Parallel and Pseudorandom Discrete Event System Specification Vs. Networks of Spiking Neurons: Formalization and Preliminary Implementation Results

Alexandre Muzy
CNRS I3S UMR 7271
06903 Sophia-Antipolis Cedex, France.
Email: alexandre.muzy@cnrs.fr

Matthieu Lerasle, Franck Grammont
Univ. Nice Sophia Antipolis
CNRS LJAD UMR 7351
06100 Nice, France.

Van Toan Dao, David RC Hill
ISIMA/LIMOS UMR CNRS 6158
Blaise Pascal University
BP. 10125, 63173 AUBIERE Cedex, France.

***Abstract*—Usual Parallel Discrete Event System Specification (P-DEVS) allows specifying systems from modeling to simulation. However, the framework does not incorporate parallel and stochastic simulations. This work intends to extend P-DEVS to parallel simulations and pseudorandom number generators in the context of a spiking neural network. The discrete event specification presented here makes explicit and centralized the parallel computation of events as well as their routing, making further implementations easier. It is then expected to dispose of a well defined mathematical and computational framework to deal with networks of spiking neurons.**

***Keywords*—Spiking neuron networks, discrete event system specification, pseudorandomness, parallel simulation, multi-threading.**

I. Introduction

Discrete events allow faithfully implementing spike exchanges between biological neurons. Discrete event spiking neurons have been widely implemented in several software environments [5]. However, as far as we know, there is no attempt to embed these works into a common mathematical framework that would allow further theoretical and practical developments between biology and computer science. To achieve this goal the Parallel Discrete Event System Specification (P-DEVS) [3], [23] is used here. This framework provides well defined structures for the formal and computational specifications of a general dynamic system structure.

From the neuronal nets perspective, DEVS has been used mainly for: the proposition of original neuron models [21], the specification of dynamic structure neurons [19], the abstraction of neural nets [22] and for the specification of continuous spike models [13]. Previous works do not consider explicitly spiking neurons in the context of DEVS parallel and stochastic modeling and simulation. *From a parallel and distributed simulation perspective*, the completeness of DEVS (germinating from mathematical proofs until simulator architectures) seems to bias researches either to formal aspects [7] or to simulator structures integrating both parallel and distributed aspects [1]. Furthermore, although the parallel occurrence of discrete events has been formally tackled [7], there are few solutions dealing exclusively with parallelism at abstract simulator level (as in [20]) and no work at network level. Modeling spiking neurons often requires stochasticity. The use of stochasticity at simulation level implies using good practices dealing with the parallelization and distribution of random streams [9]. While pseudorandom generators have been formalized in [23], the definition is generalized here and extended to parallel random streams and random graphs. Except in DEVS, discrete events have been used successfully at each level of modeling and simulation of neuronal nets: at modeling level [18], [5], simulation level [10], [17], [5], [15], and at hardware level with, e.g., the new IBM neurosynaptic chips improving both energy consumption [14] and computation times [4]. The latter hardware developments raise the question of the usage pertinence of general-purpose computers (based on Von Neumann architecture) for the parallel simulation of neural nets.

The inherent nature of a neuronal spiking models leads to a parallel implementation of the simulation. Parallel distributed simulation (PADS) has been extensively studied for various applications domains [16]. In this active research field, optimistic time management was introduced a long time ago by Jefferson's team [11] with some modern implementations at Georgia Tech (Georgia Tech Time Warp). The approach proposed here retains a *conservative approach with a thread programming model*.

However, P-DEVS cannot be used as it is. In the context of networks of spiking neurons, using P-DEVS requires first of all the development of:

- New (general) formal structures to capture explicitly and unambiguously the parallel and stochastic aspects of spiking neural networks.
- New simple and abstract algorithms for the parallelization of discrete event computations and exchanges.

Although this work is just a first step, our goal is to provide a full (hierarchical) formalization from models to implementations. With such a framework, it will be for example possible

978-1-5090-2088-1/16/$31.00 ©2016 IEEE

to compare mathematically different hardware solutions (see e.g. [24] for such modeling analysis). Also, from an application perspective, any dynamic system based on (random) graphs (agents, computer networks, gene networks, etc.) is a further potential application of this work.

This study aims at answering the following questions: What are the main computational loops to be parallelized in a DEVS simulator? How to manage rigorously the stochastic aspects of these parallel simulations? What are the corresponding mathematical structures? How can these elements be used to simulate networks of spiking neurons?

More precisely, we propose here:

- A formalization of networks as parallel discrete event system specifications using pseudorandom generators for the generation of stochastic trajectories and network structures,
- A simple parallelization technique of events in P-DEVS simulators,
- An application of all these concepts to random networks of spiking neurons.

The manuscript is organized as follows. In section 2, the formal model, simulator and executor algorithms are presented. In section 3, stochastic, parallel and pseudorandom generator structures are defined. In section 4, a spiking neuronal network model and its discrete event system specification are presented. In section 5, simulation process and results are presented and discussed. Finally, conclusion and perspectives are provided.

II. Modeling and simulation framework

The architecture for modeling, simulation and execution is an extension of the usual model/simulator separation [3] to hardware interfacing through an *executor* entity. The architecture consists of: (i) the *model*, which specifies the elements of the dynamic system for digital computers, (ii) the *simulator*, which generates the behavior of the model, and (iii) the *executor*, which runs the simulation computations on each available processor (or core). In the next subsections each element of the architecture is detailed.

A. Model

A discrete event network model is composed of basic (atomic) models. Each model interacts through external input/output discrete events changing the states of basic models. Internal discrete events can be scheduled autonomously by basic models to change internal state. Time advance is achieved by each basic component. Hereafter are provided the definitions of both basic and network models.

Definition 1. A *basic Parallel Discrete Event System Specification (P-DEVS)* is a mathematical structure

$$\textit{P-DEVS} = (X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

Where, X is the *set of input events*, Y is the *set of output events*, S is the *set of partial states*, $\delta_{ext} : Q \times X^b \to S$ is the *external transition function* with $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ the set of total states with e the *elapsed time* since the last transition, $\delta_{int} : S \to S$ is the *internal transition function*, $\delta_{con} : S \times X^b \to S$ is the *confluent transition function, where X^b is a bag of input events,* $\lambda : S \to Y^b$ is the *output function*, *where Y^b is a bag of output events,* and $ta : S \to \mathbb{R}^{0,+}_{\infty}$ is the *time advance function.*

The modeler controls/decides the behavior in case of event collisions, when the basic system, at the same simulation time, is concerned by both internal and external events. To do so, the modeler defines the *confluent transition function* δ_{con}.

Example 2. Simple P-DEVS dynamics

In Figure 1, it is considered that at time t_2, there is no collision between external event x_0 and the internal event scheduled at time $ta(s_1) = t'_2$, with $t'_2 > t_2$, thus leading to an external transition function $\delta_{ext}(s_1, e_1, x_0) = s_2$. At time $ta(s_3) = t_4$ where there is a collision between external event x_1, occurring at time t_4, and the internal event scheduled at the same time thus leading to a confluent transition function: $\delta_{con}(s_3, x_1) = s_4$.

