Concepts for Software-Hardware Run-Time Interoperability of Spiking Neural Network Models

Internship Report

Dimitri Probst
eMail: Dimitri.Probst@stud.uni-heidelberg.de

supervised by Dr. Daniel Brüderle

April 18, 2010

Contents

1 Introduction

Due to insufficient experimental methods to quantify information processing dynamics in biological neural networks, neuro-scientific modelling is an important subject area in nowadays' biophysics. A growing number of computation tools appeared in order to flexibly study neural network models, to minimize experimential cost and time, and, in addition to that, to produce reusability, though offering the user to obtain precise simulations as well as publishable figures.

Chapter 2 introduces the main facts about neuro-scientific modelling and draws a comparison between the two unlike techniques to imitate neural networks: In section 2.2, software simulation stratergies are described and two of the most known simulation environments are presented. Section 2.3 depicts facts about the FACETS[1] neuromorphic hardware developed by the Electronic Vision(s) Group of the Kirchhoff-Institut for Physics at the University of Heidelberg.

Chapter 3 starts with an overview of methods to facilitate interoperability of different simulators. One goal of these approaches is a standardisation of today's available range of simulation environments and neuromorphic hardware emulations in order to reduce their physical handicaps. Therefore, as a project result, a novel interaction concept for a software-hardware co-simulation is presented.

[1]Fast Analog Computing with Emergent Transient States

2 Neuro-Scientific Modelling

In this chapter, firstly, nowadays' utilized spiking neural models and their mathematical basics are described. Afterwards, the two disparate methods to replicate and quantify neural networks are presented: software simulators and hardware emulators. For each of them, advantages and disadvantages are revealed. To conclude the chapter, a comparison is drawn between these two techniques.

2.1 Spiking Neural Models

By including the concept of temporal development of the membrane potential, in addition to the neuronal and the synaptic states, spiking neural network models become an approximately realistic representation of neurobiological findings [2].

This section introduces the mostly used spiking neural models and their underlying differential equations.

The following information is mainly looked up from [10] and [3].

2.1.1 Leaky Integrate-and-Fire Model (LIF)

The leaky integrate-and-fire model (LIF) is a simple, i.e. analytically solvable, neural model. It is made up of a parallel circuit consisting of a capacitor C and a resistor R. The membrane potential $V(t)$ is governed by the following differential equation:

$$\tau_m \frac{dV}{dt} = -V(t) + R \cdot I(t) \ . \tag{1}$$

Here, $\tau_m = RC$ is the *membrane time constant* of the neuron and $I(t)$ the *driving current* of this basic circuit. The form of an action potential is not described explicitly [10]. Spikes are events which are represented by a *firing time* t_{fire}. If the membrane potential $V(t)$ reaches a critical value $V(t) > V_{th} = V(t_{fire})$, a spike is emitted and, immediately, the membrane potential is reset to the *membrane reset potential* V_{reset}. V_{th} is called the *threshold voltage.*

The chip model, presented in section 2.3.1, is determined by the LIF model. The emulated membrane potential is regulated by the following differential equation:

$$-C_m \frac{dV}{dt} = g_l(V - E_l) + \sum_j p_j(t) g_j(t)(V - E_x) + \sum_k p_k(t) g_k(t)(V - E_i) \tag{2}$$

The constant C_m represents the *total membrane capacitance.* g_l is the *leakage conductance* and depends on the time constant $\tau_m = \frac{C_m}{g_l}$ of the exponential convergence of the membrane potential. The first term on the right-hand side models the development of the membrane potential, if no transient conductances towards other reversal potentials are active. Thus, E_l is called *leakage reversal potential.* The transient conductances imposed by synaptic activity have different reversal potentials, E_x for excitatory synapses and E_i for inhibitory ones. The index j in the first sum runs over all excitatory synapses, while the index k in the second sum conceals the inhibitory synapses. An individual synapse s generates a conductance course (CC), which is determined by the product $p_s(t) \cdot g_s(t)$, where $p_s(t)$ stands for the *synaptic open probability* and $g_s(t)$ is the product of the *synaptic weight* $\omega_s(t)$ and the *maximum conductance* $g_s^{max}(t)$:

$$g_{j,k}(t) = \omega_{j,k}(t) \cdot g_{j,k}^{max}(t) \tag{3}$$

2.1.2 Adaptive Exponential Integrate-and-Fire Model (aEIF)

The temporal development of the membrane potential $V(t)$ in the so called *adaptive exponential integrate-and-fire model* (with conductance-based synapses) is governed by the following differential equation:

$$\begin{aligned} -C_m \frac{dV}{dt} = g_l(V - E_l) - \mathrm{g}_l \Delta_{th} \mathrm{exp}\, [\frac{V - V_{th}}{\Delta_{th}}] + \mathrm{g}_x(t)(V - E_x) \\ + \; \mathrm{g}_i(t)(V - E_i) + \omega(t) \end{aligned} \tag{4}$$

The variables C_m, g_l, E_l, E_x and E_i are defined as in 2.1.1. $g_x(t)$ and $g_i(t)$ represent the total excitatory and inhibitory conductances.

Compared to equation 2, a new mechanism is introduced to the I&F neuron by the exponential term on the right hand side. The *threshold potential* V_{th} stands for the critical value, above which the membrane potential fast develops towards infinity. The rapidity is set by the *slope factor* Δ_{th}. If the membrane potential $V(t)$ reaches a critical value $V_{spike} > V_{th}$, a spike is emitted and the membrane potential is reset to V_{reset} by a very strong conductance.

Another modification of the basic COBA I&F model is the *adaptation current* $\omega(t)$, which is determined by

$$-\tau_\omega \frac{d\omega}{dt} = \omega(t) - a(V - E_l) \ . \tag{5}$$

Every time a spike is emitted, ω changes its value instantaneously. τ_ω represents the *time constant* and a the *efficacy* of the so called *sub-threshold* adaptation mechanism [1][3].

2.1.3 Hodgkin & Huxley Model (HH)

The HH model is a detailed neural simulation model, which is not based on integrate-and-fire mechanisms, but on the actual ionic current flows in the cell membrane. These flows are controled by an, in this case simplified, differantial equation:

$$-C_m \frac{dV}{dt} = -I(t) + \sum_k I_k(t) \tag{6}$$

where V_m is the *membrane potential.* The constant C_m represents the *total membrane capacitance.* On the right hand side, $I(t)$ is the whole applied current, consisting of the *individual ionic currents*, and so called *leakage currents.* $I_k(t)$ denote the *individual ionic currents*, which have their own reversal potentials. These currents provide for a realistic empirically-based model.

Here, the neuron fires, if either the condition $\frac{dV_m}{dt} \geq \Theta$ and/or $V_m \geq \Theta$ is obtained, whereas Θ is a configurable constant.

For detailed information the reader is referred to [2] or [9].

2.1.4 Spike Response Model (SRM)

The *Spike Response Model* (SRM) is a generalization of the leaky I&F model. Characteristically for SRMs, the participating parameters en masse depend on the time that has past since the last output spike. Another difference to intergrate-and-fire models is the fact, that, in order to calculate the membrane potential at time t, the SRM has to solve integrals over the past time [10].

