# A Flexible Architecture for Performance Experiments with the pi-Calculus and its Extensions

Stefan Leye
Institute of Computer Science
Albert-Einstein-Str. 21
18059 Rostock, Germany
stefan.leye@uni-rostock.de

Mathias John
Institute of Computer Science
Albert-Einstein-Str. 21
18059 Rostock, Germany
mathias.john@uni-rostock.de

Adelinde M. Uhrmacher
Institute of Computer Science
Albert-Einstein-Str. 21
18059 Rostock, Germany
adelinde.uhrmacher@uni-rostock.de

## ABSTRACT

The $\pi$-Calculus is a modeling formalism for concurrent processes. Realized as part of the plug-in based modeling and simulation framework JAMES II, we propose an architecture for $\pi$-Calculus-based modeling and simulation, which supports both *flexibility* and *efficiency*. Facilitating the design of new $\pi$-Calculus-based formalisms and simulators is of particular relevance in the field of computational systems biology, for which many different $\pi$-Calculus dialects and simulators have been and still are being developed. Therefore, a flexible representation of $\pi$-Calculus models is used, which is illustrated by a mapping from the biochemical variant of the $\pi$-Calculus to the representation. Simulation engines are exchangeable and even automatically configurable according to the task at hand.

Moreover, we present three different simulator implementations, working on the model representation. Efficiency denotes that our architecture supports the implementation of high-performance simulators. In order to assess efficiency, we perform experiments with these simulators and compare the results to the current cutting edge implementation in the field, the Stochastic Pi Machine.

## Categories and Subject Descriptors

I.6.7 [**Simulation Support Systems**]: Environments; I.6.7 [**Simulation Languages**]; I.6.8 [**Types of Simulation**]: Discrete-event; G.4 [**Reusable Software**]

## General Terms

pi-Calculus, Simulation environment

## Keywords

Process algebra, pi-Calculus, Discre-event simulation, Simulation environment, Software

## 1. INTRODUCTION

The plethora of existing modeling formalisms is a challenge for the development of modeling and simulation tools. In order to conduct experiments, tools have to be adapted to the specific needs of a formalism. The modeling and simulation framework JAMES II [12] represents a general solution for this problem, by providing a plug-in based architecture that allows the integration of diverse formalisms and tools into one experiment structure. While experimentation is facilitated, the user still has to put effort into the structuring and implementation of the required plug-ins in order to face specific characteristics of the desired formalism. However, this effort can be reduced by structuring plug-ins according to similar characteristics that a family of formalisms is sharing to facilitate the implementation of new dialects.

The multiple variants of the $\pi$-Calculus [20] form a family of formalisms [24, 25, 13, 14]. They are used in diverse fields, e.g., performance analysis of networks [28] and systems biology [27]. The latter will be focus of this paper, since it demands a flexible and efficient architecture for the experimentation with the $\pi$-Calculus. Flexibility is necessary, to facilitate the realization of the many dialects, proposed in the context of systems biology [26, 21, 25, 13, 14]. Efficiency is required, to simulate complex models comprising thousands of molecules and reactions.

The first step in order to achieve flexibility is the clear distinction between model and simulation engine offered by JAMES II. It allows the creation of a general model representation for the desired formalism, while leaving the execution strategy exchangeable. To support the user during the implementation of $\pi$-Calculus dialects, a model representation is required, allowing mappings from those dialects to the model representation. This representation should reflect just the major characteristics of the $\pi$-Calculus in order to avoid restrictions of flexibility, while keeping the need for adaptations for new dialects as small as possible. To structure the simulation engine, essential issues during a simulation run with $\pi$-Calculus dialects have to be identified. This allows the exchange of components for a flexible combination of solutions facing them. Thereby, specific components can be created to face specific characteristics of a $\pi$-Calculus dialect, while other components may be reused.

In addition, a structured simulation engine supports efficiency. It allows the realization of different execution strategies and to adjust simulator configurations to the characteristics of the executed model. Furthermore, the integration of the strategies in one framework supports the direct com-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8820
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8820

parison of their performance. Combined with the less effort for the implementation of new components (compared to the implementation of a full simulator), this facilitates identification of new promising execution strategies.

We propose an architecture for experimentation with the $\pi$-Calculus, providing flexibility and efficiency. It has been realized within the plug-in based modeling and simulation framework JAMES II. Hence, it offers JAMES II's full set of features like flexible creation of experiments [10], and coarse-grained experiment execution [16].

We explain the architecture in two parts. The first part is focused on the model representation for the $\pi$-Calculus based on Java classes. Therefore, we describe the biochemical variant of the $\pi$-Calculus [21] and propose a mapping to the model representation. Furthermore, an outlook to other $\pi$-Calculus dialects is given. Since the focus is on the mapping, the description of the biochemical $\pi$-Calculus will not include the semantics or the abstract simulator. Further information about these aspects can be found in [20] and [21]. The class based structure enhances flexibility by allowing the exchange of components and supporting extension by inheritance. The second part is focused on the simulation engine. Based on the model representation, we present a simulation engine architecture, providing a flexible and extensible combination of simulator components. We realized different implementations for these components, with the aim of creating combinations to allow an efficient simulation. A performance study with two biochemical models shows how components can be evaluated. A comparison to the results of the actual state of the art stochastic $\pi$-Calculus simulator [22] proves the efficiency of our concept. We finish with a short discussion of related work and some concluding remarks.

## 2. BIOCHEMISTRY IN THE PI-CALCULUS

The $\pi$-Calculus was introduced by Milner as a formalism for modeling concurrent systems [20]. In 2001 Regev and Shapiro suggested that, since also molecular solutions are concurrent systems, formalisms from that field in particular the $\pi$-Calculus can be applied to the modeling of biochemistry [27]. Driven by this new application area, a dialect of the $\pi$-Calculus, the biochemical form, evolved, which explicitly regards species and reactions. It adopts the ideas of the stochastic $\pi$-Calculus [24], which introduces stochastic rates, and thus allows for a formal mapping to Continuous Time Markov Chains. By this, based on a stochastic simulator [22] and the Stochastic Simulation Algorithm (SSA) [7], quantitative trajectories reflecting the dynamic behavior of models can be obtained. Since in this paper we focus on the area of computational systems biology, we purely base our investigations on the biochemical form. In the following, we first introduce the syntax of the biochemical form and two example models, which are also used for performance experiments in Section 3.2. Then, we introduce a class based model representation for the biochemical form and provide a general mapping from models to the model representation. It allows to automatically create classes out of textual $\pi$-Calculus models. A corresponding compiler has been implemented in JAMES II.