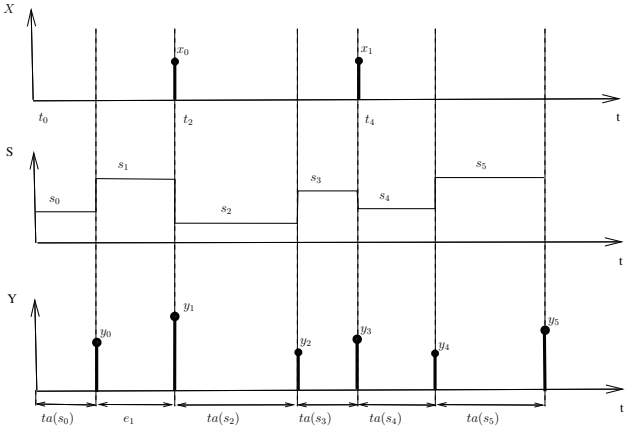

Figure 1. Simple *P-DEVS* trajectories.

Definition 3. A *P-DEVS network* is a mathematical structure

$$N = (X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\})$$

Where, X is the *set of input events*, Y is the *set of output events*, D is the set of *component names*, for each $d \in D$, M_d is a *basic or network model*, I_d is the set of *influencers* of d such that $I_d \subseteq D$, $d \notin I_d$ and, for each $i \in I_d$, $Z_{i,d}$ is the $i{-}to{-}d$ *output translation*, defined for: (i) *external input couplings*: $Z_{self,d} : X_{self} \to X_d$, with $self$ the self network name, (ii) *internal couplings*: $Z_{i,j} : Y_i \to X_j$, and (iii) *external output couplings*: $Z_{d,self} : Y_d \to Y_{self}$.

B. Simulator

Based on P-DEVS structures, different specifications can be achieved. Then, it should be possible to simulate each of these models re-using the same algorithm. This is the purpose of abstract simulators [3]. We generalize these algorithms here to parallel and/or sequential transitions under processor

supervision[1].

Algorithm 1 describes the main simulation loop of a P-DEVS model. The hierarchical structure (i.e., the composition of nested network models finally composed of basic models) is made implicit here by manipulating the set of component names (referring to all the components present in the hierarchy). This is made possible because a P-DEVS network is closed under coupling, i.e., the behavior of a P-DEVS network is equivalent to the behavior of a P-DEVS basic model resultant. In the main-loop algorithm, as in a usual discrete event simulation, simulation time advance is driven by the (last and next) times of occurrence of events. Three component sets allow focusing concisely and efficiently on active components at each time step of the simulation: (i) the *imminent set* $IMM(s)$ (the set of components that achieve both an output computation and an internal function transition), (ii) the *sender set* $SEN(s)$ (the set of components that *actually* send output events to the components they are connected with), and (iii) the *receiver set* $REC(s)$ (the set of components that receive output events). The Executor is in charge of the execution of: initialization, outputs, routing, and confluent, external and internal transitions; as well as the determination of the set of the next times of event occurrences and the imminent set.

The algorithm sequence consists of: (i) initializations of: the global time of last event t_l to 0, executor's variables (cf. Algorithm 2 for Executor procedures), components' states, the set of times of next events $TNEXT(s)$, the global time of next event t_n to the minimum time in $TNEXT(s)$; (ii) main simulation loop: imminent components (about to achieve a transition) are collected, their output is executed, outputs are routed to final basic receivers, the set of active components is computed as the union of receivers and imminents, their transition function is executed, finally global times are updated and times of next event of components are collected. The main simulation loop is executed until global time of simulation time t_{end} is reached. Notice that t_{end} can be equal to infinity meaning that all times of next event of components are infinite.

C. Executor

The executor acts as an interface between the simulation and the hardware execution. As described in Algorithm 2, the executor is called within the simulator and implements the execution of $TASKS$ (i.e., functions over components) attributing each of them to an available lightweight process $lwp \in LWP$ (here a thread). If the simulator deals with simulation time *t*, models and simulator nodes, the executor deals with *execution time* t_{exec}, lightweight processes and *logical cores* (or processors). The set $TASKS$ implements functions init, get-TN, compute-outputs, route and compute-transitions (implemented as procedures) for each component $d \in D$. Except the set of initialization functions (tasks) that depends statically on every component $d \in D$, the set of other functions evolves *dynamically* - from the *simulation* point of view. On the other hand, from the *parallelization* point of view, the number of number of lightweight processes n_{LWP} is *static*. The execution can be sequential ($n_{LWP} = 1$) or parallel ($n_{LC} \geq n_{LWP} > 1$). The algorithm described here remains deliberately abstract. Many implementation choices can be made. An implementation choice is detailed in section V.

In self-init procedure, if the *number of lightweight processes* n_{LWP} is greater than the *number of logical cores,* n_{LC}, of the machine, then $n_{LWP} = n_{LC}$[2]. In init-components procedure, the init procedure of every component is called. This procedure initializes the state and time of next event of the component. The run procedure of the executor is generic. The procedure takes a set of specified $tasks$ as argument and returns a result-Set whose content depends on the procedure that is calling (a set of times for get-TN, of imminents for get-imminents, of senders for compute-outputs, etc.) This procedure attributes each $task$ to available lightweight process $lwp \in LWP$, locks $TASKS$ set, waits a maximum time t_{exec}^{max} for each process to terminate and returns the result set. The get-TN procedure calls in parallel each component procedure getTn and returns $TNEXT(s)$ set. The get-imminents procedure calls in parallel each component procedure testTn, adds the component to the imminent set $IMM(s)$ if its time of next event $t_{n,d}$ is equal to the global time of next event t_n, and finally returns $IMM(s)$. The compute-outputs procedure calls in parallel each computeOutput procedure ($\lambda_{imminent}$) of each component $imminent \in IMM(s)$, adds the component to the sender set $SEN(s)$, and finally returns $SEN(s)$. The route procedure routes in parallel each output event ($y_{sender} \in Y_{sender}$) of each component $sender \in SEN(s)$ to the final receiver, and returns the receiver set $REC(s)$. Finally, the compute-transitions procedure calls in parallel each computeDelta procedure ($\delta_{int,active}, \delta_{ext,active}, \delta_{con,active}$) of each component $active \in ACTIVE(s)$.

III. Stochastic discrete event system specification

The formal structures reflecting the discreteness of the computations achieved by digital computers are presented here. First, a general generator definition based on sequential machines is presented. Based on this definition, a structure for pseudorandom number generators is proposed and linked to the definition of pseudorandom variables. Using a pseudorandom number generator, a pseudorandom variate generator is used for computing the realizations of pseudorandom variables. A pseudorandom and parallel event execution is specified in P-DEVS. At structural level, large numbers of connections and components in a network are captured using a pseudorandom graph definition. The latter is finally compared to the P-DEVS network structure definition.