2.2 Software Simulators

This section firstly summarizes facts about the actual simulation process and deals with different simulation strategies. Afterwards, two of the most famous freely-available simulation-environments are presented. In conclusion, the main advantages and disadvantages of using software simulators are outlined.

The following information is mainly extracted from [2].

2.2.1 Simulation Process

In computer simulations, neurons are usually represented by a hybrid system formalism. Thus, the neural membrane potential dynamics and the synapse activity is handled by solving differential equations of the form:

$$\frac{d\overrightarrow{X}}{dt} = f(\overrightarrow{X}) \tag{7}$$

There are several numerical solution methods for differential equations, such as the *midpoint method*, the *Forward-Euler*, the *Backward-Euler*, the

*Exponential-Euler*and the *Runge-Kutta* solution method. Each of the software simulators differ in their kind of applied solution methods, considering the required time resolution, on the one hand, and the precision of the researched issue, on the other hand.

In equation 7, the variable $\overrightarrow{X}$ generally stands for the state of the neuron, which is varied upon incoming spikes from connected synapses. While synapses are only updated at every incoming spike, the state of the neuron has to be updated at each time step. Additionally, it has to be reset to a determined value, if some particular threshold conditions are satisfied, e.g. the membrane potential $V_m \geq \Theta$ in I&F models. This reset can be integrated into the hybrid system formalism by considering that outgoing spikes act on $\overrightarrow{X}$ through an additional (virtual) synapse. Thus, if no transmission delays are included into the model, spike times need not be stored, whereby unnecessary computational traffics are avoided.

For fast synaptic simulations often linearities are used, where all synaptic variables sharing the same linear dynamics can be reduced to a single one. This reduction also applies to synapses with higher-dimensional dynamics, as long as it is linear and the spike-triggered changes do not depend on the state of the synapse.

2.2.2 Simulation Strategies

In this context, different types of simulation strategies and algorithms that are currently implemented are described. To these belong *synchronous/ clock-driven algorithms* and *asynchronous/event-driven algorithms.*

Synchronous or Clock-Driven Algorithms In synchronous or clock-driven algorithms, the state variables of all neurons and/or synapses are updated simultaneously at every tick of a clock $X(t) \rightarrow X(t+dt)$ by solving linear or non-linear differential equations. After updating all variables, the threshold condition is checked for every neuron, and, if the neuron satisfies this model-dependent condition, a spike is emitted and the membrane potential is reset. Figure 1 in [2] shows a basic clock-driven algorithm.

The computational cost C_{tot} per 1s of biological time for these processes consists of the state updating and the propagation of spikes:

$$C_{tot}(1s) = (c_U \times \frac{N}{dt} + c_P \times F \times N \times p) \times 1s \tag{8}$$

Here c_U and c_P are the costs for one update or one spike propagation respectively. N stands for the number of neurons, dt for the duration of the time bin, F represents the average target rate and p the target neurons, both refering to one neuron. Considering *transmission delays*, which are representing real axonal delays, as far as they store future synaptic evens in a circular array, additional costs

$$C_{del}(1s) = c_D \times F \times N \times p \times 1s \tag{9}$$

appear, where c_D determines the cost of one store and retrieve operation in the circular array.

External noise can be introduced in clock-driven algorithms by either adding random external spikes or simulating a stochastic process. Simulating random external spike trains lets each tick of the clock trigger a random number of synaptic updates, while accounting for additional computational cost proportional to $F_{ext} \cdot N$.

Generally, a clock-driven algorithm is useful where the time step-based evaluation of each element of the simulation progress is cheaper, according to computational cost, than event-based communication.

Asynchronous or Event-Driven Algorithms In asynchronous algorithms, the state of a neuron and a synapse is only updated, if an event arrives, which can be spikes coming from neurons in the network, or external spikes. Such algorithms can be distiguished to such with instantaneous synaptic interactions and to such with non-instantaneous synaptic interactions.

In *instantaneous synaptic interactions*, spikes can be produced by a neuron only at times of incoming spikes, while timed events are stored in a *priority queue*. An iteration consists of: (see figure 2 in [2])

1. extracting the next event,
2. updating the state of the corresponding neuron,
3. checking if the neuron satisfies the threshold condition, and, if it does, insert the events in the queue.

As transmission delay algorithm, the FIFO[2] algorithm is used.

All in all, asynchronous algorithms with instantaneous synapses interactions are fast to implement.

In *non-instantaneous synaptic interactions*, spike times do not necessarily occur at times of incoming spikes, so that the algorithm implementation becomes more complex, illustrated in figure 3 in [2]. The following iteration of the algorithm guerantees, that the simulation is correct:

1. extracting the spike with the lowest timing from a provisory queue, which maintains a sorted list of the future spike timings of all neurons,
2. updating the state of the corresponding neurons and recalculate its future spike timing,

[2]First In, First Out

3. update the state of its target neurons,

4. recalculate the future spike timings of the target neurons

Considering transmission delays, another non-modifiable priority queue stores future synaptic events with their timings, which makes the whole system more complicate and difficult to implement.

The main difficulty in implementing such a priority queue is the fact, that scheduled outgoing spikes can be canceled, postponed or advanced by future incoming spikes, if the transmission delays exceed a determined value τ_{min}. Consequently, all outgoing spikes scheduled in the interval $[t, t + \tau_{min}]$ are certified, which speeds up the simulation.

Taking everything into account, the total computational cost comes up to:

$$C_{tot}(1s) = (c_U + c_S + c_Q) \times F \times N \times p \times 1s \tag{10}$$

with c_U as the cost of one update of the state variables, c_S as the cost of calculating the time of the next spike, and c_Q as the average cost of insertions and extractions in the priority queue(s). Obviously, the the simulation time is linear in the number of synapses, which is optional.

Like in synchronous algorithms, external noise can be introduced by either adding random external spikes or simulating a stochastic process. In this case, simulating random external spikes means adding a queue with external events. Since event-driven algorithms assume that the state of any neuron can be exactly calculated at every time, only simple pulse-coupled integrate-and-fire models or basic SRMs can come into question for a usage in an event-driven fashion.

Generally, an event-driven algorithm is useful where the event-based communication of the simulation progress is cheaper, according to computational cost, than time step-based evaluation of each element.

2.2.3 Overview of Simulation Environments

Reviewing software simulators, there is a manifold amount of publically-available and non-commercial simulation environments. In the following sections, the two simulators NEURON and NEST, which were used during this project, are discussed.

NEURON NEURON is a simulation tool for creating and using experimentally based biological models of neurons and neural systems. It is especially suitable for COBA models with complex anatomy, including extracellular potential near the membrane and several biophysical properties, such as different ionic channel types. The network size may reach from a part of a membrane to large networks of 10^5 neurons. A key attribute of neurons simulated with NEURON is the possiblity to spatially split up the cellular compartments, providing for a realistic representation of biological models.