### 2.1 The Biochemical Form

The main idea of the biochemical form is to omit explicit reaction rules and instead to define processes that describe

| | | | |
|---|---|---|---|
| $D_c$ | $::=$ | $x : r \in R$ | reaction definition |
| $D_s$ | $::=$ | $A(\tilde{x}) \triangleq P$ | species definition |
| $P$ | $::=$ | $(\nu D_c)P$ | new reaction |
| | $\mid$ | $S$ | solution |
| | $\mid$ | $M$ | summation |
| $S$ | $::=$ | $S_1 \mid S_2$ | parallel composition |
| | $\mid$ | $A(\tilde{x})$ | molecule |
| | $\mid$ | $\mathbf{0}$ | empty solution |
| $M$ | $::=$ | $M_1 + M_2$ | choice |
| | $\mid$ | $\pi.P$ | reaction capability |
| $\pi$ | $::=$ | $x?\tilde{y}$ | receive |
| | $\mid$ | $x!\tilde{y}$ | send |

**Figure 1: The biochemical form of the $\pi$-Calculus, with $A \in \textbf{\textit{Proc}}, x, \tilde{x}, \tilde{y} \in \textbf{\textit{Chans}}$, and $R = \mathbb{R}^+ \cup \{\infty\}$.**

species regarding their interaction capabilities [27]. The syntax of the biochemical form is presented in Figure 2.1. It is based on two infinite sets, the set of channel names $x, y \in \mathit{Chans}$ and the set of process names $A \in \mathit{Proc}$, which are used to identify reactions and species, respectively. Reactions are composed by a channel name $\tilde{x} \in \mathit{Chans}$ and a stochastic rate constant $r \in R$, with $R = \mathbb{R}^+ \cup \{\infty\}$, i.e. the rates of reactions can either be positive real numbers or infinite, the latter denoting immediate interactions. A species definition provides a name $A \in \mathit{Proc}$, formal parameters $\tilde{x} \in \mathit{Chans}$, and a process $P$. Actual parameters of a species are the names of reactions it can be evolved in. With process $P$ a species' reaction capabilities are described. Process $(\nu x : r)P$ introduces a new reaction, with scope to $P$, i.e. reaction $x$ can only occur in the context of $P$. Tuples $(\nu \tilde{D}_s)$ denote sequences of $\nu$-operators. Solutions are parallel compositions of molecules. With $\prod_{i=1}^{n} A_i(\tilde{x}_i)$ we denote solutions of $n$ molecules and with $\mathbf{0}$ the empty solution. Summations define an exclusive choice between different reaction capabilities. This means, that a single molecule can choose to take part in exactly one reaction by which it is consumed. With $\sum_{i=1}^{n} \pi_i.S_i$, we refer to a summation with $n$ interaction capabilities. In the biochemical form all reactions are binary, i.e. they have exactly two reactants, a sender and a receiver. Senders can deliver reaction names to receivers, which can be used by the latter for further interaction. Which role in a communication a molecule takes, is up to the modeler. After two molecules interacted, they each proceed with a single solution, providing their successors. The union of both successor solutions defines the set of reaction products. Molecules also have the ability to perform sequences of interactions. More precisely, successors are not solutions but sheer interaction choices without parameters or names.

A program in the biochemical form is a tuple $\Delta = (\Delta_c, \Delta_s, S)$, where $\Delta_c$ is a set of reaction definitions, $\Delta_s$ a set of species definitions, and $S$ a parallel composition, defining the initial solution of the model. A program is considered well-formed if every species has a unique name, for every molecule a species definition exists, and for every species definition $A(\tilde{x}) \triangleq P$ it holds that $P$ only refers to names in $\Delta_c$ or in $\tilde{x}$ and the names in $\Delta_c$ and $\tilde{x}$ are distinct. Thus, two molecules can be considered to be of the same species if both their names and their actual parameters equal. Such notion of equality allows to identify molecules without using the usual structural congruence of the $\pi$-Calculus, which im-

*Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8820*
*http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8820*

**Channels**

```
ion1:10.0, ion2:100.0
dion1:50.0, dion2:5.0
```

**Process definitions**

```
Mg() ≜ ion1?().MgP()
MgP() ≜ ion2?().Mg2P() + dion1?().Mg()
Mg2P() ≜ dion2?().MgP()
Cl() ≜ ion1!().ClM() + ion2!().ClM()
ClM() ≜ dion1!().Cl() + dion2!().Cl()
```

**Initial solution**

$\prod_{i=1}^{1000}$ (Mg() | Cl())

**Figure 2: A model of the Magnesium ionization in the biochemical form.**

plies high computational costs [15] - an idea, which is heavily used in the context of efficient simulators, see e.g. [22]. Notice, however, that, when omitting structural congruence, reaction sequences require special attention, as they introduce successors without explicit names or parameters. We treat this exception in the context of mapping models to our model representation, see Section 2.2.

As already mentioned in Section 1, we omit the formal semantics of the biochemical form here. However, by the following example, we provide an impression on how the calculus works: Consider the solution $A() \mid B()$, with $A() \triangleq x!(z).(\nu x : r)A()$ and $B(y) = x?(y).B(y)$, with channels $x, y, z$ defined in $\Delta_c$. When replacing molecules by their definition, we obtain $x!(z).(\nu x : r)A() \mid x?(y).B(y)$. One interaction on $x$ reduces the prefixes and we obtain $(\nu x : r)A() \mid B(z)$. To ensure that $(\nu x)$ introduces a fresh name, we extend the scope of the $\nu$-operator and perform a renaming on the definition of $A()$, such that we obtain $(\nu x' : r)(x'!().(\nu x' : r)A() \mid x?(z).B(z))$, when replacing molecules by their definition once more. This does not allow for any further interaction.

In the following, we, present two models, that on one hand shall provide a deeper insight into the formalism and on the other hand form the bases of our performance experiments in Section 3.2.

*Example: Magnesium.* Our first example considers a simple model of the ionization processes in a solution with Magnesium and Chlorine, which is characterized by the following two reversible reactions:

$$\mathrm{Mg} + \mathrm{Cl} \underset{50}{\overset{10}{\rightleftharpoons}} \mathrm{Mg}^{+} + \mathrm{Cl}^{-}$$
$$\mathrm{Mg}^{+} + \mathrm{Cl} \underset{5}{\overset{100}{\rightleftharpoons}} \mathrm{Mg}^{2+} + \mathrm{Cl}^{-}$$

Our model is presented in Figure 2. It comprises five species definitions Mg(), MgP(), Mg2P(), Cl(), and ClM() and four reactions with names ion1, ion2, dion1 and dion2 for ionization and de-ionization of Magnesium, respectively. When interacting on ion, Mg() gains an ion and Cl() loses one, as denoted by their respective successors MgP() and ClM(). All other interactions work analogously. The initial solution contains 1000 molecules of Magnesium and Chlorine.

This model is interesting for performance experiments, because of the low amount of channels, while containing many processes. This allows a specific investigation of the influence of many processes in the system to the performance of the simulation engine, since the influence of the channels is negligible.

