Journal of Intelligent & Robotic Systems manuscript No.
(will be inserted by the editor)

A Framework for Coupled Simulations of Robots and Spiking Neuronal Networks

Georg Hinkel · Henning Groenda · Sebastian Krach · Lorenzo Vannucci · Oliver Denninger · Nino Cauli · Stefan Ulbrich · Arne Roennau · Egidio Falotico · Marc-Oliver Gewaltig · Alois Knoll · Rüdiger Dillmann · Cecilia Laschi · Ralf Reussner

Received: April 4, 2016/ Accepted: not yet

Abstract Bio-inspired robots still rely on classic robot control although advances in neurophysiology allow adaptation to control as well. However, the connection of a robot to spiking neuronal networks needs adjustments for each purpose and requires frequent adaptation during an iterative development. Existing approaches cannot bridge the gap between robotics and neuroscience or do not account for frequent adaptations. The contribution of this paper is an architecture and domain-specific language (DSL) for connecting robots to spiking neuronal networks for iterative testing in simulations, allowing neuroscientists to abstract from implementation details. The framework is implemented in a web-based platform. We validate the applicability of our approach with a case study based on image processing for controlling a four-wheeled robot in an experiment setting inspired by Braitenberg vehicles.

G. Hinkel, H. Groenda, S. Krach, O. Denninger, S. Ulbrich, A. Roennau, R. Dillmann and R. Reussner
FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
E-mail: {hinkel,groenda,krach,denninge,sulbrich}@fzi.de
{roennau,dillmann,reussner}@fzi.de

L. Vannucci, N. Cauli, E. Falotico and C. Laschi
The BioRobotics Institute at Scuola Superiore Sant'Anna (SSSA), viale Rinaldo Piaggio 34, 56025 Pontedera, Italy
E-mail: {l.vannucci,n.cauli,e.falotico,c.laschi}@sssup.it

M. Gewaltig
Ècole Polytechnikum Fèdèral de Lausanne (EPFL), Station 1, 1015 Lausanne, Switzerland
E-mail: marc-oliver.gewaltig@epfl.ch

A. Knoll
Technische Universität München (TUM), Arcisstraße 21, 80333 München, Germany
E-mail: knoll@in.tum.de

Keywords Neurorobotics · Human Brain · Spiking Neuronal Networks · Domain-specific Languages · Model-driven Engineering

1 Introduction

Bio-inspired robots such as the walking machine LAURON V [28] often use classic robot control software whereas the robots parameters such as the kinematics are adapted from nature. This can be problematic as classical controllers require to express for instance the kinematics of the robot explicitly. For example, the kinematics of LAURON V is inspired by the stick insect Carausius morosus with four joints per leg, depicted in Fig. 1. This amounts to 24 degrees of freedom to control the legs, which is fairly difficult to express explicitly.

However, advances in neurophysiology often offer inspiration not only for parameters such as kinematics but also for robot control algorithms. Spiking neuronal networks mimic nature's behavior in detail and can be used to replace parts of or the entire robot control software, utilizing the ability of neural networks to learn and adapt.

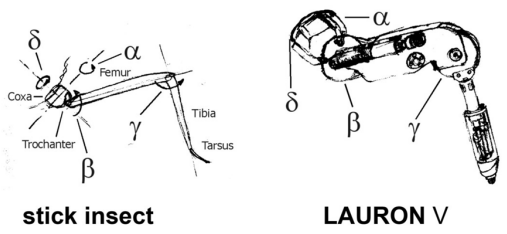

Figure 1 The kinematics of LAURON V compared to the stick insect Carausius morosus [28]

Contrary, the integration especially of spiking neuronal networks in robot control also yields a possibility to validate our understanding of how biological neural networks are connected to actuators in nature. This is especially interesting for neurophysiologist research.