One of the advantages of NEURON is the conceptual control, which is facilitated by features, like the *native syntax* of the *hoc* language, an extensive GUI, e.g. containing cell and network builders, and other programming interfaces, like Python or NMODL. Remarkably, once a neuron or a network has been created with the help of the GUI, it can be converted into a hoc file in order to reuse the model in larger networks. Furthermore, NEURON has a reputation for its computational robustness, accuracy, and efficiency. The simulation environment maintains both clock-driven and even-driven algorithms, while using the *Backward-Euler* or the *Crank-Nicolson* integration technique. NEURON has a range of operating systems it is working on. E.g. the simulator has several distributions for Windows, Mac and Linux operating systems. Besides, NEURON supports several kinds of parallel processing, e.g. distributed cell compartments or distributed networks.

A failed point is the online documentation, which leaves much to be desired. The examples are, in some places, too unprecise for newcoming neural network developers, thus, it is e.g. severe to create simple networks with point neurons, including synapse connectivity weights and probabilities, or an external random spike generator.

NEST NEST (NEural Simulation Tool) is another simulation environment primarily designed to simulate large neural networks up to 10^5 neurons with realistic connectivity and to guerantee strict reproducibility. Typical neuron models in NEST have one or a small number of compartments. As well, the simulator supports heterogenity in neuron and synapse types.

Regarding networks of real connectivity, the memory consumption depends on the number of synapses. As a result, NEST's focus is the efficient representation and update of synapses.

The primary language interpreter is the simulation language interpreter SLI, which has a high level expressive syntax. Furthermore, a Python interface called PyNEST has been created using the whole comfort of the Python scripting language. [16] presents felicitous documentations for PyNEST and SLI.

As a possible disadvantage, NEST prefers to maintain the, in many cases unprecise, clock-driven algorithm, while mainly using the *Runge-Kutta-Fehlberg* integration method. A graphical interface is missing, which, nevertheless, does not affect a user-friendly operability. As well, the SLI simulation language has, at the first view, a strange syntax, which takes much getting used to it.

Further information on NEURON or NEST, tutorials and the current release can be found at the NEURON web site [8] and NEST web site [16], respectively. A reference to network examples is given in the appendix A.1. As well, more examples are freely available at the ModelDB [12].

The features of other simulation environments are discussed in detail in [2].

2.2.4 Pros and Cons of Software Simulators

Pros One of the mentionable pros is the vast availability of simulation environments and freely-available documentations and examples of modeled neurons and networks, like the database ModelDB. Moreover, the time discretization e.g. serves as an opportunity, to interrupt an experiment and, later on, continue at this point, under exactly the same or quite modified circumstances, thus, creating an profitable experimental flexibility. When regarding a re-design of already existing models, the effort does not take much more than a simple programming practice.

Cons Firstly, the almost exact event-driven algorithm, as already mentioned, is very difficult to implement to more complex neural models, such as the HH model, and otherwise takes too much time to simulate. Therefore, in many cases, clock-driven algorithms have to be used in complicated neural networks. However, the disadvantage of software simulation tools concerning clock-driven algerithms is the fact, that the events occured during one of the time steps are recorded as coincident events, thus providing for an *artificial synchronization* , which decreases the exactness of the experiments. Consequently, the experimenter himself has to decide on using small and simple networks computed with an increased exactness, or, on the other hand, fast experiments of huge networks but with a low level of accuracy.

2.3 Neuromorhic Hardware

The Electronic Vision(s) Group at the University of Heidelberg has created a highly accelerated analog VLSI[3] model of leaky integrate-and-fire neurons with conductance-based synaptic plasticity. Furthermore, as an extension, a wafer-scale neural network model has been developed. This section describes facts about this neuromorphic hardware tools and argues their pros and cons.

The following information is mainly extracted from [3] and [15].

[3]Very Large Scale Integration: process of combining 10^3-10^5 transistor-based circuits into a single chip

2.3.1 Chip-Based Neural Network Model

Considering synaptic plasticity in biological models, the experimental timescales range from milliseconds to minutes, thus including seven orders of magnitude. In order to make this temporal range available to the experimentalist, a highly accelerated analog VLSI model containing implemented LIF neurons with COBA[4] synapses has been developed. The chip reaches an acceleration factor of 10^5, while recording the neural action potentials with a temporal resolution better than 30 μs biological time. This wide range will allow to study the different time domains from short term plasticity to long time learning, and possibly even evolution [7].

Chip Overview The existing versions of the so called FACETS Stage 1 hardware are built using a standard $180nm$ CMOS[5] process. The die size comes up to $25mm^2$ including a number of 384 neurons, each connected to a maximum of 256 conductance-based synapses. This maximum results from the size of the chip. Concerning the inter-neuron connectivity, action potentials (AP) are propagated as digital pulses, which are received by these synapses.

Membrane Potential Dynamics The utilized neural model corresponds to the described LIF model in section 2.1.1.

Synaptic Dynamics The synaptic weights are altered dynamically by the implemented STDP[6] algorithm and vary in the order of tens of milliseconds, thus slowly with time t. This *long-term plasticity* reveals a correlation measurement of the time Δt that has passed since the last pre- or post-synaptic action potential, and, hence, modifies the synaptic weight stregth.

On the other side, the maximum conductance $g_s^{max}(t)$ is governed by the so called *short-term plasticity*, which is only based on the pre-synaptic activity and emulates the limitation of resources involved in the synaptic transmission.

In the FACETS Stage 1 neuron model, *emitting a spike* means that a circuit separate from the neuron membrane releases a short voltage pulse, which is delivered to the synapse circuits of possibly connected target neurons. The neuron emits a spike as soon as an adjustable threshold voltage V_{thres} is exceeded. Once a spike has been released, the membrane potential is reset to an also configurable reset voltage V_{reset}, where it remains for the *refractory period* τ_{ref}, before it is excitable once again.

[4]conductance-based: simplest possible biophysical representation of an excitable cell: protein molecule ion channels are represented by conductances and the lipid bilayer by capacitors

[5]Complementary Metal Oxide Semiconductor

[6]Spike-Timing Dependent Plasticity

Detailed information about the layout and the operating principles of the chip, as well as synaptic plasticity, can be extracted from [15] and [3].

2.3.2 Wafer-Scale Neural Network Model

Wafer Overview The *wafer-scale model* is an extension of the chip-based FACETS hardware system introduced in previous section 2.3.1.

The wafer is produced by photolithographically applying sets of masks, which contain the spatial patterning information for determined production steps, on a round shape with a diameter of $200mm$. Concerning technical limitations, such a set occupies only a fraction of the full wafer area, the so called *reticle*. During the production, the reticle is replicated several times on one wafer.

Like the Stage 1 system, Stage 2 also uses a standard 180nm CMOS process. Each wafer represents a Stage 2 unit, consisting of at least 44 reticles, whereas the reticle itself contains 8 so-called HICANN[7] chips. On each HICANN chip, more than 115000 synapses and 8-512 neurons are implemented on a $50mm^2$ surface. Summing up, one wafer comprises up to 180224 neurons and over 40 million synapses. [3]

The implemented model is the *adaptive exponential integrate-and-fire* (aEIF) neuron model with COBA synapses [1], which is presented in section 2.1.2. Like the Stage 1 system, Stage 2 models cell bodies as point neurons, i.e. they do not consist of any compartements as in the actual biological system. All in all, the wafer reaches an acceleration factor of about 10^4 [1].