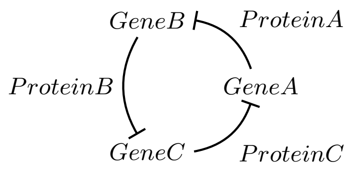

**Figure 3: The *Repressilator* - genes produce proteins inhibiting the protein production of their neighbors.**

**Channels**

```
a:1.0, b:1.0, c:1.0
p:0.1, r:0.0001, d:0.001
```

**Process definitions**

```
Gene(me,pe) ≜ p?().(Prot(pe)|Gene(me,pe))
            + me?().r?().Gene(me,pe)
Prot(ta) ≜ ta!().Prot(ta) + d?().0
T() ≜ p!().T() + r!().T() + d!().T()
```

**Initial solution**

```
Gene(a,b) | Gene(b,c) | Gene(c,a)
```

**Figure 4: A model of the *Repressilator* in the biochemical form.**

*Example: Repressilator.* The *Repressilator* is a biologically inspired model, which represents a prototype example of oscillatory systems [4]. Its main idea is depicted in Figure 3: three genes produce proteins that inhibit the protein production of their neighbors. In order to implement the *Repressilator*, we define two species Gene(me,pe) and Protein(ta). Parameters me and pe represent the inhibition reactions of the gene itself and its peer, respectively. Parameter ta captures the reaction, which inhibits the target of the protein. The three channels a,b, and c identify the actual inhibition reactions involving the three genes. Channels p, r, and d represent the protein production, the re-activation of genes after inhibition, and the decay of proteins, respectively.

Species Gene(me,pe) has two interaction capabilities: it can either produce a protein with its peers pe as target for inhibition by interaction on channel p, or it gets inhibited on channel me. When inhibited it can only be re-activated by interaction on channel r. A protein can either block its target gene by sending on channel ta or decay on channel d. We introduce process T() as an artificial partner for interactions on channels p, r, and d, by this unary reactions are modeled.

The standard variant of this model contains three genes and proteins and six channels. They can be increased very easily, by enlarging the cycle. This allows a scaling of the processes and channels in the system, which is beneficial for investigating the performance of the simulation components with differently sized models.

## 2.2 Model Representation

Our model representation is a simple class hierarchy, designed as an interface between models and simulators. It provides all fields and methods, that are needed by a simulator to process models. A model is represented by classes,

*Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8820*
*http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8820*

which inherit from the model representation, accordingly. The basic idea, is to create classes for all summations, parallel compositions, and reaction capabilities in a model. Furthermore, we represent channels as objects. By this, we introduce a clear distinction between names and channels, i.e. a channel object is a value of a name and bindings assign channel objects to names. The advantage is, that then the $\nu$-operator does not lead to any dynamic name changes, but can be represented by allocating freshly created objects to names. In the following, we first describe the model representation and then the mapping of the biochemical form.

The model representation is presented in Figure 5. Classes **PiChannel** and **StoPiChannel**, represent channels or channels with rates, respectively. The class **BasicConstruct** provides basic fields and functionality, common to all syntactic constructs, apart from channels. For our simulators, it is of advantage to reflect models in a parse-tree-like structure. Thus, we introduce the field parent, which holds in case of e.g. a molecule the solution it is in or in case of a reaction capability the choice it is part of. Class **PiAction** provides the method perform to execute communications and the field channel, reflecting the channel on which communication happens. Classes **PiSend** and **PiReceive** represent reaction capabilities of corresponding type. Processes inherit either from class **PiSummation**, where the field actions lists a choice's reaction capabilities, or class **PiParallel**, with the field processes, which refers to the set of molecules, that a parallel composition comprises. Parameters of classes can be general objects not strictly channels as in standard $\pi$-Calculus processes, which increases the extensibility for other $\pi$-Calculus variants, such as [13, 14].

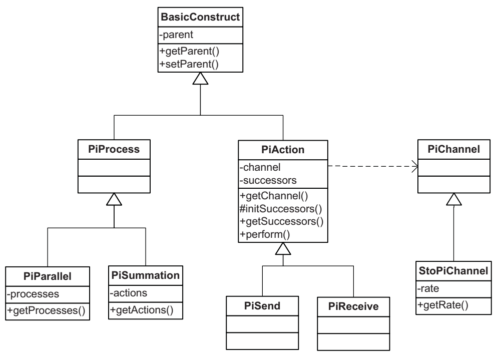

**Figure 5: Class diagram for the model representation.**

The mapping of models is presented in Figure 6. We distinguish between mappings of programs $[\![\cdot]\!]$, species definitions $\mathcal{S}[\![\cdot]\!], \tilde{f}, \kappa$, processes $\mathcal{P}[\![\cdot]\!], \tilde{b}, \tilde{f}, \kappa$, and reaction capabilities $\mathcal{A}[\![\cdot]\!], \tilde{b}, \tilde{f}, \kappa$. Mappings create classes, that inherit from the model representation, correspondingly. Thereby, parameter $\kappa$ reflects the name of a class. In order to obtain unique class names, we use the following convention: classes representing species definitions $A(\tilde{x})$ get the name $A$. Classes representing sends or receives are associated with names $\kappa$_a_$i$, where $\kappa$ is the name of the summation, that contains the action, and $i$ the position of the action in the summation. Classes representing the successors of actions are identified by $\kappa$_Succ, where $\kappa$ is the name of the action. Parameters $\tilde{b}$ and $\tilde{f}$, capture all names accessible by a class, where the first represents the tuple of free names, i.e. all names in $\Delta_c$ of a program and the latter the tuple of bound names, i.e. those names, which are introduced by bindings like $\nu$-operators, species parameters, or receive parameters. When mapping a single program, parameter $\tilde{f}$ is always associated with the same tuple. By contrast, $\tilde{b}$ may need to be extended. Consider e.g. the species $A(x) \triangleq (\nu y : r)x?(x).P$. Initially, we obtain a tuple of bound names $(y)$. However, $P$ can also access $x$, such that for the mapping of $P$, the value of parameter $\tilde{b}$ has to be $(x, y)$.

Our presentation of the mapping follows the idea of meta modeling, i.e. we present pseudo-code to transform programs into Java classes. We make use of the following notations: $\forall e \in E$ denotes an iteration over all elements in $E$. Thereby, $E$ can be either a set or a tuple. Functions **class**, **init**, and **func** generate classes, constructors, and methods, respectively. Function **class** awaits three parameters, the name of the class, the class to inherit from, and a code sequence describing the content of the class. Similarly, function **init** has two parameters, a tuple of parameter names and a function body. Function **func** extends on **init** by an additional parameter for the function name. With **return**, we define the return value of a function. Function **set** creates assignments, where the first parameter represents the variable name and the second the value. Function **new** denotes class instantiation, where the first parameter provides the name of the class and the second a tuple, holding the actual parameters of the instance. Constant **this** represents the self-reference of an object. With **new** T[], we refer to the generation of an array of type T. We denote field c to be an array with c[], while referring to its $i$-th element with c[i]. We omit types, when obvious from the description of the model representation. With $(a_1, \ldots, a_n) \cup (b_1, \ldots, b_m)$, we denote the tuple $(a_1, \ldots, a_n, b_1, \ldots, b_n)$, and with $(a_1, \ldots, a_n) \setminus (b_1, \ldots, b_m)$ the tuple $(a_1, \ldots, a_{n'})$ where for all $a \in \{a_1, \ldots, a_{n'}\}$ it holds that $a \in \{a_1, \ldots, a_n\}$ and $a \notin \{b_1, \ldots, b_m\}$. The operator $a \oplus b$ describes the extension of tuple $a$ by all elements in tuple $b$, which are not in $a$, i.e. it abbreviates $a \cup (b \setminus a)$.