From a researchers point of view, such an integration requires frequent adaptations of the wiring between a robot's sensors, the network and the robot's actuators. However, the multitude of technical problems involved in running a robot and, last but not least, also the price for more complex biologically inspired robots with many joints pose a large obstacle for experiments integrating neuronal network models into the robot controllers. Therefore, an integrated simulation platform that allows users to concentrate on the connection between the robot and the network, leaving aside technical implementation details, is beneficial both for neurophysiology and robotics.

To the best of our knowledge, existing approaches do not sufficiently bridge this gap between robotics and neuroscience. The simulation of experiments is often hand-crafted, resulting in duplicated code to couple the simulations. Furthermore, such simulation scripts must be adapted if the simulator underneath changes.

In this paper, we present a framework to support coupled simulations of robots and spiking neural networks through the metapher of Transfer Functions. To focus on the specification of the wiring between the neuronal network and the physics simulation, we created an architecture independent of both the experiment simulated as well as the used simulators. Further, we designed PyTF, a Domain-Specific Language (DSL) on top of it. This DSL concisely captures the connection between a robot and the network in Transfer Functions while the architecture underneath allows adjusting parameters of the connection or the network during a running simulation.

The paper extends prior work on PyTF that discussed the applicability of model-driven engineering for the coupled simulation of robots and spiking neural networks [15]. Here, we explain the concepts of PyTF and the software architecture underneath from which the language abstracts.

Our approach is implemented in the Neurorobotics Platform (NRP) [15,31]. This simulation platform fosters the research of neuroscientists, especially neurophysiologists, by providing an integrated simulation platform for the simulation of robots and their physical environment coupled to biologically plausible spiking neuronal networks. It is based on existing open-source implementations of simulators for the neuronal network (Nest [11]) and the robot and its physical environment (Gazebo [18] and ROS [26]). Aside the coupling of simulations, the NRP also consists of a library of robots such as the above mentioned LAURON V, the humanoid iCub robot, a four-wheeled Husky[1] robot and a controllable model of a mouse. We also provide editors for all artifacts of a coupled simulation such as robots, environments, neuronal networks and their connection. As the NRP is a web-based application, neuroscientists can use the simulation platform as well as most editors for the simulation models conveniently without any local installation. However, these artifacts are out of scope of this paper and are thus not described further here.

We applied our approach in a case study where we migrated a classical robot controller for a Husky robot with a mounted camera to a neural implementation in two steps, demonstrating the applicability of our approach. We selected this case study because of its simplicity, though the NRP has been used for more sophisticated experiments such as visual tracking [31].

The remainder of the paper is structured as follows: Section 2 discusses related work. Section 3 presents and discusses the language PyTF to specify coupled simulations of spiking neuronal networks with robots in Python. Section 4 introduces the framework architecture underneath PyTF to implement these coupled simulations. Section 5 presents a case study evolving a classical controller for a simple four-wheeled robot with a mounted camera to a neural implementation. Finally, section 6 concludes the paper and provides an outlook on future research.

2 Related Work

Approaches of simulating neuronal networks to control robots can be traced back at least until the early 1990ies [25]. Nevertheless, to the best of our knowledge, all approaches required hand-crafted solutions to couple robot sensorimotor functions and brain simulation. There exists no dedicated approach to facilitate this interplay. While there are DSLs targeting either the neuronal network simulation or robotics, our language is the first to describe their interplay on a high abstraction level.

In the remainder of this section, we present the related work in several areas in more detail.

2.1 Evolution of classical robot controllers

The transition from using classic robot controllers to spiking neuronal networks can be found in various works. For instance, Hagras et al. [13] implemented a spiking

[1] http://www.clearpathrobotics.com/husky/

neural network based robot controller and described an experiment involving a wheeled robot which follows along the edge of a wall using ultrasound sensors. Nichols et al. describe a similar experiment [22] involving a more complex scenario including behavioral learning. The Braitenberg vehicle inspired experiment we are using in the case study is much simpler, but we see this only as a case study to validate our framework for coupled simulations.