[1]Corresponding source code has been implemented in GRADES (Graph-based and RAndom DiscrEte-event Simulator), which is accessible at https://redmine.i3s.unice.fr/projects/compsys.

[2]Otherwise the parallelization will be inefficient. Also it is expected then that the operating system assigns available cores to available lightweight processes.

A. Pseudorandom variables and number generators

Definition 4. A *generator* is an autonomous sequential machine $G = (S, s_0, \gamma)$, where S is the *set of states*, s_0 is the *initial state* and $\gamma : S \to S$ is the *state generation function*.

Algorithm 1 Main simulation loop of Root Coordinator.

Variables:
t_l: Global time of last event
t_n: Global time of next event
t_{end}: Global time of simulation end
$s = (\dots, (s_d, e_d), \dots)$: Global state
$TNEXT(s) = \{t_{n,d} \,|\, d \in D\}$: set of times of next events
$IMM(s) = \{d \in D \,|\, t_{n,d} = t_n\}$: set of imminents for next output/internal transition
$SEN(s) = \{d \in D \,|\, \lambda_d(s_d) \neq \emptyset\}$: set of senders
$REC(s) = \{d \in D \,|\, i \in I_d \wedge i \in IMM(s) \wedge x_d^b \neq \emptyset \wedge Z_{i,d}(x_d^b) \neq \emptyset\}$: set of receivers
n_{LWP}: number of lightweight processes

Begin
$t_l \leftarrow 0$
$Executor.$*self-init*(n_{LWP})
$Executor.$*init-components*(D)
$TNEXT(s) \leftarrow Executor.$*get-TN*$(D)$
$t_n \leftarrow min(TNEXT(s))$
while $t_n < t_{end}$ **do**
  $IMM(s) \leftarrow Executor.$*get-imminents*$(D, t_n)$
  $SEN(s) \leftarrow Executor.$*compute-outputs*$(IMM(s), t_n)$
  $REC(s) \leftarrow Executor.$*route*$(SEN(s))$
  $ACTIVE(s) \leftarrow IMM(s) \cup REC(s)$
  $Executor.$*compute-transitions*$(ACTIVE(s), t_n)$
  $t_l \leftarrow t_n$
  $TNEXT(s) \leftarrow Executor.$*get-TN*$(D)$
  $t_n \leftarrow min(TNEXT(s))$
end while
End

Definition 5. A *pseudorandom number generator (RNG)* (cf. [23], p.132, whose definition is extended here) is defined as $RNG = (S_P, s_{P_0}, \gamma_P)$, with $S_P = R$ the *generator state set* with $R \subset \mathbb{R}_{[0,1]}$ the *finite set of pseudorandom numbers* (with each pseudorandom number a realization of a uniformly distributed random variable, i.e., $r \sim \mathcal{U}(0,1)$), $\gamma_P : R \to R$ the *pseudorandom number generation* map, and $s_{P_0} = r_0$ the *initial status* (or *seed* for old generators). A *stream (i.e., a sequence) of independent and identically distributed (i.i.d.) pseudorandom numbers of length period l*, noted $(r_i)_{i=0}^{l-1} = r_0, r_1, \dots, r_{l-1}$, for $i = 0, 1, \dots, l-1$, is *defined by* $\gamma_P(r_i) = r_{i+1}$ and with $\gamma_P(r_{l+i}) = r_i$.

Definition 6. *A pseudorandom variate generator (RVG)* is defined as $RVG = (RNG, S_V, s_{V_0}, \gamma_V)$, with $S_V = V$ the *generator variate set*, $V \subset \mathbb{R}$ the *finite set of pseudorandom variates* (with each random variate $v \in V$ being a realization of a random variable with inverse non-uniform cumulative function distribution γ_V), $\gamma_V : R \to V$ the *pseudorandom variate generation* map, and s_{V_0} the *initial pseudorandom variate*. A *stream of pseudorandom variates follows exactly the sequence of the pseudorandom numbers generated by RNG and is of equal length l*, i.e., for $(r_i)_{i=0}^{l-1} = r_0, r_1, \dots, r_{l-1}$, there exists $(v_i)_{i=0}^{l-1} = v_0, v_1, \dots, v_{l-1}$.

Definition 7. *A pseudorandom variable* consists of the *map* $\gamma_V : R \to V$ of a *pseudorandom variate generator* $RVG = (RNG, S_V, s_{V_0}, \gamma_V)$, where $R \subset \mathbb{R}_{[0,1]}$ is a *finite set of uniformly distributed pseudorandom numbers.* Every time a *random variate* $v_i \in V$ *of* the pseudorandom variable $\gamma_V(r_i)$ is obtained, the next pseudorandom number is generated through $r_{i+1} = \gamma_P(r_i)$.

Example 8. For a pseudorandom variable following an exponential law $\gamma_{exp} \sim Exp(\lambda)$, each realization (pseudorandom variate) is obtained by $\gamma_{exp}(r) = \frac{-ln(1-r)}{\lambda} = v$ (i.e., inverting the cumulative distribution function of the exponential law).

Example 9. For a pseudorandom variable following a Bernouilli distribution $\gamma_{\mathcal{B}} \sim \mathcal{B}(p)$ of probability p, each realization (pseudorandom variate) is obtained by $\gamma_{\mathcal{B}}(r) = v = \begin{cases} 1 & if\ r \leq p \\ 0 & otherwise \end{cases}$.

B. Pseudorandom Parallel Discrete Event System Specification

As previously defined, randomness is simulated at the computer level using a pseudorandom number generator modeled as a deterministic sequential machine. Corresponding pseudorandom variables are maps taking the generated pseudorandom numbers in argument and generating corresponding pseudorandom variates. At formal P-DEVS level, the set of pseudorandom variates V can be embedded as part of the partial state.

Definition 10. A *basic Pseudorandom Parallel Discrete Event System Specification* (*PP-DEVS*) is a structure

$$PP\text{-}DEVS = (X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

Where, X and Y defined previously, $S \supseteq V$ is the *set of sets of global pseudorandom variates* $V = \Pi_{i=1}^{n} V_i$ with *n* the number of pseudorandom variables. Each set V_i contains the pseudorandom variates of a stream $(v_i)_{i=0}^{l-1} = v_0, v_1, \dots, v_{l-1}$ generated by a corresponding pseudorandom variate generator $RVG_i = (RNG_i, S_{V_i}, s_{V_{i,0}}, \gamma_{V_i})$ (cf. Definition 6), thus defining a pseudorandom variable $\gamma_{V_i} : R_i \to V_i$. At each transition function execution the next state is computed *deterministically* based on a global pseudorandom variate $v \in V$ and a partial state $s \in S$, i.e., $\delta_{int}(s, v) = s'$, $\delta_{ext}(q, x, v) = s'$, and $\delta_{con}(s, x, v) = s'$[3].