Membrane Potential Dynamics The utilized neural model corresponds to the described aEIF model in section 2.1.2.

Synaptic Dynamics Like in the Stage 1 system, COBA synapses are implemented, the dynamics of which can be described by a characteristic shape, consisting of a steep exponential rise followed by a plane exponential decay. The basic short-term plasticity mechanisms reveals hardly any differences to these in section 2.3.1. For long-term plasticity, a more flexible programmability of the weight modification functions is currently under development. For a detailed discussion of the connectivity and synapse model, see [3].

2.3.3 Pros and Cons of Hardware Emulators

Pros Firstly, the main advantage of the hardware emulation of neural network models is the high scalability arising from the chip's intrinsic parallelism of the circuit operations [3]. It is possible to emulate neural networks in real time or about 10^5 times faster, a value, which is only limited by the

[7]High Input Count Analog Neural Network

inter-chip event-communication bandwidth. This allows the experimenter to do extensive researches even for experiments requiring long biological time. A further advantage is the analog nature of the VLSI circuits, which increases the exactness of the experiment by preventing *artificial synchronization* as in the case of software simulators. Furthermore, compared to computing numerical solvers of differential equations, neuromorphic models have a low power consumption. To date, any stimuli, like Poisson distributed spike trains, can be implemented into the Stage 1 and Stage 2 hardware via the PyNN interface (see 3.1.2).

Cons One of the disadvantages of the hardware models is a limitation of flexibility, when thinking of changing the implemented model. In some cases, a reprogramming can handle this handicap, but often it requires a hardware re-design. Probably, the main disadvantage concerning the analog nature and the, compared to software simulators, missing temporal discretization is the impossibility to abandon an experiment at any point in time and restart it from this point again. Due to this fact, a dynamical interaction of a software simulator and a hardware emulator is a difficult issue. Besides, the parameter ranges are limited in consequence the low flexibility, and the experiments inclose unavoidable fluctuation noise.

2.4 Implications for the Presented Work

Taking all advantages and disadvantages of software simulators and hardware emulators into account, it is remarkable, that many avantages of the one approach are the disadvantages of the other and vice versa. Having this complementary nature in mind, the following chapter, firstly, names opportunities for interoperability methods involving several software simulators, and as an ensuing approach, presents a concept for such a software-hardware co-simulation of spiking neural network models with the MPI-based framework MUSIC.

3 Interoperability of Different Neural Modelling Tools

The development of simulator-independent modelling tools is a great task, considering the many different simulators with their different scripting laguages and graphical interfaces. Due to the fact, that the complex software packages must yield reproducible and comparable results, and, further, have hidden implementation-dependent flaws, the neuroscientific community would profit from the ability to easily simulate a model with multiple simulators, and, as a consequence, the fragmentation of researched effort would be reduced.

3.1 Simulator-Independent Model Specification

In the following, three different approaches to developing simulator-independent modelling tools are described, and relevant model environments are presented.

The subsequent ideas are mainly extracted from [2], [5] and [6].

3.1.1 NeuroML

NeuroML is an open-source cooperation created mainly to support, develop and extend the use of declarative specifications for models in neuroscience, using an XML standard, which has the great convenience to be both human- and machine-readable. There are several standards already developed, like MorphML (neuroanatomy), ChannelML (models of ion channels and receptors), BiophysicsML (compartmental cell models) and NetworkML (cell positions and connections in a network).

Such a declarative model specification has the advantage of the simplicity in setting up models and guarantees a well-defined behaviour of distinct cell compartments. Another attractive feature is that the language NeuroML is not fixed forever, but easily extended, as far as new models are compatible with original NeuroML specifications.

Otherwise, the declarative approach has the disadvantage of less flexibility, considering the fixed library of neuron models, synapse types, plasticity mechanisms etc. Furthermore, it does not support every model yet, e.g. the I&F-type is still under development [2].

Regarding the usage of NeuroML with specific simulators, the source code has always to be translated either by accepting NeuroML documents as input, while these are translated by the simulator, or applying the XSL Transformation language to generate native simulator code (e.g. hoc and NMODL in the case of NEURON).

Additional information can be looked up in [13].

3.1.2 PyNN

As a programmatic alternative to NeuroML, PyNN [4] tries to obtain more flexibility and a simple conversion between simulators. It allows to write a simulation code for a model once, and then run the code on multiple simulators, by defining a Python-based meta-language, which is either translated by individual simulation engines into simulator-specific code, or controls the simulator directly.

PyNN's API[8] consists of two parts, a low-level API (functions *create()*, *connect()*, *set()*, *record()*, etc.), which is good for small networks and a high flexibility, and a high-level API (classes *Population* and *Projection*, both have different specification methods), which is designed to hide the details and the bookkeeping, and have a one-to-one mapping with NeuroML. As well, PyNN possesses standard cell models, that can be easily translated into simulator-specific models.

Regarding the usage of PyNN with specific simulators, it currently supports NEST(via PyNEST), NEURON (via nrnpython), PCSIM, Brian and the FACETS hardware. A key point is the Python scripting language, which is well human-readable and simply to handle.

Detailed information, well-made documentations and examples can be looked up at [4] and [14]. As well, appendix A.1 names further links for PyNN examples.

3.1.3 MUSIC

In constrast to NeuroML and PyNN, MUSIC is a standard API allowing large scale neuron simulators using MPI[9] internally to exchange data *during* runtime. Especially, it is designed to perform data transport of high bandwidth and low latency within a *cluster environment*, while running a *multi-simulation* (both see appendix A.2).

Ensuring, that the existing simulators can be easily adapted to it, MUSIC provides for a *run-time interoperability* by allowing models written for differnet simulators to be simulated together in a larger system, whereby each individual application does not need special adaptation to specific properties of other applications.

As well, MUSIC guarantees the *re-usability* by providing a standard interface, using C++, which is the standard for current high-end hardware. This allows to build larger neural networks, without re-implementate them into other simulation scripts. For example, this re-implementation would be a problem, if a certain software does not support the favourite model.

As a key point, it should be possible to add MUSIC library support without invasive restructuring of the existing code. For example, in future

[8]Application Programming Interface

[9]Message Passing Interface

releases, PyNN could be extended to support multi-simulations using the MUSIC library, e.g. publishing the named ports.

The further description introduces MUSIC's operating principles and main features. [5]

Phases of Execution A MUSIC-controlled multi-simulation is executed in three different phases:

The *launch* phase starts applications on the participating processors. In this time in particular, MUSIC is responsible for distributing and launching the application binaries on the set of MPI processes. As well, for different MPI implementations, the accession of the command line argument of the MUSIC launch utility and the determination of the process ranks have to be handled separately before MPI is about to be initialized. The *launch* phase begins when `mpirun` launches the MUSIC binary, for example by typing: `mpirun -np _ music demo.music`. In this case, `demo.music` is an example for a configuration file and `-np` determines the number of participating processes represented by _ (see A.1 for examples), while *music* is the special launcher program, utilized by MUSIC. In [5], page 15f., a common configuration file is described.