2.2 Semi-automated Evaluation of Robot Controllers

Multiple approaches target the iterative evaluation of robotic controllers through simulation approaches [2, 19]. However, these approaches do not consider the robot controller but treat it rather as a black box. Therefore, no coupling is in place.

2.3 DSLs for Neuroscience

In the field of neuroscience, DSLs can be found in the NEURON simulator [14, 7], whose network models are based on Hoc [16]. Strey [30] presents a language to describe neuronal networks to enable code generation for efficient simulations. More recent, current research projects focus on describing the structure of spiking neuronal networks [6, 12, 27] and allow for a detailed description of neuron models [24, 27]. The languages can be regarded as complementary to our approach as they do not describe data transfer to entities outside the simulated brain, while our approach relies on a formal representation of the brain model.

2.4 DSLs for Robotics

Despite software playing a basic role in implementing the functionality of robotics systems, most robotics software systems are still hand-crafted based on frameworks. However, in recent years a migration from code-driven approaches towards more flexible model-based ones started to emerge [29, 1].

Several works [10, 4, 8] have been proposed for DSLs in robotics, covering some specific aspects of robotic software systems. Nordmann et al. have published a list of DSLs in robotics[2] and created a survey [23]. Most of these languages utilize the knowledge of a particular sub-domain of robotics to create an abstract syntax and a DSL for it. These DSLs target the generation of entire robot controllers or at least large parts of them. This is different to our approach where we assume the robot controller exists as a neuronal network that needs to be integrated with the robot.

Rather focused on the implementation, the DSL by Moghadam et al. for the ATRON self-reconfigurable robot system also contains an internal DSL embedded in Python [21]. However, their usage of Python is different to our approach. As they do not reuse semantic of the Python language, there seems to be no reason in favor of using their Python DSL over their external DSL.

3 A Python DSL for Transfer Functions

In this section, we present the language and its abstractions that we use to specify the connection of robots and spiking neural networks.

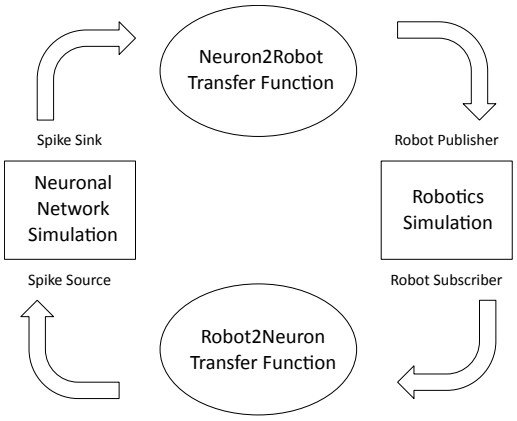

Figure 2 A closed loop between spiking neural networks and robots (sketched)

Our main metapher for connecting spiking neuronal networks with robots are Transfer Functions such as sketched in Fig. 2. Transfer Functions consist of connections into the simulators and an executable specification of how the data of one simulator should be transmitted to the other. Ideally, the executable part is trivial as the purpose of most Transfer Functions is limited to transmission, simple arithmetic adjustments and multiplexing the data from different parts of the simulators.

The role of the simulators is to some extend interchangeable in the sense that both take information from one simulator and put it into the other, but the ways how this is implemented differs for spiking neuronal networks and robots. As a consequence, we have sticked to the terminology common in the disciplines of the simulators to give users a better intuition. On the other hand, we made the implementation flexible

[2] http://cor-lab.org/robotics-dsl-zoo

to allow alternatives as well. For example, the terminology of Robot Publishers and Robot Subscribers in Fig. 2 is adapted from ROS as these would typically be implemented by asynchronous ROS topics, but the architecture is flexible enough to cope with synchronous communication to the robot as well.

The DSL to specify Transfer Functions is introduced in the following sections. First, we present the abstract syntax of PyTF in Section 3.1, describe supported neuron access patterns in Section 3.2 before we describe the mapping to Python in Section 3.3.