[3]The same reasoning can be used based on each pseudorandom number $r_i \in R_i$, such that the set of sets of (global) pseudorandom numbers is $R = \Pi_{i=1}^{n} R_i$, with *n* the number of pseudorandom numbers. Then, at each transition function execution the next state of *P-DEVS*$_R$ is computed based on each global pseudorandom number $r \in R$, i.e., $\delta_{int,R}(s, r) = s'$, $\delta_{ext}(q, x, r) = s'$, and $\delta_{con}(s, x, r) = s'$.

Algorithm 2 Variables, procedures and functions of Executor.

Variables:

$LWP = \{lwp \,|\, n_{LWP} \leq n_{LC}\}$: set of lightweight processes

n_{LWP}: number of lightweight processes

n_{LC}: number of logical cores

$TASKS = \{f_d \,|\, d \in D\}$: set of tasks with f_d a function to execute over $d \in D$

t_{exec}^{max}: maximum execution time of a process

Begin

procedure SELF-INIT(n_{LWP})
$n_{LC} \leftarrow getNbOfLogicalCores()$
if $n_{LWP} > n_{LC}$ **then**
$n_{LWP} \leftarrow n_{LC}$
end if
end procedure
procedure INIT-COMPONENTS(D)
set $TASKS = \{(d, init(0)) \,|\, d \in D\}$
$run(TASKS)$
end procedure
function RUN($TASKS$)
In parallel $\forall task \in TASKS$ **do**
run $task$ on available $lwp \in LWP$
add possibly result **to** ResSet
end In parallel
lock $TASKS$
wait t_{exec}^{max} for each $lwp \in LWP$ to terminate
return ResSet
end function
function GET-TN(D)
$TASKS = \{(d, getTn()) \,|\, d \in D\}$
$TNEXT(s) \leftarrow run(TASKS)$
return $TNEXT(s)$
end function
function GET-IMMINENTS(D, t_n)
set $TASKS = \{(d, testTn()) \,|\, d \in D\}$
$IMM(s) \leftarrow run(TASKS)$
return $IMM(s)$
end function
function COMPUTE-OUTPUTS($IMM(s), t_n$)
set $TASKS = \{(imminent, computeOutput(t_n)) \,|\, imminent \in IMM(s)\}$
$SEN(s) \leftarrow run(TASKS)$
return $SEN(s)$
end function
function ROUTE($SEN(s)$)
$TASKS = \{(sender, route()) \,|\, sender \in SEN(s)\}$
$REC(s) \leftarrow run(TASKS)$
return $REC(s)$
end function
procedure COMPUTE-TRANSITIONS($ACTIVE(s), t_n$)
$TASKS = \{(active, computeDelta(t_n)) \,|\, active \in ACTIVE(s)\}$
$run(TASKS)$
end procedure

End

The use of pseudorandom numbers in deterministic sequential DEVS models has been discussed in the context of probability spaces [6]. This work pinpointed cases where the previous definition may show inconsistencies as well as convergence issues (when elements are not measurable or corresponding sets infinite). However, our goal here is not to redefine a new formalism at continuous system specification level but rather to specify the deterministic foundations of the stochastic simulations achieved at computer level and how this can be modeled in P-DEVS as a first step. This does not prevent achieving further mathematical extensions, as done in [6].

C. Pseudorandom graph-based network

A pseudorandom directed graph generator generates a set of simple directed graphs with the same coupling probability and the same number of vertices.

Definition 11. A *Pseudorandom Generator of Directed Graphs (RGG)* is a structure $RGG = (\mathcal{G}^{n,p}, S_G, s_{G_0}, \gamma_G)$, where $\mathcal{G}^{n,p}$ is the *set of all pseudorandomly generated directed graphs* such that $\mathcal{G}^{n,p} = \mathcal{G}\{n, P(arrow) = p\}$, with n the *number of vertices* and $p \in \mathbb{R}_{[0,1]}$ the *probability of choosing an arrow*. Each graph $G(U, A) \in \mathcal{G}^{n,p}$ is described by $U = \{1, 2, \ldots, n\}$ the *set of vertices* and A a set of *(ordered pairs) arrows*; $S_G = A \times V_{coupling} = U^2 \times \mathbb{B}$ with $V_{coupling}$ the *set of coupling pseudorandom variates* obtained by sampling corresponding (Bernouilli) *coupling pseudorandom variable* $\gamma_{coupling} \sim \mathcal{B}(p)$[4]; $s_{G_0} = v_{coupling,0}$ is the *initial coupling pseudorandom variate*; Last map $\gamma_G : \mathcal{G}^{n,p} \times S_G \rightarrow \mathcal{G}^{n,p}$ is the *directed graph generation map* using the coupling pseudorandom variates $V_{coupling}$ to construct a graph $G(U, A) \in \mathcal{G}^{n,p}$.

Example 12. Simple graph generation

A graph $G \in \mathcal{G}^{n,p}$ can be iteratively constructed by a pseudorandom generator of directed graphs $RGG = (\mathcal{G}^{n,p}, S_G, s_{G_0}, \gamma_G)$, with S_G, s_{G_0}, $\mathcal{G}^{n,p}$ as defined previously and $\gamma_G(G_i, v_i) = G_{i+1}$, with $G = \cup_i G_i$, $G_0(U, A = \emptyset)$ (here the initial graph has all vertices but no edges), for $i = 0, 1, \ldots, n^2 - 1$. The length period n^2 is due to the algorithmic testing of edges, i.e., for each vertex, each arrow to each other vertex is tested.

Example 13. A directed graph $G_1(U_1, A_1) \in \mathcal{G}^{n_1,p_1}$ can be connected to another directed graph $G_2(U_2, A_2) \in \mathcal{G}^{n_2,p_2}$ with probability p, leading to a pseudorandom directed graph $G(U, A) \in \mathcal{G}^{n,p}$ with $U = \{U_1, U_2\}$, $A = \{A_1, A_2, A_3\}$ and p the probability of choosing an arrow in A_3 from vertices in U_1 to vertices in U_2. Algorithmically, graphs G_1 and G_2 are generated and finally G is generated coupling G_1 and G_2 with probability p.

Once the graph structure has been generated, the graph can be transformed into a network where to each node corresponds

[4]Pseudorandom variates in S_G are generated by a pseudorandom variate generator $RVG_{coupling} = (RNG_{coupling}, V_{coupling}, v_{coupling,0}, \gamma_{coupling})$.

a P-DEVS component and to each arrow a coupling.

Definition 14. A *Graph-to-P-DEVS Network Transformer* (GNT) is a structure $GNT = (G, N, \{m_{i,j}\})$, where G is a *directed graph*, N is a *P-DEVS network*, $m_{i,j}$ is a *one-to-one map* (from the elements of G to the elements of N) defined for: (i) *vertices-to-components* $m_{u,c} : U \to D$, (ii) *arrows-to-couplings* $m_{a,c} : U \times U \to D \times D$ with $D \times D = \{(a, Z_{a,b}(a)) \,|\, a \in I_b\}$ the influencer-to-influencee pairs, and (iii) *arrows-to-influencers* $m_{a,i} : U \times U \to \{I_i\}$ with $m_{a,i}(u, u') \in I_{u'}$ the selection of the influencer of u'.