During the *setup* phase, applications are allowed to publish ports, the time step they will use, and where data will be present. In a further step, MUSIC establishs all the selected connections. The configuration parameters can be read from the configuration file (see A).

In the *runtime* phase, the simulation time of applications is kept in a consistent order, via *tick* calls at regular intervals in simulated time, which is handled by each of the applications. Only at these *tick* calls, MUSIC is allowed to use MPI to transfer data. This fragmentation into three different phases of execution might have the disadvantage, that once a simulation has reached the *runtime* phase, it is not possible to, for example, change back to the setup phase and create new ports.

Distribution of Data MUSIC handles data transfer between applications that use different timesteps and different data allocation strategies. To manage the amount of data to transfer, MUSIC uses *shared global indices* in order to enumerate the complete set of data to be sent over the connection, while each particular MPI process stores data using local indices. The *index map* maps local indices to global ones. The *data map* includes an index map, but also contains information about the residence within the memory, the local data structure, and the type of data elements. Data to be transferred can be regarded as a large array distributed over multiple processors.

Timing Considerations By using a global micro-timestep common for all applications and an internal schedule, MUSIC ensures, that data is delivered at the appropriate time, concerning the fact, that different applications use different timesteps. If loops occur during the communication, MUSIC handles the transfer via *acceptable latency*, handled by the input ports, and thus, allows for data arriving late. The receiving application declares how late, according to simulation time, data may arrive, and, hence, specifies a *delay* to fullfill the purpose.

Furthermore, MUSIC, tries to minimize *handshaking* in order to decrease unnecessary data traffic, which would charge the execution time of the multi-simulation. Both parts of a connection pair locally calculate, when the actual data transfer over MPI takes place. Since MUSIC uses *blocking communication* (see appendix A.2), one of the applications will, in practice, have to wait for the other to reach the same point in its execution.

Further facts concerning time considerations are outlined in section 3.2.

Ports Each application declares its ability to produce and consume data by publishing ports, which are regarded as so called *proxy-objects* and provided with information about the datatype (continuous data, spike events, messages). Ports are either sinks (*input ports*) or sources (*output ports*). Pairs of ports form a *connection.* Data is transferred over the connection from the producer to the consumer. Input ports (w.r.t. MUSIC) can only have one connection, output ports can be connected to multiple input ports.

In the current version of MUSIC, three kinds of ports are used:

To control the communication of multidimensional timeseries, e.g. membrane voltages, *continuous ports* are utilized.

The communication of spikes will use *event ports*, which call special functions to send and receive individual spike events. The procedure of receiving spikes requires a kind of sorted buffer (in this case a *priority queue*). An event, in this case, is a pair of an index identifier, consisting of either a global or a local index, and a double precision floating point time-stamp.

Additionally, MUSIC uses *message ports* to allow for controlling information between applications, e.g. in the form of time-stamps. As well, parameters can be altered or stimuli turned on via message ports, externally. Every receiver on the receiver side has to announce its willingness to achieve messages form the sender side.

Detailed information about the API, an instruction to adapt existing applications, and a complete example to demonstarate the usage of MUSIC (but not MUSIC with a neular simulator) can be looked up at [5]. Future plans are well described in [6].

Pros & Cons of MUSIC The main advantages already were described in the pleluding MUSIC description (see 3.1.3).

A disadvantage of MUSIC is the lack of broad user experience and available documentation as well as examples (personal experience, or further reports from [6]). Furthermore, MUSIC only supports the simulators NEST and MOOSE up to date. The adaptation of NEST (via PyNEST) and MOOSE into MUSIC is well-described in [6].

3.2 Interaction Concepts using MUSIC

With regard to the figures presented in [5], the following approaches are developed only by taking the written description of MUSIC into account, but matching it perfectly.

Firstly, a sample timing consideration for a software-software interaction via MUSIC, as it ought to operate, is described in detail. The example network in this gedankenexperiment is depicted in figure 1. A and B are two different simulators, simulating neural networks, individual neurons or cell compartments with different speed and different timesteps, while exchanging spike events via MUSIC, which provides for delays (including MUSIC and MPI delays), thus, possibly representing axonal delays within the model.

Subsequently, an idea for a software-hardware spike interchange is suggested, and a feasibility study is thought through. At the end, an instruction suggests the preparing that work should be done and issues that should be taken into consideration when actually implementing the presented concept.

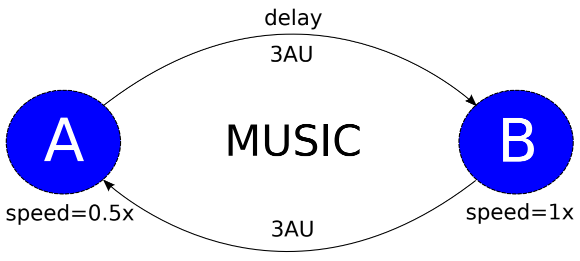

Figure 1: **Sample Simulation.**

3.2.1 Software-Software Interaction

The following description submits some useful theoretical aspcets thinking of a parallel simulation with spike interchanging.

Concept A software-software interaction (e.g. via MUSIC) could be well described by the following pseudo code:

```
 1 // software pseudo code
 2 // w.r.t. biological time
 3 // T_sim: simulation duration, defined externally
 4
 5 t_rec_last = 0; // t_rec_last: last recorded time from opposite sim.
 6 t = 0;  // t: own time
 7
 8 do {
 9   send own time
10
11   if send-queue.size > 0:
12   {
13     send events
14     set wait-ack true
15   }
16
17   do {
18     // t_rec_update: updated recorded time of the opposite simulation
19     // T_tick_opp: duration between ticks of the opposite simulation
20
21     check inport
22     if event arrived: store event, send ack
23     if ack arrived: set wait-ack false
24     if t_rec_update arrived: set t_rec_last = t_rec_update
25   } while ( (t >= t_rec_last + T_tick_opp) || wait-ack )
26
27   continue simulation until next tick,
28   apply received events at appropriate time
29   t = t + T_tick  // T_tick: duration between ticks, defined externally
30
31 } while(t <= T_sim)
```

The code consists in the main of a `do/while` loop, which advances until the *simulation time* (see appendix A.2) is over.

At a *tick* call, MUSIC forces the simulator to send its own *biologcal time t*, as a kind of control message, to let the participating target simulator know about its own progress (line 9). Furthermore, the *events* stored in the priority queue, which have occured during the last simulation step, are also transmitted to the target, setting the "waiting for an acknowledgement"-bool (`wait-ack`) true (lines 13f). After this procedure, the simulator remains in the *tick* position, while awaiting both, the acknowlegment of the broadcasted *events* and the right moment to proceed until the next *tick* (line 25). Concerning this right moment, the time `T_tick_opp` can be arbitrarily aligned, according to the minimum time mismatch to reach during the experiment. In this case, MUSIC determines a minimum tolerance time. In the examples below (Figure 2 and 3), `T_tick_opp` always is the time between two *ticks* of the opposite simulation.