Programs $(\Delta_c, \Delta_s, S)$ are transformed by first mapping all species definitions in $\Delta_s$, with a tuple of all names in $\Delta_c$, and then creating a class Init for the initial solution $S$, which inherits from **PiParallel**. The constructor of Init is implemented to introduce channels according to the definitions in $\Delta_c$ and to assign molecules to the array processes according to the initial solution $S$. Thereby, to create molecule $A(\tilde{y})$, the class defining species $A(\tilde{x})$ is instantiated with the values of the names $\tilde{y}$, the channels in $\Delta_c$, and **this**, the latter denoting a molecule's parent. As the root of a model, the initial solution has no parent, such that the parameter p of the constructor is ignored.

The mapping of species definitions $A(\tilde{x}) \triangleq P$ is rather indirect by mapping process $P$ with bound names $\tilde{x}$ and name $A$. The idea is that, conceptually, $A(\tilde{x})$ only provides a mapping from actual parameters to formal ones, which can be directly implemented in the constructor of a class representing $P$.

Processes preceded by a sequence of $\nu$-operators are mapped by deriving either **PiParallel** or **PiSummation** according to the process type. In case of a summation, classes are created for each contained reaction capability. Thereby, as discussed above, the list of bound names is extended by the newly introduced names in the definitions of the $\nu$-operator. In case of a parallel composition, classes for the contained molecules do not need to be created, since their species are already mapped as discussed above. For either process type, the constructor is implemented to provide parameters for the values of the bound and free names $\tilde{b} \cup \tilde{f}$ and a parent p, the latter being directly assigned to field parent. Variables are created for all those names, which are newly introduced by the definitions of the $\nu$-operator, analogously to the extension of $\tilde{b}$. Furthermore, for each channel definition of the $\nu$-operator, a new channel object is created with the corresponding rate and assigned to the corresponding name. In case of mapping a parallel composition, molecules are created and assigned to the array processes, in the same way as done in the initial solution. Similarly, for a summation, the instances of the corresponding action classes are generated and assigned to the array actions.

A send action is mapped by first transforming its successor process, with the same bound and free names as the action and an identifier following our naming convention, and then creating a class, which inherits from **PiSend**. The latter contains fields for all accessible names $\tilde{b} \cup \tilde{f}$ and also a constructor with a parent p and names $\tilde{b} \cup \tilde{f}$ as formal parameters. The parameters are assigned to the corresponding fields and the channel on which the sending is performed is set. Function perform is implemented to instantiate and assign the successor and to return the channels, which are to deliver on interaction. Since no channels are received, parameter cs [] of perform is ignored. Receive actions are implemented similarly, but due to the receive parameters, the tuple of bound names is extended. Function perform is implemented to assign the received channels objects to the names in $\tilde{y}$. The instantiation of the class of the successor process is adapted accordingly. Notice, that by generally mapping successors to processes, we automatically solve the problem of action sequences. Consider e.g. the species definition $A(\tilde{x}) \triangleq x!().y!()$. For the successor of the first action, we create class A_a_1_Succ, with access to the same names as class $A$, representing $A(\tilde{x})$.

*Mapping of variants.* The formalism SpacePi extends on the $\pi$-Calculus by associating processes and channels with spatial information [13]. A mapping to our model representation has been realized by encoding the spatial information into the model classes. This includes position and movement information for the extensions of **PiSummation** and **PiParallel**, as well as a range for PiAction to denote the sphere of influence of an action.

Beta-binders [25] introduces a new layer into the $\pi$-Calculus by wrapping $\pi$-Calculus processes into boxes (*BioProcesses*). For a mapping of Beta-binders to our model representation, new classes had to be introduced, to represent the BioProcesses, holding references to the internal $\pi$-Calculus processes, as well as to the external interface, see [18]. Additional abstract classes for the representation of *hide*, *unhide*, and *expose* actions had to be introduced as well.

```
$[\![(\{x_1 : r_1, \dots, x_n : r_n\}, \Delta_s, \prod_{i=1}^n A_i(\tilde{x}_i))]\!] \triangleq$

 $\forall D \in \Delta_s$  $\mathcal{S}[\![D]\!], (x_1, \dots, x_n)$
 class Init PiParallel
  init (p)
   $\forall x \in (x_1, \dots, x_n)$ var $x$ (new StoPiChannel ($r$))
   $\forall i \in \{1, \dots, n\}$ set processes[$i$] (
    new $A_i$ $\tilde{x}_i \cup (x_1, \dots, x_n) \cup$ (this))

$\mathcal{S}[\![A(\tilde{x}) \triangleq P]\!], \tilde{f}, \kappa \triangleq \mathcal{P}[\![P]\!], \tilde{x}, \tilde{f}, A$

$\mathcal{P}[\![(\nu\Delta_c) \prod_{i=1}^n A_i(\tilde{x}_i)]\!], \tilde{b}, \tilde{f}, \kappa \triangleq$

 class $\kappa$ PiParallel
  init $\tilde{b} \cup \tilde{f} \cup$(p)
   set parent p
   $\forall x \in (x_1, \dots, x_n) \setminus (\tilde{b} \cup \tilde{f}), \Delta_c = (x_1 : r_1, \dots, x_n : r_n)$ var $x$
   $\forall x : r \in \Delta_c$ set $x$ (new StoPiChannel ($r$))
   $\forall i \in \{1, \dots, n\}$ set processes[$i$] (
    new $A_i$ $\tilde{x}_i \cup \tilde{f} \cup$ (this))

$\mathcal{P}[\![(\nu(x_1 : r_1, \dots, x_m : r_m)) \sum_{i=1}^n \pi_i.P_i]\!], \tilde{b}, \tilde{f}, \kappa \triangleq$

 $\forall i \in \{1, \dots, n\}$  $[\![\pi_i.P_i]\!], \tilde{b} \oplus (x_1, \dots, x_m), \tilde{f}$, $\kappa$_a_$i$
 class $\kappa$ PiSummation
  init $\tilde{x} \cup$(p)
   set parent p
   $\forall x \in (x_1, \dots, x_m) \setminus (\tilde{b} \cup \tilde{f}), \Delta_c =$ var $x$
   $\forall x : r \in \Delta_c$ set $x$ (new StoPiChannel ($r$))
   $\forall i \in \{1, \dots, n\}$ set actions[$i$] (
    new $\kappa$_a_$i$ $\tilde{b} \oplus (x_1, \dots, x_m) \cup \tilde{f} \cup$(this))

$\mathcal{A}[\![x!\tilde{y}.P]\!], \tilde{b}, \tilde{f}, \kappa \triangleq$

 $\mathcal{P}[\![P]\!], \tilde{b}, \tilde{f}, \kappa$_Succ
 class $\kappa$ PiSend
  $\forall x \in \tilde{b} \cup \tilde{f}$ var $x$
  init $\tilde{b} \cup \tilde{f} \cup$(p)
   set parent p
   $\forall x \in \tilde{b} \cup \tilde{f}$ set this.$x$ $x$
   set channel $x$
  func perform cs[]
   set successors (new $\kappa$_Succ $\tilde{b} \cup \tilde{f}$)
   return new StoPiChannel[] $\tilde{x}$

$\mathcal{A}[\![x?\tilde{y}.P]\!], \tilde{b}, \tilde{f}, \kappa \triangleq$

 $\mathcal{P}[\![P]\!], \tilde{b} \oplus \tilde{y}, \tilde{f}$ $\kappa$_Succ
 class $\kappa$ PiReceive
  $\forall x \in \tilde{b} \oplus \tilde{y} \cup \tilde{f}$ var $x$
  init $\tilde{b} \cup \tilde{f} \cup$(p)
   set channel $x$
   $\forall x \in \tilde{b} \cup \tilde{f}$ set this.$x$ $x$
  func perform cs[]
   $\forall i \in \{1, \dots, n\}, \tilde{y} = \{y_1 \dots, y_n\}$ set $y_i$ cs[$i$]
   set successors (new $\kappa$_Succ $\tilde{b} \oplus \tilde{y} \cup \tilde{f}$)
```