IV. Spiking neural network model

Mathematical modeling of a random spiking neural network is presented here. The model is specified after using the main mathematical structures presented in previous sections.

A. Biological neuron

Figure 2 depicts a single biological neuron. Most commonly, inputs from other neurons are received on *dendrites*, at the level of *synapses*. The circulation of neuronal activity (electric potentials) is due to the exchange through the neuron *membrane* of different kinds of ions. Dendrites integrate locally the variations of electric potentials, either excitatory or inhibitory, and transmit them to the *cell body*. There, the genetic material is located into the *nucleus*. A new pulse of activity (an *action potential*) is generated if the local electric potential reaches a certain threshold at the level of the *axon* hillock, the small zone between the cell body and the very beginning of the axon. If emitted, action potentials continue their way through the axon in order to be transmitted to other neurons. Action potentials, once emitted, are "all or nothing" phenomena: $0, 1$. The propagation speed of action potentials can be increased by the presence of a *myelin* sheath, produced by *Schwann cells*. This insulating sheath is not continuous along the axon. There is no myelin at the level of the *nodes of Ranvier*, where ionic exchanges can still occur. When action potentials reach the tip of the axon, they spread over all *terminals* with the same amplitude, up to synapses. The neuron can then communicate with other following neurons. Notice that a focus on electrical signals (without dealing with chemical signals) is achieved here.

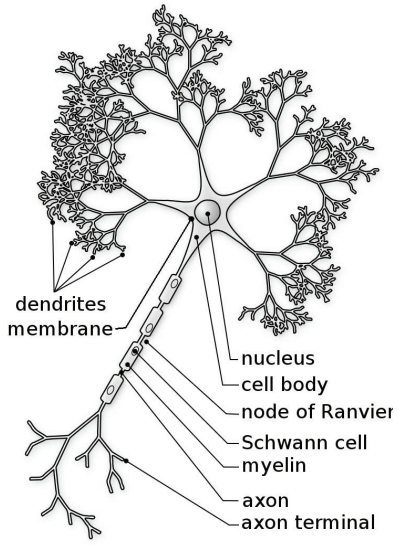

Figure 2. Sketch of a neuron (adapted from http://fr.wikipedia.org/wiki/Neurone).

B. Model

At the model level, the network structure and the behavior of the neurons are described here. While definitions provided here are general, they are mapped in next subsection to the mathematical structures provided in subsections III-B and III-C.

The structure presented here consists of: an input layer of independent firing neurons, an intermediate layer embedding a pseudorandomly generated directed graph of neurons, an output layer of independent receiving neurons.

Definition 15. The structure (cf. Figure 3)

Let I, B, O be 3 finite sets with respective cardinality n, M and N. It is always assumed that $M \geq N$. Let $(p_i)_{i \geq 0}$ denote real numbers in $[0, 1]$. For any $(i, j) \in B^2$, assume that there exists an arrow $i \to j$ with probability p_0, for any $i \in I$ and $j \in B$, assume that there exists an arrow $i \to j$ with probability p_1 and for any $i \in B$ and $j \in O$, assume that there exists an arrow $i \to j$ with probability p_2.

The dynamics in each layer is provided in next definition. In input layer, neurons fire randomly while neurons in both intermediate and output layers follow a deterministic behavior.

Definition 16. The dynamics

Assume that the activities $(X_t(i))_{i \in I, t \in \mathbb{N}}$ of the sites in I and time t are i.i.d. $\mathcal{B}(p_3)$. Let a be a positive real number. For all $(i, j, t) \in (B \cup O)^2 \times \mathbb{N}$, we choose i.i.d. thresholds $\tau_i \sim \mathcal{N}(m, \mathcal{S}^2)$, i.i.d. $w_{i,j} = 1$ with probability $1 - p_4$ and $-a$ with probability p_4. Then, the *membrane potential* $P_i(t)$ of a neuron i, initially null is updated thanks to the following rule

$$P_i(t) = (rA_i(t-1) + \sum_{i \sim j} w_{i,j} A_j(t-1))(1 - A_i(t-1))$$

Where, $r \in (0, 1)$ is the *activity remaining from time* $t-1$, $\sum_{i \sim j} w_{i,j} A_j(t-1)$ is the *activity received from other neurons* at time $t-1$, $(1 - A_i(t-1))$ reflects a *refractory period* of 1 (if the neuron fired at time $t-1$ it cannot fire at time t), and the *activity of a neuron* i is provided by

$$A_i(t) = \begin{cases} 1 & if\ P_i(t-1) \geq \tau_i \\ 0 & otherwise \end{cases}$$

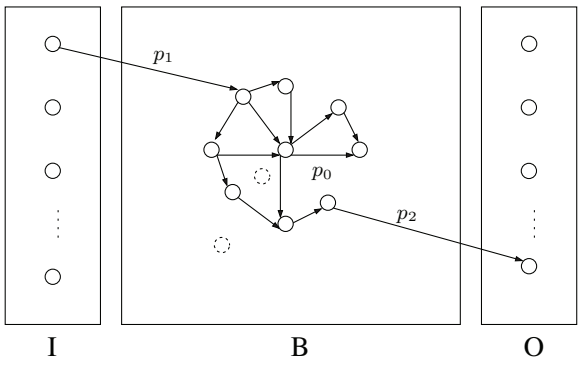

Figure 3. Structure of the neuron model.

C. Specification in PP-DEVS

The structure of Definition 15 can be set by a pseudorandom generator of directed graphs (RGG) (cf. Definition 11). The coupling of the different layers is based on Example 13 with each layer being a directed graph (with both input and output layers having no internal connections). The resulting coupled graph is finally mapped to a P-DEVS network through a *Graph-to-P-DEVS Network Transformer* (GNT) (cf. Definition 14).

From a dynamical point of view, each neuron $i \in I$ of Definition 16 is specified as a PP-DEVS reduced to internal transitions as

$$M_i = (Y_i, S_i, \delta_{int,i}, \lambda_i, ta_i)$$

Where, $Y_i = \{\emptyset, 1\}$, with null event $\emptyset$ (resp. 1) if the neuron is non-firing (resp. firing), $S_i = V_{firing} = \mathbb{B}$ with V_{firing} the set of firing pseudorandom variates, *internal transition function* $\delta_{int,i}(s, v_{firing})$ samples the pseudorandom variable $\gamma_{firing} \sim \mathcal{B}(p_3)$ indicating the activity of the neuron depending on probability p_3, output function $\lambda_i(v_{firing})$ sends an unitary event if the neuron is active and *time advance function* $ta_i(s) = 1$ ensures the discrete time sampling of γ_{firing}.