As long as the simulator remains in this stationary situation, the input ports are checked (line 21). Now, the simulator is prepared to receive external *events*, *time messages* and *acknowledgements*, produced by the parallel simulation (line 22f). Arriving *events*, all of which have time stamps, are buffered in a queue, and an acknowlegdement is sent to approve the arrival. Furthermore, the received *time message* (`t_rec_update`) replaces the existing one (line 24).

If the time condition `t >= t_rec_last + T_tick_opp` is reached, and the *acknowledgements* of all the sent *events* have arrived, MUSIC permits the simulator to continue until the next *tick*, while applying the received *events* at the appropriate or at the best possible time.

Example Run Figure 2 shows an example run for a parallel simulation of two simulators A and B. The dark red areas represent the time during which the simulators are actually executing their tasks, while the light blue areas stand for the *tick* phase, when the simulators are stagnating. The light green circles depict generated spike *events.* The simulation time evolves top down in the schematic. Concerning the *wall clock time* and the *biological time*, the precise time scales are not important for this thought experiment, but in fact, the simulation usually will last longer as pictured here, thus, the brown rectangles would be stretched in time. The numbers beside of the blue areas represent the *biological time.* Bold arrows highlight the actual transport of the *events* (ev) and time control messages (t), dashed arrows reveal the *acknowledgements* (ACK), and dotted arrows simple *time messages.* All the message types are taking an axonal delay of 3 arbitrary time units (AU) into account, which is handled by MUSIC via MPI.

Here, B runs two times as fast as A, while performing two times as large timesteps (`T_tick(A)` $= 2ms$, `T_tick(B)` $= 4ms$). The starting time of both simulations is chosen freely, and does not affect the actual multi-simulation progress (it has only an influence on the duration of the *tick* calls at the beginning).

In the presented example, A and B are starting at the same time. At the *wall clock time* $2AU$, both have their first *tick.* Thus, they are sending their *events* and, afterwards, are waiting for incoming *events*, *time messages* or *acknowledgements.* At $8AU$ (w.r.t. *wall clock time*), simulation A, which stagnates at *biological time* $t = 2ms$, has received its *acknowledgement* (wait-ack is set false), and, therefore, is allowed to proceed, since the last recorded time from B is $4ms$, so the inequation is wrong. On the other side, simulator B also gets the *acknowledgement*, but does not fulfill the time condition in the inner `while` loop. Hence, B remains checking the input ports till such time as the time condition is wrong.

Another interesting procedure is happening at *wall clock time* $18AU$. Here, A's previous simulation step does not contain any *events.* Thus, the

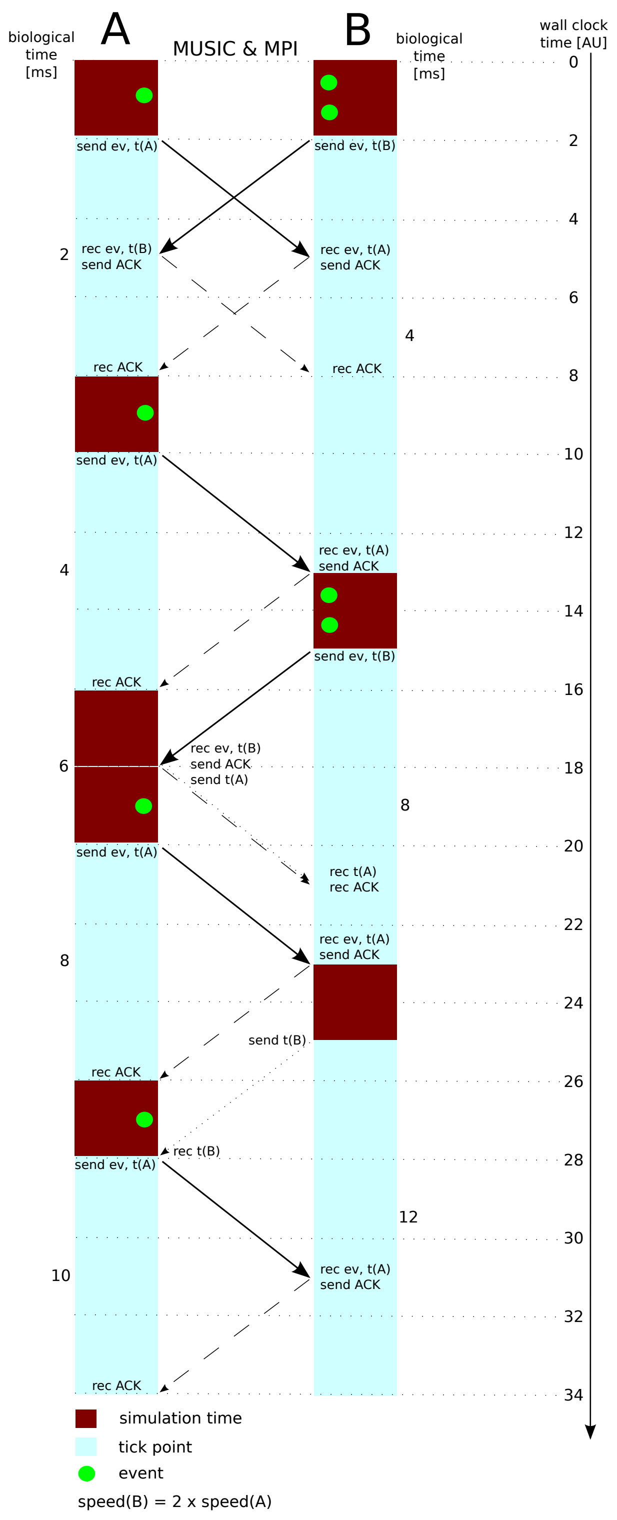

Figure 2: **Software-Software Interaction Example.** Simulator B is 2 times faster and uses a 2 times larger timestep than simulator A.

simulation has not to wait for *acknowledgements*. Knowing, that B has simulated $4ms$ of *biological time*, the time condition still is false, which makes simulator A continue immediately.

Considering the precision of our gedankenexperiment, the tolerance limit is $4ms$, as can be seen by checking the arrival of A's *event* with the time stamp $1ms$ (see green circle at *biological time* $1ms$ - this fact requires, that the simulation has the same speed during the execution, which is not that realistic, but in this case irrelevant). The first point in time, when this *event* actually can be registrated by B is $4ms$ (*biological time*). In an extreme case, this *event* might be happened directly after starting, which explains the tolerance limit.

To reduce this limitation of precision, the experimenter should preferably use equal timesteps `dt`. In order to avoid further imprecision, these timesteps should be as small as possible, which, though, decelerates the whole experiment (see section 2.2).

For *events* arriving before their time, MUSIC guarantees, that they are applied at the appropriate time.

3.2.2 Software-Hardware Interaction

Concept Now, one of the previous simulators is exchanged by a hardware emulator. The software still functions as described in the upper part.