**Figure 6: Mapping the biochemical form to the model representation.**

## 3. SIMULATION OF THE PI-CALCULUS

Due to the attention of the $\pi$-Calculus in the field of systems biology, different simulation engines [2, 21, 22] have been proposed. All of them execute two different tasks, in each simulation step. Firstly, the type of the next reaction (i.e., the next channel on which two processes will communicate) has to be calculated. Secondly, according to this reaction, the communication has to be executed and required updates on the model structure have to be performed. While various approaches for handling both tasks of a $\pi$-Calculus simulation engine exist [7, 8, 6, 21, 22], so far the implemented solutions just focused on realizing fixed combinations of components. However, simulator components used during a simulation run can influence the results of the simulation experiment. Bias can result from a sloppy implementation, from differences in the accuracy, from bugs, or from unexpected behaviour due to the interaction of components. These problems can hamper the performance or influence the validity of the simulation. Hence, a flexible combination of solutions for the two tasks is required, allowing reusability, extensibility, and exchangability. Therefore, we created a flexible and extensible structure for implementing accordant simulation engines and components, The concrete components have been realized as plug-ins of JAMES II. For a further increase of flexibility, existing plug-ins facing general issues of simulation, e.g., random number generators [5] or event-queues [11] can be reused, extended, or exchanged as well. The opportunities to configure experiments in JAMES II like flexible instrumentation and observation of simulations, coarse-grained parallel execution, or dynamic calculation of required replications and simulation end times are applicable for experiments with the $\pi$-Calculus as well.

## 3.1 Simulation Components

Each $\pi$-Calculus simulator is based on *BasicPiProcessor*. To handle the first phase, which highly depends on the reaction method (e.g., Gillespie's method), the BasicPiProcessor holds a reference to an object implementing the interface *IReaction*. The interface offers the method *react*, which is called by the simulator to calculate the next communicating channel. Thereby, different implementations of IReaction can be combined with different simulators. Based on the next channel, the simulator randomly selects a pair of actions communicating on that channel.

The implementation of the second phase is realized in the extensions of BasicPiProcessor. This includes performing the actions and creating the successor processes as described in Section 2. Furthermore, updates have to be executed on the model structure and on the information required to select the communicating actions in the future simulation step.

So far we implemented three simulation engines for the $\pi$-Calculus as well as three reaction methods. They work on the model representation introduced in section 2. We will give a brief overview over those components and discuss how they have have been adapted to simulate further $\pi$-Calculus dialects.

### *Communication Based Simulation.*

**Algorithm 1** Pseudocode for the execution of a simulation step with the ComFS.

```
communication_based(PiParallel proc,    (PiChannel
    , PiSend, PiReceive)[] map) {
  (PiChannel, int)[] c = communicationCounts(map);
  (PiSend, PiReceive) pair = getCommunication(c);
  performCommunication(pair);
  updateModel(process, first(pair));
  updateModel(process, second(pair));
  PiSend[] senders = newSenders(pair);
  PiReceive[] receivers = newReceivers(pair);
  put(map, senders $\times_{channel}$ getReceivers(map));
  put(map, receivers $\times_{channel}$ getSenders(map));
  put(map, senders $\times_{channel}$ receivers);
}
```

The communication focused simulator (ComFS) maintains a map where the pairs of actions able to communicate are stored, sorted by their channel (see Algorithm 1). During a simulation step, the counts of the possible communications for each channel are retrieved and used as input

**Algorithm 2** Pseudocode for the individual based update of the model structure after a communication happened.

```
updateModel(PiParallel process, PiAction action) {
  PiProcess succ = action.getSuccessor;
  succ.setParent(process);
  process.getProcesses.add(succ);
  process.getProcesses.remove(action.getParent);
}
```

for the reaction method. Thereby the actual communication pair is selected, and executed. The model structure is updated (see Algorithm 2) according to the actions contained in the communication pair. This part is done individual based, by adding the successor processes to and removing the parent processes of the actions from the top most parallel process. To update the map of communication pairs, the pairs comprising new sender and retriever actions have to be created. This is done, by calculating the cross product of the new senders and retrievers with the corresponding actions that use the same channel. Since the creation of the pairs demands to associate each possible sender with each possible receiver, the algorithm has to tackle a combinatorial explosion. However, despite those efficiency issues the simulator showed the benefits of its design during the creation of a simulator for the space $\pi$-Calculus [17].