Neurons in B and O of Definition 16 are P-DEVS models specified as

$$M_j = (X_j, Y_j, S_j, \delta_{ext,j}, \delta_{int,j}, \delta_{con,j}, \lambda_j, ta_j)$$

Where, $X_j = \{\emptyset, 1\}^n = \{\emptyset, 1\} \times \ldots \times \{\emptyset, 1\}$ (with n the *number of inputs*), $Y_j = \{\emptyset, 1\}$, $S_j = \{\{w_k\}, c, a, p, phase = \{\mathsf{firing}, \mathsf{active}, \mathsf{inactive}\}\}$ with w_j the *weight of corresponding input* k, c (resp. c') the *sum of received inputs* at a time step t (resp. at a time step $t+1$), a (resp. a') the *activity of the neuron* at a time step t (resp. at time step $t+1$), p (resp. p') the *membrane potential* of the neuron at a time step t (resp. at time step $t+1$), *external transition function* $\delta_{ext}(q, x)$ collects the inputs received at time t, computes the next phase and the next membrane potential p' and activity a', and after call for a next internal transition at time $t+1$, *internal transition function* $\delta_{int}(s)$ updates $p \leftarrow p'$ and $a \leftarrow a'$ and reset inputs ($c \leftarrow 0$), if the neuron is in phase active or firing and receives an input the confluent transition function is called as $\delta_{con}(s, x) = \delta_{ext}(\delta_{int}(s), 0, x)$, i.e., first update variables and after collect inputs, and finally *time advance function* $ta_j(s) = 1$ if the neuron is in phase active or firing and $ta_j(s) = \infty$ if the neuron is in phase inactive.

V. Simulation process and results

Model generation and simulation process are introduced first here. After, the speedup results are presented and discussed. The goal of this speedup analysis is only an application proof of the whole hierarchy developed here. Gaining genericity has usually a cost. It is not our purpose here to prove the supremacy of this approach at the parallel implementation level but rather to prove completeness and applicability.

A. Environment infrastructure and graphical outputs

The steps and the elements of the process of generation and simulation of the model consist of the following sequence: (i) Initialization of all models, (ii) Graph generation using a model RGG, (iii) Graph-to-network transformation (GNT), which generates a PP-DEVS network from the graph, and (iv) Simulation.

Notice that as defined previously, each object uses one RNG for each pseudorandom variable. This ensures: (i) the statistical independence between pseudorandom variables, and (ii) the reproducibility of pseudorandom simulations [8].

Figure 4 depicts a snapshot of the graph corresponding to neurons of set *B*. Notice how dense is the graph connection making it difficult to differentiate edges.

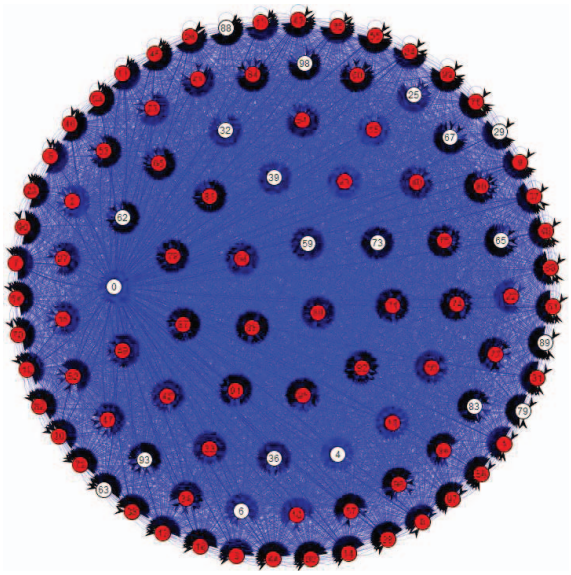

Figure 4. Graph snapshot of *B* set.

Simulations have been performed on a Symmetric Multiprocessing (SMP) machine with *80* physical cores and *160* logical cores, 8 processors Intel(R) Xeon(R) CPU *E7-8870@2.40GHz*[5], and *1Tb* RAM. Figure 5 presents the firing of neurons for neurons of each set *I*, *O*, and *B*.

[5]stepping: 2, cpu: 1064 MHz, cache size: 30720 KB.

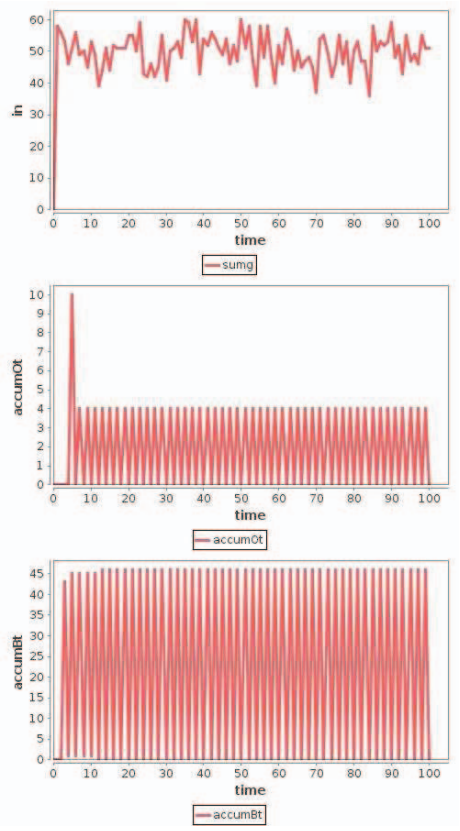

Figure 5. Firing outputs in sets *I*, *O*, and *B*.

B. Speed-up results

In [25], an interesting perspective is drawn concerning the usage of clusters with low latency communication capabilities. Our idea here is to assume (even at abstract simulator level) that all the computations are centralized on a single computer, a Symmetric Multiprocessing (SMP) machine[6]. The latter allows sharing memory and minimizing the latency of communications. Besides, centralizing all the computations facilitates the control of their executions and their synchronization at each time step. Different sizes of neural networks are simulated here for different numbers of threads.

Input parameters are set to values: $p_0 = p_1 = p_2 = 0.9$, $p_3 = 0.5$, $p_4 = 0.2$, $a = r = 1$, each threshold $\tau_i \sim \mathcal{N}(m, \mathcal{S}^2)$, with $m = 250$ and $\mathcal{S} = 1$. The whole simulation has been implemented in Java programming language.

The *sequential execution time of methods* $t_{methods}$ has been considered as the sum of the execution times for methods: *initialization* (t_{init}), *output* (t_{out}), *routing* (t_{rout}), and *transitions* (t_{trans}) (cf. Algorithm 1), for different sizes of networks. Considering t_{total} as the *total parallelizable execution time*, and t_{seq} as the *sequential execution time that cannot be parallelized*, it has been noticed that most of the execution times of a simulation is due to the execution of these methods, i.e., $\frac{t_{total}}{t_{methods}} = 93.2\%$ for 140 neurons, increasing quickly to 99.3% for 240 neurons. This shows the high parallelizability of P-DEVS simulations. Besides, it has also been noticed that most of the execution time is due to the execution of atomic output and transition functions, i.e., $\frac{t_{total}}{t_{trans}+t_{out}} = 91.51\%$ for 140 neurons increasing quickly to 99.08% for 240 neurons.

