plug'n simulate. In *Proceedings of the 40th Annual Simulation Symposium*, pages 137–143, 2007.

[10] C. Kuttler, C. Lhoussaine, and J. Niehren. A stochastic pi calculus for concurrent objects. Technical report, INRIA., 2006.

[11] Yi-Bing Lin and Edward D. Lazowska. Exploiting lookahead in parallel simulation. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 1:457, 1990.

[12] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes part i and ii. *Information and Computation*, 96:1–40, 41–77, 1992.

[13] Robin Milner. The polyadic π-calculus: a tutorial. *Logic and Algebra of Specifications*, 94, 1993.

[14] Andrew Phillips and Luca Cardelli. A correct abstract machine for the stochstic pi-calculus. *Electronic Notes in Theoretical Computer Science*, 2004.

[15] Corrado Priami. Stochastic π-calculus. *The Computer Journal*, 38/7:578–589, 1995.

[16] Corrado Priami and Paola Quaglia. Beta binders for biological interactions. In *Computational Methods in Systems Biology: International Conference CMSB 2004*, 2004.

[17] Corrado Priami and Alessandro Romanel. On the decidability and complexity of the structural congruence for beta-binders. Technical report, The Microsoft Research - University of Trento Centre for Computational and Systems Biology, 2006.

[18] A. Regev, E.M. Panina, W. Silverman, L. Cardelli, and E.Y. Shapiro. Bioambients: an abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, 2004.

[19] Alessandro Romanel, Lorenzo Dematté, and Corrado Priami. The beta workbench. Technical report, The Microsoft Research - University of Trento Centre for Computational and Systems Biology, 2007.

[20] L. Sokol and B. Stucky. Mtw: Experimental results for a constrained optimistic scheduling paradigm. *In Proc. SCS Multiconf. Distributed Simulation*, 22:169–173, 1990.

[21] B.P. Zeigler, D.H. Kim, and A.C. Chow. Abstract simulator for the parallel devs-formalism. 1994.