3.1 Main Concepts

The basic idea behind PyTF is that the functional specification of a Transfer Function, that is how the input from a Transfer Function is converted to a robot control signal, can be specified in a regular Python function. Thus, the effect of PyTF is to wrap Python functions into Transfer Functions, map their parameters to parts of either neural network or robot simulation and manage the execution of this function.

The abstract syntax of PyTF to achieve this functionality is depicted in Figure 3. A Transfer Function consists of an underlying Python Function and parameter mapping specifications. Multiple types of parameter mappings exist in order to connect to either neural simulation or robotics simulation. We differentiate between mappings to the neural network (`SpikeMapping`), to the robotics simulation (`RobotMapping`) and to internal variables. These mapping specifications each have subclasses to specify whether the parameter is an input or output to the simulation.

As the parameter mapping specifications are contained in the Transfer Functions, a Transfer Function does not have external references. In particular, the deployment of Transfer Functions could be distributed to multiple nodes in case the Transfer Functions contain computational expensive transmission logic such as processing of large matrices for image processing.

All parameter mappings share an attribute specifying which parameter they belong to and a method to create an adapter component instance. This can be a mapping to a simulation or just a reference to a local or global variable. A reference to the surrounding TF Manager is passed into the mapping specification that contains references to the communication adapters for both neural and robotics simulation, so that the mapping specification itself can be independent of the used simulators.

PyTF has two subtypes of Transfer Functions, `Robot2Neuron` and `Neuron2Robot`, represented by the upper and lower Transfer Function in Fig. 4. The rationale behind this decision is simply to order Transfer Functions in the unlikely case that a control topic is both read from and written to. Thus, `Robot2Neuron` Transfer Functions are executed first. On the other hand, `Neuron2Robot` Transfer Functions often result in sending a message to a particular robot control topic. For this rather common case, the class contains a reference to a publisher so that the Transfer Function may simply use the return value of the function to publish on this topic. Other than that, the type of Transfer Functions has no implications to the allowed parameter mappings. In particular, a `Robot2Neuron` Transfer Function may for example also contain a publisher or a spike sink.

3.2 Neuron Access Patterns

Whereas robot control signals or sensory inputs from the robot can be bundled in arbitrary data structures sent over ROS, the interface of a neuron is determined through its underlying neuron model. In many cases, this interface is limited to a few parameters such as the membrane potential or a history of spikes. As a consequence, a single control signal for a robot is often multiplexed to a multitude of neurons and vice versa sensory inputs such as a camera image are fed into a multitude of neurons. Therefore, Transfer Functions often require to connect multiple neurons at once.

On the other hand, spikes as the usual interface of a neuron in a spiking neural network are discrete events in time whereas control commands for robots usually consist of continuous data sent to the robot in short intervals. Likewise, sensory inputs from the robot that shall be transmitted to the neural network need to be discretized to spikes. To perform these conversions, there are multiple approaches. This includes integration of spikes to obtain continuous data, generating a current or generating spikes either constantly or according to some probability distribution, most notably Poisson distributions.

In PyTF, users can choose between a set of access patterns predefined in the language. Each connection to a particular set of neurons and according to a given access pattern is realized by an object we call device (as this terminology is also partially used in the neural simulators) where the access pattern is called the device type. Depending on whether the device is an input into the network (spike source) or an output (spike sink), different device types apply. Each device can be connected to arbitrary many neurons that can be selected by navigating through the populations of the neural network model.