### *Channel Based Simulation.*

**Algorithm 3** Pseudocode for the execution of a simulation step with the ChanFS.

```
channel_based(PiParallel process,
    (PiChannel, int, int, int)[] map) {
  (PiChannel, int)[] c = communicationCounts(map);
  (PiSend, PiReceive) pair = getCommunication(c);
  performCommunication(pair);
  updateModel(process, first(pair));
  updateModel(process, second(pair));
  PiSend[] senders = newSenders(pair);
  PiReceive[] receivers = newReceivers(pair);
  for (PiAction action $\in$ (senders $\cap$ receivers))
    add_action(action, map);
}
```

**Algorithm 4** Pseudocode for the update of the inputs, counts, and mixes for the ChanFS.

```
add_action(PiAction action,
    (PiChannel, int, int, int)[] map) {
  int mix = getMixOnChannel(map, c);
  int outs = getOuputOnChannel(map, c)
  int ins = getInputOnChannel(map, c)
  PiProcess parent = action.getParent
  if (action instanceof PiSend) {
    outputs = outputs + 1;
    mix = mix + getReceiversOnChannel(parent, c);
  } else {
    inputs = inputs + 1;
    mix = mix + getSendersOnChannel(parent, c);
  }
  put(map, (action.getChannel, ins, outs, mix);
}
```

The channel focused simulator (ChanFS) implements the management of channel information proposed by SPiM [21], to avoid the combinatorial explosion (see Algorithm 3). In

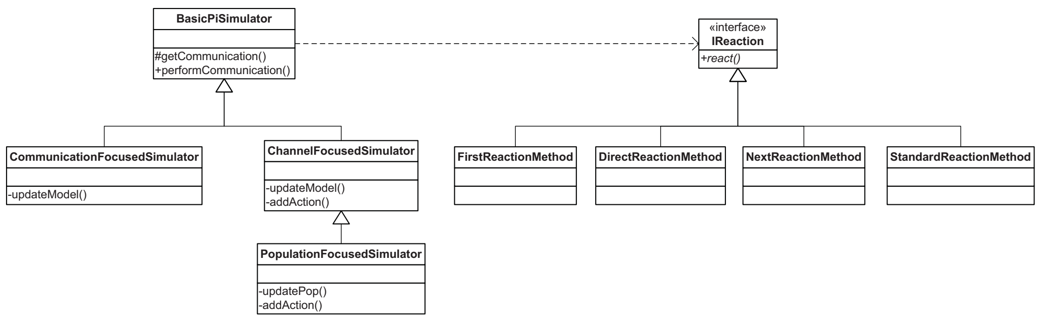

**Figure 7: Class diagram for the simulation engines.**

contrast to storing the communication pairs, now the channel map just contains three information about each channel. These information are the count of receivers on the channel (*input*), the count of senders on the channel (*ouput*), and the count of communication pairs which are part of the same summation (*mix*). The count of possible communicating pairs on a channel can be calculated by $input * output - mix$. The selection and execution of the communication, as well as the update of the model structure is done similar to the ComFS. To update the map, instead of creating each communication pair, just the *inputs*, *ouputs*, and *mixes* have to be recalculated (see Algorithm 4), which avoids the expensive cross product. Thereby, the algorithm postpones the handling of the communication pairs until they have been selected to be the next communication. Hence, no unnecessary pairs are created. This simulator has been used as the base for the *Simulator* component of a distributed Betabinders simulator [18].

### *Population Based Simulation.*

**Algorithm 5** Pseudocode for the execution of a simulation step with the PopFS.

```
population_based((PiProcess, int)[] population
        (PiChannel, int, int, int)[] map) {
  (PiChannel, int)[] c = communicationCounts(map);
  (PiSend, PiReceive) pair = getCommunication(c);
  performCommunication(pair);
  updatePop(population, first(pair));
  updatePop(population, second(pair));
  PiSend[] senders = newSenders(pair);
  PiReceive[] receivers = newReceivers(pair);
  for (PiAction action ∈ (senders ∩ receivers))
    add_action(action, population, map);
}
```

The population focused simulator (PopFS) is inspired by the latest version of SPiM [22]. It exploits the structure of $\pi$-Calculus models comprising more than one process of the same species (see Section 2.1) to execute those *populations* of processes more efficiently (see Algorithm 5). It is an extension of the ChanFS and uses a map containing a representative for each set of structural congruent processes as well as the corresponding count. A translation from the model to this map and vice versa, is possible. Hence, only the representatives have to be handled, which reduces the required operations during the update of the model structure (see Algorithm 6). In the worst case, if no congruent processes exist, the algorithm works similar to the ChanFS, by adding and removing processes from the population. Otherwise, just the corresponding numbers of the processes are manipulated, which reduces computation time. Furthermore, the updates of *inputs*, *outputs*, and *mixes* is less time-consuming, since the accordant methods do not have to be called for each single action. By considering the corresponding count, only actions contained in the representative summations, have to be handled.

**Algorithm 6** Pseudocode for the population based update of the model structure after a communication happened.

```
updatePop((PiProcess, int)[] population, PiAction
    action) {
  PiProcess succ = action.getSuccessor;
  if (contains(population, succ))
    increase(population, succ);
  else
    add(population, succ);
  PiProcess parent = action.getParent;
  if (count(population, parent) = 0)
    remove(population, succ);
  else
    decrease(population, parent);
}
```

### *Reaction Methods.*

To realize reaction methods we implemented three variants of Gillespie's algorithm [7] using the notion of the channel activity to stochastically select the channel on which the next reaction occurs. The first reaction method (FRM), calculates the delay for each channel and returns the channel with the minimum delay [7]. The higher the amount of channels, the less efficient is this method, since the whole list of channels has to be iterated and for each channel a random number has to be calculated. The direct reaction method (DRM) tries to reduce the amount of random number calculations [8]. Thereby, only two random numbers have to be taken, independently from the count of channels. The next reaction method (NRM) reduces the calculation of random numbers by storing the channels and their time of next event in an event-queue to avoid unnecessary delay recalculations [6]. The performance of this method depends on the used event-queue implementation.

### *Extensibility.*

While the option to encode additional information into the model classes is useful to realize different $\pi$-Calculus dialects, sometimes alterations of the simulators operating on the classes are required as well. The plug-in based structure of the simulator allows exchanging components, to reflect the specific needs of those dialects, while other components may be reused if no additional adaptations are required.

For the simulation of the space $\pi$-Calculus [13], an additional method to trigger the movements of the processes was required for the simulator. In addition, the component to calculate the next reacting channel has been exchanged by a collision detection, which profits from the format of handling communication pairs, provided by the ComFS. It could be integrated very easily by iterating the list of communication pairs and calculating the distances between them.

For the simulation of Beta-binders [25], the structure of the simulation engines had to be adapted. A new component type the *Coordinator* was introduced, to organize the interaction between the BioProcesses, while *Simulator* components are responsible for the execution of the internals of BioProcesses. Since those internals are based on $\pi$-Calculus processes, just some minor adaptations of the ChanFS were necessary to realize the Simulator component. Those adaptations included interfaces for the communication with the Coordinator and the handling of additional action types.

## 3.2 Experiments with the pi-Calculus

A central account in literature on computational experimentation is the handling of side effects [1, 19]. To understand models and algorithms it is essential to investigate the effects of variable factors on the overall behavior. The presented architecture offers the opportunity to face this problem systematically, by using its flexible structure. It is possible to exchange parts of the simulation engine to identify the influences of the different components step by step. Furthermore, all these components are brought together in one framework which improves comparability.