Considering the analog nature of the hardware, and therefore, the mentioned inability to interrupt and continue the emulation progress plus the rapidity of the model dynamics (see section 2.3.1), the software-specific pseudo code has to be adapted to the hardware's features, still making it possible to interchange spike events with a software simulator:

```
 1 // hardware pseudo code
 2 // w.r.t. biological time
 3 // T_sim: simulation duration, defined externally
 4
 5 dt     // dt: simulation timestep (ms)
 6 t_next = 0  // t_next: duration of the next run (ms), def. externally
 7
 8 while (t_next < T_sim) {
 9
10   send own time
11
12   if send-queue.size > 0:
13   {
14       send events
15       set wait-ack true
16   }
17
18   t_next = t_next + dt
19
20   do
21   {
22     // t_rec_last: last recorded time from (opp.) simulation
23     // t_rec_update: updated recorded time from (opp.) simulation
24     // T_tick_opp: duration between ticks of the (opp.) simulation
25
26     check inport
27     if event arrived:
28     {
29       store event into (expanding) event playback memory
30       send ack
31     }
32     if t_rec_update arrived: set t_rec_last = t_rec_update
33     if ack arrived: set wait-ack false
34   } while ( (t >= t_rec_last + T_tick_opp) || wait-ack )
35
36   run hardware experiment: t = 0..t_next,
37   use extended event playback memory
38 }
```

This concept was created and deliberated during the documented internship project. It requires an external loop, which controls the emulation between the starting and the varied stopping points, and additionally, activates the hardware's *event playback memory* (epm).

In contrast to the simulator's activity, the incoming *events* are not only stored in a provisory queue, but in a spike train expanding at every *tick* (line 29).

Another difference is the fact, that the hardware has to be restarted at every *tick*, if the time condition in line 34 is wrong. After every restart, the hardware runs the experiment again, this time a timestep dt longer than the run before, while using the extended playback memory to insert the previous as well as the new *events* into the experiment. This time, a `while` loop is used (line 8), until the *emulation time* `t_next` reaches the entire duration of the experiment `T_sim`. At every *tick*, `t_next` is increased by `dt`, which can be defined externally. Here, the mentioned times also refer to *biological time*.

The remaining progress does not differ from that of the software-software co-simulation: At *tick* call, MUSIC provides for the interchange of the own *biological time* and the occurred *events* stored in the sending queue. Subsequently, the emulator awaits the *acknowledgement*, and, as well, checks the, already described, time condition `t >= T_tick_opp + t_rec_last`. As long as the hardware is in the waiting phase, MUSIC forces it to check the input ports for incoming *events*, updated progress times and *acknowledgements*.

Considering the outgoing spikes, the external program has to make sure, that each time only the spikes occurred during the last timestep `dt` have to be transmitted to the appropriate software simulation step.

Example Run Figure 3 depicts an example of a software-hardware (S-H) run-time interaction provided by MUSIC. Here, again, the simulation or emulation run is coloured dark red, the *tick* times light blue and the events are light green.

Compared to figure 2, the added gradient in the emulation procedure reveals, that the hardware restarts at every *tick*, and, runs an additional timestep dt further (the red area becomes lighter).

In contrast to a real experiment, the emulation speed is illustrated as 4 times faster than the simulation speed. Actually, the hardware emulation would exceed the simulation by several magnitudes (see section 2.3.1), which could not be pictured in this case. In fact, the red areas on the hardware(H) side would shrink, and compared to these, the *ticks* would take the main part of the hardware's experiment.

The rest of the experiment proceeds in exactly the same manner as the previous example has already shown.

Due to incompleteness of the MUSIC interface and a lack of time and know-how, an actual software-hardware interaction could not be realized up to date, but a software-software communication via MUSIC using PyNEST could be observed. In one of the experiments, the intern MUSIC-eventgenerator generates randomly distributed spike events, and is connected to a spike detector in a separate PyNEST-file and, additionally, to the MUSIC-eventlogger via MUSIC. After running the experiment, both the spike detector and the eventlogger display the same arriving times of the recorded events. The path to the MUSIC-configuration file `events_in_out.music` and the launch file `nestlauncher.sh`, through which MUSIC can access the PyNEST script, can be looked up in appendix A.1. In another experiment, which was named `sd.music`, an integrate-and-fire neuron, which is influenced by a Poisson distributed spike train containing excitatory and inhibitory stimuli, is connected to a spike detector via the MUSIC interface.

In the next section, the preparing work for similar software-hardware experiments is discussed.

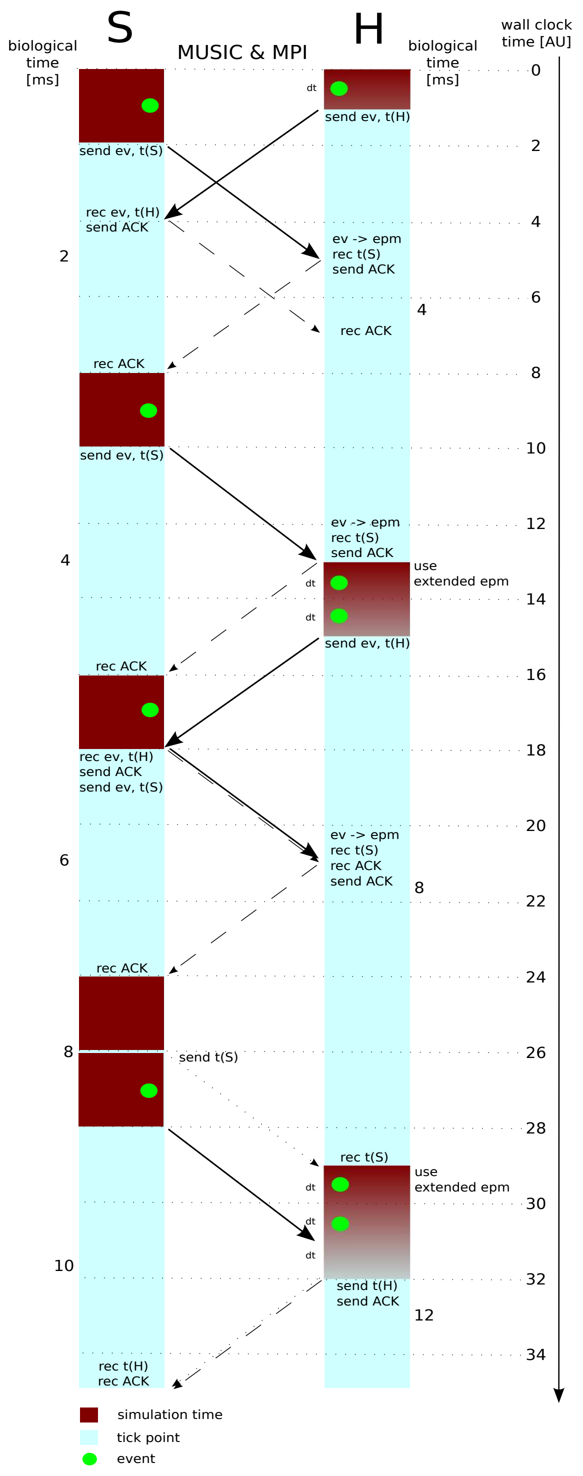

Figure 3: **Hardware-Software Run-Time Interaction Example.** Emulator H is 4 times faster (which is very understated) and uses a 2 times larger timestep than simulator S.

4 Conclusion and Outlook

Reminiscing about the the previous section, a software-hardware run-time interaction of spiking neural network models via the MUSIC interface would be a possible endeavour. But, before experiments can be run, there are still several things to be realised and deliberated.