Figure 6 presents the speedup obtained for different sizes of networks according to different numbers of threads (implemented in a pool[7]). Each replication has been replicated 30 times leading to a total number of $19 \times 30 \times 4 = 2280$ simulations. It can be seen that in each simulation, the speedup reaches a maximum which remains constant (cf. Figure 6.c and Figure 6 .d) or decreases (cf. Figure 6.a and Figure 6.b).

Each best average speedup obtained in Figure 6 is presented in Figure 7. The optimal number of pool threads is: 20 for 140 neurons, 60 for 240 neurons, 100 for 340 neurons and 50 for 440 neurons. Increasing the number of neurons the average best speedup decreases and a practical maximum speedup of 23.5 is achieved.

Finally, to investigate the parallelizability of our simulation model, let's consider Amdahl's law [2] as $S(n) = \frac{1}{\tau_{seq}+\frac{1}{n}(1-\tau_{seq})}$ with the *maximum theoretical speed up* $S(n)$ (considering no parallelization overhead) for a *number of threads* n, and the f*raction of total execution time as strictly sequential* as $\tau_{seq} = \frac{t_{seq}}{t_{total}}$. Having $n = 80$ physical cores on the SMP machine used, for 140 neurons, the theoretical maximum speedup is $S(80) = 14.3$ (while the practical speedup is 5.14) and for 240 neurons, the theoretical maximum speedup is $S(80) = 53$ (while the practical maximum speedup is 22.2). Practical maximum speedup is less than half of theoretical maximum speedup, suggesting great further potential speedup.

[6]Simulations have been performed on a Symmetric Multiprocessing (SMP) machine with *80* physical cores and *160* logical cores, 8 processors Intel(R) Xeon(R) CPU *E7-8870@2.40GHz* (stepping: *2*, cpu: *1064MHz*, cache size: *30720KB*.), and *1Tb* RAM. Each Java class main has been executed in command line using the exec-maven-plugin-1.2.1. Execution times correspond to the total (processor) time information provided by Maven. Finally, although when running the simulations the machine was possibly executing other simulations (launched by other users), the number of available threads has been verified at each simulation time and 30 replications of each simulation have been achieved showing a good confidence interval of the results obtained.

[7]Notice that for each simulation the Java Virtual Machine added also 16 threads for garbage collection and specific to the libraries used in the simulator.

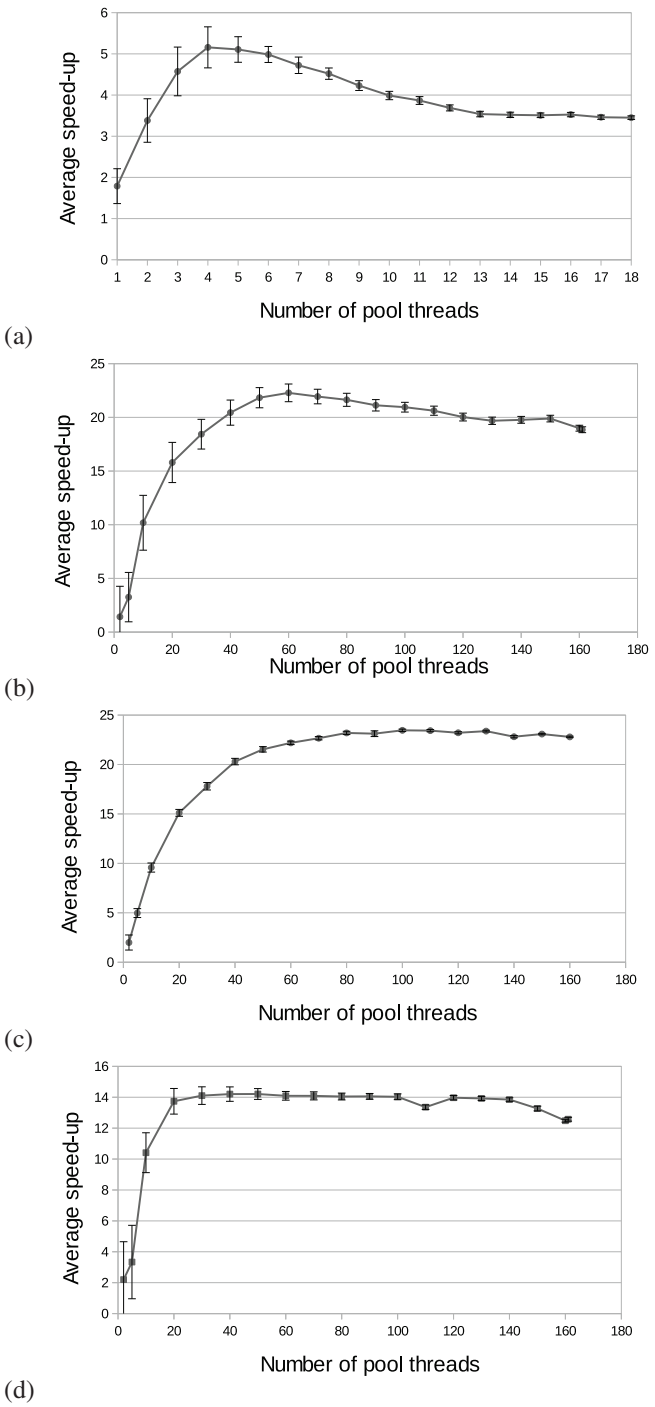

Figure 6. Comparison of execution time results for an increasing number of pool threads and: (a) 140 neurons, (b) 240 neurons, (c) 340 neurons, and (d) 440 neurons.

The cap speedup obtained (while increasing the number of threads) can be explained by JVM intrinsic limitations. In [12], different very basic experiments have been implemented in parallel for different numbers of threads (quicksort, calculation of π value by Monte Carlo method, Fast Fourier transform, discrete cosine transform, etc.). The platform consisted in an Intel Xeon Phi Coprocessor *5100* accelerator with a memory size of *8GB DDR5*, an L2 cache size of *30MB*, 60 physical cores and 240 hardware threads, and a base processor frequency of *1.1GHz*. For quicksort and Monte Carlo experiments, the speedup obtained shows the same cap with no improvement respectively above 30 and 60 threads. For other experiments, the results are even worse showing a decrease of the speedup above 60 threads even though the codes were compute intensive. Furthermore, JVM opacity makes difficult further investigations of both load balancing and memory access (to test a possible memory bandwidth issue). For example, when writing these lines we were not able to find any good quality profiling software for analyzing Java parallel simulation results. This is why, although we believe that better speedup results can be obtained, we recommend further investigations to use another programming language (e.g., *C++*).