We executed performance experiments in order to show how different components can be evaluated using the presented simulator architecture. In addition to the 3 simulators and the 3 different reaction types, at the moment 13 event-queues and 7 random number generator implementations exist for JAMES II. Regarding the fact that an event-queue is only necessary for the NRM, this leads to 315 possible configurations to test, specific parameters for the different components not considered. Since the analysis of such an experiment would exceed the size of this paper we tested a smaller selection of the components. During the experiments we executed configurations using all three simulators and reaction methods as well as three event-queues, including a simple list implementation and furthermore two more sophisticated concepts, the MList [9] and the calendar queue [3]. Since we were not interested in the simulation output results, bias from a random number generator was negligible. Hence, we just used one random number generator, a Mersenne Twister implementation. To prove that the proposed architecture does not hamper the efficiency of the simulators, we compared our performance results with results produced by the latest version (0.05) of SPiM [22].

The experiments have been executed on an Intel Xeon 8 core processor with 2.5 GHz, 8 GB of RAM, running Ubuntu 9.04 and the Java JDK 6.0 build 13. Scimark [23] produced a result of 483.6 Megaflops on this machine. For confidential results, we replicated each simulation run 10 times.

As benchmark, we used the two models described in Section 2. From the modeling point of view, the those benchmark models may seem too simple. However, to models for simulation benchmarks different criteria apply. For comparability in each simulation run a similar amount of calculation steps is needed, which can only be ensured for rather simple models with predictable behavior. Reproducibility of benchmark tests is also supported by simple models. Furthermore, we chose models that challenge the simulators by large amounts of either channels (Repressilator) or processes ($MgCl_2$). Both models have been validated in [18].

### *Results.*

Throughout all of the experiments, the simulator configurations using the PopFS have been the most efficient simulator configurations of JAMES II (see Figure 8). For the reaction method components, the picture is not as definite. We skip a detailed analysis of the experiments with the NRM, since their performance constantly has been slightly behind the corresponding experiments with the DRM, with negligible differences for the different event-queues. The reason for this behaviour lies in the similar aim of the two methods, which is the reduction of iterated channels during the calculation of the next event. Due to the similar aim, the impact of both methods on the simulation's performance is similar as well, while the overhead in our NRM implementation is slightly higher than in our DRM implementation. However, notable differences exist in the performance of FRM and DRM.

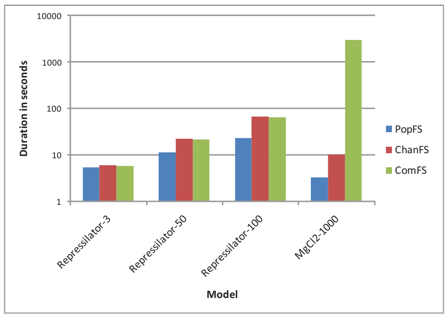

**Figure 8: Execution times of the simulation engines of james ii, all using the DRM (examplarily for other the methods). Note the logarithmic scale of the duration axis.**

The Repressilator model with 3 genes, has been executed for 500,000 seconds of simulation time. The exchange of the reaction method has a different impact on the performance results in connection with different simulators. By switching from the FRM to the DRM the PopFS got a speedup of 1.2 percent, while the ComFS was 13 percent and the ChanFS even was 20 percent faster. A reason for this discrepancy lies in the different format of information the methods retrieve from the simulators. The DRM just requires information about those channels where change happened during the last simulation step, while the FRM requires information about all channels able to communicate. The PopFS removes channels with zero communication from its internal map, to save memory. The other two simulators keep the information about those channels to speed up future updates.

*Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8820*
*http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8820*

This results in an increased list of channels that have to be iterated by the FRM, hampering its efficiency. Improvements of the simulation engines should include both options (keeping or skipping channels) for all three simulator types.

A similar picture arose from the experiments with the Repressilator model with 50 genes, which has been executed for 50,000 seconds of simulation time. The differences between the reaction methods have been increased. Even with the PopFS a considerable speedup of 22.4 percent could be detected, while the speedup with the other two simulation engines was over 300 percent. The reason for the increased speedup result lies in the higher amount of channels that had to be handled during the execution of the model with 50 genes, leading to a larger list to be iterated by the FRM. A further interesting result is the significantly better performance of the PopFS compared to the other two simulator implementations (see Figure 8). This result is caused by the population based approach, playing out its advantages with a model comprising more processes.

The Repressilator model with 100 genes, has been executed for 50,000 seconds of simulation time as well. The results from the previous experiments are confirmed. The difference between FRM and DRM is even higher (33.6 percent with the PopFS, over 450 percent with the other two simulators) with more channels in the model. The PopFS is even faster (at least 3 times) than the other two simulation engines, which results from the higher amount of processes.

The $MgCl_2$ model has been executed for 0.04 seconds of simulation time. The FRM has been the most efficient reaction method, during this experiment. During the executions with the PopFS, the method has been 8 percent faster than the DRM. With the other simulators this difference was not as high, which results from the worse performance of the simulation engines and therefore, the lesser impact of the reaction method on the simulation duration. The reason for the higher efficiency of the FRM lies in the low amount of channels in the model. Thereby, the part of channels where a change happens in a simulation step is very small as well. Thus, the DRM (or NRM) cannot compensate the overhead required for its strategy to avoid the iteration of all channels. The experiments with the $MgCl_2$ strengthen the observation, that a high amount of processes in the model can have a very high impact on the performance of the simulation engines. The PopFS was nearly 3 times faster than the ChanFS and nearly a 1,000 times faster than the ComFS (see Figure 8). The inefficiency of the latter results from the combinatorial explosion to create up to 1,000,000 communication pairs.

Figure 9 shows the performance of the PopFS using the DRM compared to the results produced by SPiM. During the execution of the Repressilator model with 3 genes, SPiM was 33.5 percent faster. This result reversed, as the complexity of the model has been increased by adding more processes and channels. Executing the Repressilator with 50 genes, JAMES II was 12.5 percent faster than SPiM and with 100 genes the difference raise to 31.9 percent. Apparently, the PopFS using the DRM gains efficiency compared to SPiM as the count of channels in the simulated model is increased. This may be explained by the implementation of the DRM for SPiM, where all channel rates are iterated to calculate the sum of propensities (see [21]). The JAMES II implementation avoids this full iteration by keeping a reference to this sum and updating it in each step. During the executions of the $MgCl_2$ model, the PopFS has been over 3 times faster than SPiM (see Figure 9). Even the ChanFS was slightly more efficient. This result is a surprising result, since SPiM as well follows a population based approach, which should be an advantage for the simulation of the $MgCl_2$ model. Probably, there is a feature in the structure of the model that is more beneficial for the technical details of JAMES II than those of SPiM. A profiling of the simulation engines could give hints about that feature. However, those experiments would exceed the size of the paper and thus have to be postponed to a possibly interesting future discussion on the similarities and differences of SPiM and our implementation.