Firstly, a preferably simple multi-simulation via MUSIC has to be got to work. Therefore, the MUSIC implementations have to be created and documentated for the simulators, which are about to be used. As a pilot test, one could try to let a I&F neuron, which is excited by a set of Poisson distributed spike trains, and a *parrot* neuron communicate. The only thing a *parrot* neuron does is emitting the received spike events. The location of the already prepared SLI code and the MUSIC configuration file can be extracted from appendix A.1, but still some key words are missing.

Secondly, MUSIC classes should be built into the PyNN language, enabling the PyNN software to instantiate `music_out_proxy` and `music_in_proxies`. This is the only way to make the hardware accessible for MUSIC up to date. Having done this "preparing work", the experimenter could try to build larger networks using PyNN, like the already created one (see A.1). Here, two different neural populations (excitatory and inhibitory) are connected with themselves via synapses with different weights `w` and connection probabilities `p`. As well, a set of external Poisson distributed spike trains, also consisting of excitatory and inhibitory spike sources, is affecting the excitatory population (Figure 4). After connecting the two populations via MUSIC, a noticeable decrease of emitted spike events of the excitatory population should be observed.

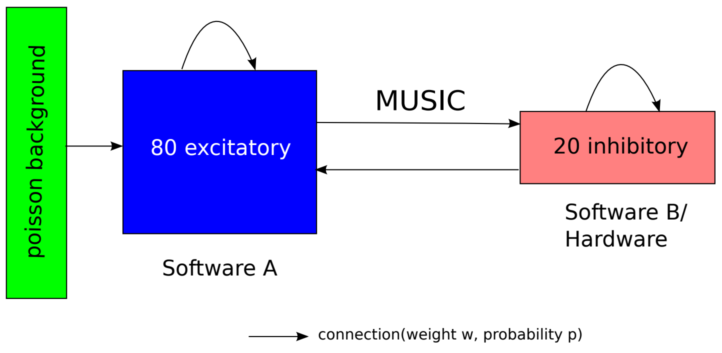

Figure 4: **Network Example.**

Thirdly, the previous hardware pseudo code should be implemented, which lets the hardware restart at every tick, provides for an extendable event playback memory, and a, by `dt`, extended simulation duration.

As a further step, if a software-hardware run-time interaction has succeeded, the experimenter has to determine, what spikes actually are interchanged. For example, after each emulation restart, the whole already produced spike train could be transmitted to the software simulation. Here, it should be asserted, that MUSIC has only the permission to make use of the latest outcoming events. Otherwise, MUSIC could also make sure, that events expire, if they arrive much too late (in *biological time*) on the other communication side.

All in all, a software-hardware run-time interoperability of spiking neural network models is a possible issue, but there is still a lot of preparation work to be arranged.

A Appendix

A.1 Reference to Sample Networks

The example network models can be retrieved from by Dr. Daniel Brüderle. Among these are models of neurons or simple networks built up with NEURON (hoc files), NEST (PyNEST and SLI) and PyNN. Each of them has got a header with a short description. These models can be viewed in the respective directories found in the `/basic` directory. As well, the directory contains the pictures presented within this documentation and further material.

More examples can be looked up in the corresponding example folders shipped with each freely-available simulation environment mentioned in this context.

A.2 Useful Terminology

Blocking Communication In a *blocking communication*, each of the participating systems in a parallel simulation is waiting for acknowledgements of the partners after having sent data to parallel processes. During this waiting time, the sending simulator pauses until a receiving message has arrived. Taking into account that parallel processes contain delays, this fact can have a vast effect on the simulation duration, or even lead to a deadlock.

Cluster Environment In a *cluster environment*, the workload of the models to simulate is distributed across multiple machines. For example, each of the simulating participants simulate a part of the cell or a neural population.

Delays While, in real world, physical delays e.g. depend on the quantities of the speed of light or represent axonal delays in neural transmission mechanisms, in the simulation world, delays do not need to be affected by the simulation models. Thus, time anomalies may occur. Therefore, a time stamp often has to be assigned to sended events, or messages, to determine a correct time ordering of events.

On the one hand, this correction provides simulation models' reproducibility, but, on the other hand, increases the simulator's latency time, thus, incresing the temporal duration of the simulation process. [11]

Handshaking *Handshaking* usually is a protocol-based mechanism in parallel computing that, firstly, produces a communication channel, on which all participating processes have to agree on the data to send, before the actual data transfer begins. Though this process is very safe, it provides an emormous additional amount of transfer traffic.

Multi-Simulation A *multi-simulation* is a parallel execution of multiple applications.

Time Types Generally, it is useful to get familiar with the three different time types concerning simulation and emulation procedures.

The *physical time* (or, in this case, *biological time*) is the real temporal duration of a natural process, which is about to be modeled.

In contrast to this, the *simulation time* is the simulators representation of time. For example, the actual time of the natural process can be represented during the simulation as floating point values, whereby e.g. a simulation time unit stands for an hour of physical time.

The so called *wall clock time* expresses the actual temporal duration of any simulation process, which is measured by an external clock that is not influenced by the simulator. [11]

References

[1] R. Brette and W. Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.*, 94:3637 – 3642, 2005.

[2] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris Jr, M. Zirpe, T. Natschlager, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. El Boustani, and A. Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies, 2006.

[3] Daniel Brüderle. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System.* PhD thesis, 2009.

[4] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.*, 2(11), 2008.

[5] Mikael Djurfeldt and Örjan Ekeberg. MUSIC - Multi-Simulation Coordinator - Users Manual. 2009.

[6] Mikael Djurfeldt, Johannes Hjorth, Jochen M. Eppler, Niraj Dudani, Moritz Helias, Tobias C. Potjans, Upinder S. Bhalla, Markus Diesmann, Janette Hellgren Kotaleski, and Örjan Ekeberg. Run-Time Interoperability Between Neuronal Network Simulators Based on the MUSIC Framework. *Front. Neuroinform.*, 2010.

[7] FACETS. Fast Analog Computing with Emergent Transient States – project website. `http://www.facets-project.org`, 2009.

[8] Michael Hines, John W. Moore, and Ted Carnevale. Neuron, 2008.

[9] Alan Lloyd Hodgkin and Andrew F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, 117(4):500–544, August 1952.

[10] Integrate-and-Fire Model. Website. `http://icwww.epfl.ch/~gerstner/SPNM/`.

[11] Introduction to Time Management. Website. `http://www.sisostds.org/webletter/siso/iss_35/art_197.htm`.

[12] ModelDB. Website. `http://senselab.med.yale.edu/modeldb`, 2008.

[13] NeuroML. Model Descriptions for Computational Neuroscience – website. `http://www.neuroml.org/`.

[14] PyNN. A Python package for simulator-independent specification of neuronal network models – website. `http://www.neuralensemble.org/PyNN`, 2008.

[15] J. Schemmel, D. Brüderle, K. Meier, and B. Ostendorf. Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07)*. IEEE Press, 2007.

[16] The NEST Initiative Website. `http://www.nest-initiative.org`, 2009.