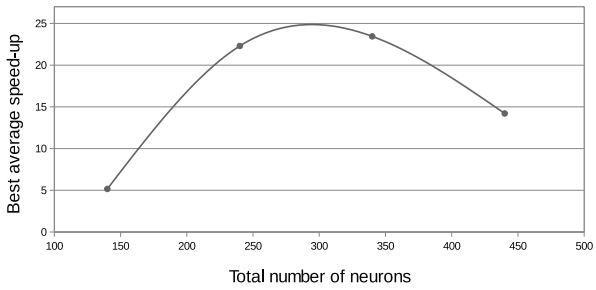

Figure 7. Best average execution-time speedup for each total number of neurons.

VI. Conclusion and perspectives

This article presented a first formal bridge between computational discrete event systems and networks of spiking neurons. Parallel and stochastic aspects (and their relationship) have been defined explicitly. In P-DEVS a simple way of parallelizing simulations and a link between P-DEVS and (pseudo)random graphs/generators/variables have been proposed. Finally all these structures have been applied to a network of spiking neurons. From a simulation point of view, it can be seen that most of the sequential execution times (more than 90%) can be reduced theoretically. In practice, the simplicity obtained by centralizing most of the computations at the same place requires a strong optimization at software level and a suitable solution at hardware level.

In conclusion, although further technical investigations need to be achieved, it is believed that: the formal structures provided here allow mathematical reasoning at (computational) system level and that the simplicity of the parallel (reproducible[8]) implementation technique should allow further

[8]Parallelizing the discrete event execution of a scheduler, at each time step, and encapsulating each stream of random numbers in corresponding pseudorandom variable (in their turn encapsulated in atomic models) simply preserves simulation reproducibility [8]. Furthermore, the technique has also the advantage to do not require any control over the order of execution of threads (that is not guaranteed by some programming languages, e.g. Java) to preserve simulation reproducibility.

(more efficient) parallelization developments, based on our theoretical maximum speedup results.

ACKNOWLEDGEMENTS

Many thanks to Gaëtan Eyheramono and especially to Antoine Dufaure who achieved a first version of the multithreaded implementation. This work has been partially funded by a contract Projets Exploratoires Pluridisciplinaires Bio-Maths-Info (PEPS-BMI 2012), entitled Neuroconf, and funded by Centre National de la Recherche Scientifique (CNRS), Institut national de recherche en informatique et en automatique (INRIA) and Institut National de la Santé et de la Recherche Médicale (INSERM).

REFERENCES

[1] ADEGOKE, A., TOGO, H., AND TRAORÉ, M. K. A unifying framework for specifying DEVS parallel and distributed simulation architectures. *Simulation 89*, 11 (2013), 1293–1309.

[2] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485.

[3] B. P. ZEIGLER, T. G. KIM, H. P. *Theory of Modeling and Simulation*. Academic Press, 2000.

[4] BEHRENDS, R., DILLON, L. K., FLEMING, S. D., AND STIREWALT, R. E. K. 10^{14}. Tech. Rep. RJ10502 (ALM1211-004), IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, USA, November 13 2012.

[5] BRETTE, R. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience 23*, 3 (2007), 349–398.

[6] CASTRO, R., KOFMAN, E., AND WAINER, G. A formal framework for stochastic discrete event system specification modeling and simulation. *Simulation 86*, 10 (2010), 587–611.

[7] CHOW, A. C. H., AND ZEIGLER, B. P. Parallel devs: A parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Conference on Winter Simulation* (San Diego, CA, USA, 1994), WSC '94, Society for Computer Simulation International, pp. 716–722.

[8] HILL, D. Parallel random numbers, simulation, and reproducible research. *Computing in Science Engineering 17*, 4 (July 2015), 66–71.

[9] HILL, D. R. C., MAZEL, C., PASSERAT-PALMBACH, J., AND TRAORE, M. K. Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience 25*, 10 (2013), 1427–1442.

[10] HINES, M., AND CARNEVALE, N. Discrete event simulation in the NEURON environment. *Neurocomputing 58-60*, 0 (2004), 1117–1122.

[11] JEFFERSON, D., BECKMAN, B., WIELAND, F., BLUME, L., AND DILORETO, M. Time warp operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1987), SOSP '87, ACM, pp. 77–93.

[12] MALINOWSKI, A. Modern platform for parallel algorithms testing: Java on intel xeon phi. *International Journal of Information Technology and Computer Science(IJITCS)* (2015), 8–14.

[13] MAYRHOFER, R., AFFENZELLER, M., PRÄHOFER, H., HÖFER, G., FRIED, A., AND FRIED, E. Devs simulation of spiking neural networks. In *Cybernetics and Systems: Proceedings EMCSR 2002* (2002), vol. 2, pp. 573–578.

[14] MEROLLA, P., ARTHUR, J., AKOPYAN, F., IMAM, N., MANOHAR, R., AND MODHA, D. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE* (Sept 2011), pp. 1–4.

[15] MOURAUD, A., PUZENAT, D., AND PAUGAM-MOISY, H. DAMNED: A Distributed and Multithreaded Neural Event-Driven simulation framework. *Computing Research Repository abs/cs/051* (2005).

[16] RM., F. *Parallel and distributed simulation systems*. Wiley, New York, 2000.

[17] TANG, Y., ZHANG, B., WU, J., HU, T., ZHOU, J., AND LIU, F. Parallel architecture and optimization for discrete-event simulation of spike neural networks. *Science China Technological Sciences 56*, 2 (2013), 509–517.

[18] TONNELIER, A., BELMABROUK, H., AND MARTINEZ, D. Event-driven simulations of nonlinear integrate-and-fire neurons. *Neural Computation 19*, 12 (2007), 3226–3238.

[19] VAHIE, S. *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*. Springer-Verlag, 2001, ch. Dynamic Neuronal Ensembles: Neurobiologically Inspired Discrete Event Neural Networks.

[20] WANG, Y.-H., AND ZEIGLER, B. Extending the devs formalism for massively parallel simulation. *Discrete Event Dynamic Systems 3*, 2-3 (1993), 193–218.

[21] ZEIGLER, B. Discrete event abstraction: an emerging paradigm for modeling complex adaptative system.

[22] ZEIGLER, B. P. Statistical simplification of neural nets. *International Journal of Man-Machine Studies 7*, 3 (1975), 371–393.

[23] ZEIGLER, B. P. *Theory of Modeling and Simulation*. Wiley, 1976.

[24] ZEIGLER, B. P., NUTARO, J. J., AND SEO, C. What's the best possible speedup achievable in distributed simulation: Amdahl's law reconstructed. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015* (2015), pp. 189–196.

[25] ZENKE, F., AND GERSTNER, W. Limits to high-speed simulations of spiking neural networks using general-purpose computers. *Frontiers in Neuroinformatics 8*, 76 (2014).