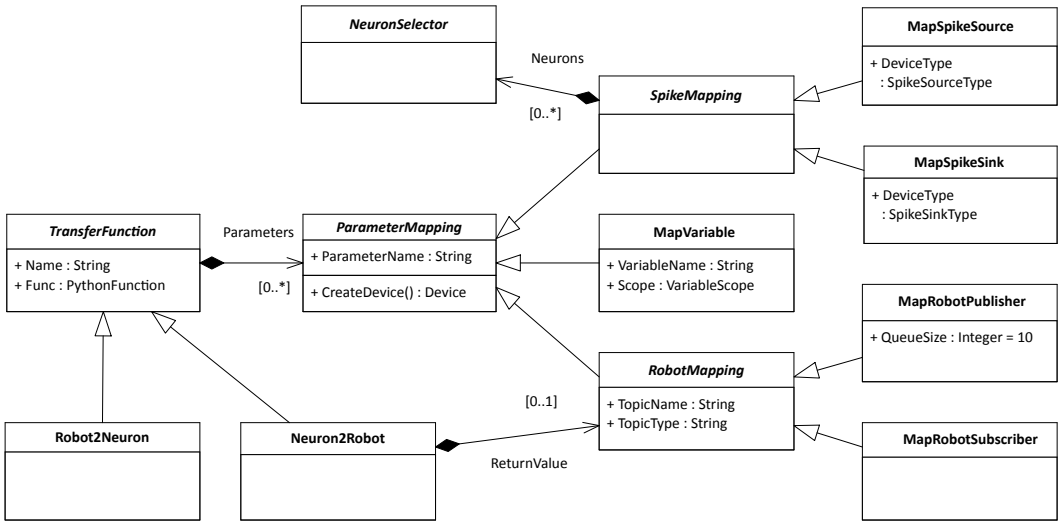

Figure 3 The abstract syntax of PyTF

So far, we support the following spike source device types:

1. **Current Generators:** The current generators for direct current, alternating current or noisy current do not generate spikes but inject currents of the specified type into all of the connected neurons. These devices receive the amplitude of the generated current as inputs. The noisy current generator can also be used to test whether the neural network currently simulated is robust with regard to noise.
2. **Poisson Generator:** A Poisson generator issues spikes according to a Poisson distribution. Here, the inverse of the λ parameter can be set in accordance to sensory inputs. This inverse reflects the rate in which spikes are generated by this device.
3. **Fixed Frequency Generator:** A fixed frequency generator deterministically generates spikes at a given frequency. Here, the frequency is set as a parameter and can be adjusted to sensory inputs. Unlike the other spike generators, this device type is not directly implemented in neural simulators but can be implemented by connecting a current generator with an integrate-and-fire neuron.

This selection is based on the observation that neural simulators, in particular Nest, let simulated neurons communicate through the delivery of spikes and currents. Based on the experiments we performed in the NRP so far, we believe that this list suffices for most applications. However, new device types can be added upon request.

On the contrary, the following spike sinks are supported:

1. **Non-spiking Leaky Integrators:** The concept of leaky integrators is to simply integrate spikes coming from a neuron under observation and add a leak term to it. The rationale behind this is that, in spiking neuronal networks, the membrane potential is highly fragile. Shortly after a spike has been issued, the membrane potential is reset and therefore, it has a high importance whether any measurement is taken before or after a neuron spikes. Therefore, we augment the neural network with an additional leaky integrate-and-fire neuron with an infinite threshold potential (so that it never spikes) and measure the membrane potential of this neuron. The result is much less fragile and therefore appropriate for robot control signals.
2. **Population Rate:** Also a very common pattern is to simply take the average incoming spike rate of a neuron or a set range of neurons (such as a set population). This is, again, stable and can be used for translation into robot control signals.
3. **Spike Recorder:** The simplest thing a spike sink can do is to simply record all spikes issued to a neuron under observation. However, this has two major drawbacks. At first, the communication overhead is increased since all spikes are transmitted between the neural simulation and the Transfer Function but more importantly, the Transfer Function has to interpret this spike train. This allows great flexibility as this approach is very extensible, but it is not suited for the common case.

With the spike sink devices, we tried to reflect the common information encoding of spiking neuronal networks. Again, this list only contains the device types we deem practical for a range of applications and we do not claim that this list to be sufficient for all experiments. This list is subject to change meaning that poorly used

device types may no longer be supported whereas device types frequently asked for may be added. For example, so far we did not include a device capturing the time until the first spike in a simulation loop. As a reason, this value is also highly fragile and thus considered less meaningful at the moment.