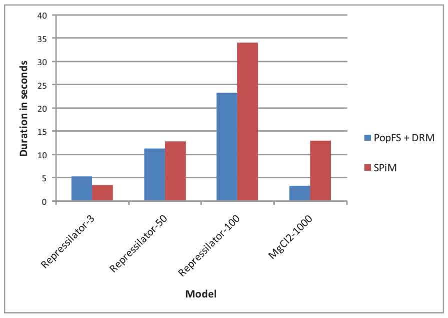

**Figure 9: Execution times of the PopFS using the DRM, compared to those of spim.**

All in all, we can draw three conclusions from the experiments. The PopFS has been the most efficient simulator implementation for the $\pi$-Calculus in JAMES II. It is not as definite, which reaction method is the most efficient. However, a rule of thumb would be, that the more channels there are in the system, the better the DRM is. Finally, during the experiments it turned out, that SPiM has been very efficient with the small Repressilator model, while the best JAMES II configuration became more efficient, as the complexity of the model (processes and channels) has been increased.

## 4. RELATED WORK

Bloch et. al. [2] presented a stochastic $\pi$-Calculus simulator with a divided model structure. One part is responsible for the static and one part for the dynamic information. The static part basically contains the same information as the model representation in the approach we proposed. The simulator is basically used to calculate the reactions. Since the dynamic information are hard-coded in the model, an exchange of different execution strategies for the same model as described in Section 3.1 is not possible.

The variants of SPiM provided the main inspiration for the ChanFS [21] and the PopFS [22]. However, it does not support the flexibility of exchanging and extending simulator components. In the latest version it realizes a simulator similar to the population focused variant presented in section 3.1, using the DRM. Furthermore, it has been implemented with the functional programming language Ocaml, which led to a different model representation based on lists.

## 5. CONCLUSION AND OUTLOOK

We introduced an architecture for experimentation with the $\pi$-Calculus, realized within the plug-in based modeling and simulation framework JAMES II. The focus of the architecture is flexibility and efficiency, to face challenges like handling many different dialects and simulating complex models, that exist especially in the field of systems biology. We realized flexibility by using a class based model representation, allowing extension and forming an interface between the textual representation of the $\pi$-Calculus and a simulator. Furthermore, we introduced a structure for the simulation engine, allowing the reuse and exchange of components. Efficiency is supported by the opportunity to combine promising components.

We conducted performance experiments and showed, that combinations of components exist which execute the used benchamrk models faster than the current state of the art $\pi$-Calculus simulator SPiM.

Further improvements of efficiency could be realized, by new components.The flexibility of the simulator structure could be increased by separating the handling of model structure (population- or individual-based) and the management of communicating actions (channel- or communication-based). So far, both tasks are hardwired in the specific simulation engine, but could be divided to allow a flexible combination and to facilitate the implementation of new solutions.

## 6. ACKNOWLEDGMENTS

This research is sponsored by the German research foundation (DFG).

## 7. REFERENCES

[1] S. Balakirsky and T. R. Kramer. Comparing algorithms: Rules of thumb and an example. In *Performance Metrics for Intelligent Systems*, 2004.

[2] A. Bloch, B. Haagensen, M. K. Høyer, and S. U. Knudsen. The StoPi-calculus and simulator - a stochastic pi-calculus and the implementation of a simulator. Technical report, University of Aalborg, 2003.

[3] R. Brown. Calendar queues: a fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31:1220–1227, 1988.

[4] M. B. Elowitz and S. Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403:335–338, 2000.

[5] R. Ewald, J. Rössel, J. Himmelspach, and A. M. Uhrmacher. A plug-in - based architecture for random number generation in simulation systems. In *WSC*, 2008.

[6] M. Gibson and J. Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *J. Chem. Physics*, 104:1876–1889, 2000.

[7] D. T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *The Journal of Physical Chemistry*, 22, 1976.

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

[9] R. S. M. Goh and I. L.-J. Thng. Mlist: An efficient pending event set structure for discrete event simulation. *International Journal of SIMULATION*, 4:66–77, 2003.

[10] J. Himmelspach, R. Ewald, and A. M. Uhrmacher. A flexible and scalable experimentation layer for JAMES II. In *WSC*, pages 827–835, 2008.

[11] J. Himmelspach and A. M. Uhrmacher. The event queue problem and pdevs. In *Proceedings of the SpringSim '07, DEVS Integrative M&S Symposium*, pages 257–264. SCS, 2007.

[12] J. Himmelspach and A. M. Uhrmacher. Plug'n simulate. In *Proceedings of the 40th Annual Simulation Symposium*, pages 137–143. IEEE Computer Society, 2007.

[13] M. John, R. Ewald, and A. M. Uhrmacher. A spatial extension to the pi calculus. In *Electronic Notes in Theoretical Computer Science*, volume 194, pages 133–148, 2008.

[14] M. John, C. Lhoussaine, J. Niehren, and A. Uhrmacher. The attributed pi calculus. In *CMSB*, pages 83–102, 2008.

[15] V. Khomenko and R. Meyer. Checking Pi Calculus Structural Congruence is Graph Isomorphism Complete. Technical Report CS-TR: 1100, School of Computing Science, Newcastle University, 2008.

[16] S. Leye, J. Himmelspach, M. Jeschke, R. Ewald, and A. M. Uhrmacher. A grid-inspired mechanism for coarse-grained experiment execution. In *DSRT*, pages 7–16, 2008.

[17] S. Leye, M. Jeschke, and M. John. The spacepi simulator, 2007.

[18] S. Leye, C. Priami, and A. M. Uhrmacher. A bounded-optimistic, parallel beta-binders simulator. In *DSRT*, pages 139–148, 2008.

[19] C. C. McGeoch. Experimental analysis of algorithms. *Notices of the AMS*, 48:304–311, 2001.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Inf. Comput.*, 100(1):1–77, 1992.

[21] A. Phillips and L. Cardelli. A correct abstract machine for the stochastic pi-calculus. *Electronic Notes in Theoretical Computer Science*, 2004.

[22] A. Phillips and L. Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *CMSB*, pages 184–199, 2007.

[23] R. Pozo and B. Miller. Java scimark: http://math.nist.gov/scimark2/.

[24] C. Priami. Stochastic $\pi$-calculus. *The Computer Journal*, 38/7:578–589, 1995.

[25] C. Priami and P. Quaglia. Beta-binders for biological interactions. In *CMSB*, 2004.

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

[27] A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochimcal processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, volume 6, pages 459–470, 2001.

[28] Z. Y. Xia, Y. Zhong, and S. Zhang. Analysis for active network security abased on pi-calculus model. *Computer Networks and Mobile Computing*, 0:366, 2003.