The implementation how exactly a given device type is realized is here up to the communication adapter that will ultimately create the appropriate communication objects. For example, the leaky integrator device can be implemented in the Nest simulator by simply inserting a new integrate-and-fire neuron with adequately set parameters and an infinite spiking threshold so that the result is directly available as the membrane potential of the additionally inserted neuron. This is possible since the Nest simulator runs in main memory and therefore allows an arbitrary communication. Other simulators such as SpiNNaker may be based on spike-based communication, only. Here, the implementation of the leaky integrator would rather be to record the spikes and do the integration manually.

Each of these device types has their own additional configuration such as weights and delays in which the spikes are issued to spike generators or from existing neurons into leaky integrators. On the other hand, all devices share the connection specification towards the neural simulator, that we call `NeuronSelector` (cf. Fig. 3). This is a function that, given a model of the neural network, selects the neurons a device should be connected to.

While a single device merely suffices to transmit simple sensory data to the network or to issue command control signals to the robot, the transmission of complex sensory inputs such as camera images requires multiple devices connected to different neurons each. This is the reason that a device mapping can specify not only a single but multiple neuron selectors. In case multiple neuron selectors are present, the framework will create not a single device but one for each neuron selector.

The advantage of these device groups is that they aggregate the values from individual devices to arrays, making this a suitable choice when the according data in the robotics simulator is also available as arrays. This is the case e.g. for camera inputs that can then be for example transmitted to an array of Poisson generators. Furthermore, device groups can be configured comfortably as in such a scenario all devices usually share large proportions of their configuration.

3.3 Mapping to Python

Applying a typical query-and-command programming interface for managing Transfer Functions would presumably result in verbose schematic code (cf. [9]). Thus, we use techniques from the area of Domain-Specific Languages to raise the abstraction level of the target platform by means of an internal DSL, PyTF. With PyTF, we obtain a more concise representation of Transfer Functions. Users can specify Transfer Functions as regular Python functions decorated[3] with their connections to neural and world simulator. The coordination regarding data synchronization and simulation orchestration is hidden in the platform abstractions.

We chose an internal DSL and Python as a host language mainly because Python is popular both among robotics and neuroscience users. Given the research results from Meyerovich et al. [20] that suggest that developers refrain from changing their primary language, we wanted to make the barrier for neuroscientists as low as possible and therefore created a Python API[4]. Furthermore, there is an API for both for the neural simulations and the robotics side. As a consequence, large parts of the framework are implemented in Python and this allows an easy implementation of the DSL as a Python API.

To implement Transfer Functions in PyTF, we have decided for a decorator syntax. A first set of decorators turns a regular Python function into a Transfer Function and a second set of decorators specifies parameter mappings. Everything else, especially including the neuron access patterns and device types is specified as parameters for these decorators.

A consequence of this design is the name of the classes in the abstract syntax. They are adjusted to yield an understandable syntax when used as decorators.

```
1 import hbp_nrp_cle.tf_framework as nrp
2
3 @nrp.Neuron2Robot()
4 def my_transfer_function(t):
5     pass
```

Listing 1 A minimalistic Transfer Function in PyTF

In particular, the classes `Neuron2Robot` and `Robot2Neuron` create a new Transfer Function object with no reference yet to a regular Python function such as sketched in Listing 1. When used as a decorator and applied to a Python function, the underlying Python function of the Transfer Function is set and placehold-

[3] Decorators in Python are syntacically similar to annotations in Java, augmenting methods or classes with additional information.

[4] Application Programming Interface

ers for the parameter mappings are created (Python allows to retrieve the parameter names of a method using the `inspect` module). The function will then be called for each simulation loop, passing the current simulation time as a parameter.

The mapping specification classes `MapSpikeSource`, `MapSpikeSink`, `MapVariable`, `MapRobotPublisher` and `MapRobotSubscriber` then create a parameter mapping specification object that, when called with a Transfer Function, replace the according placeholder with themselves and return the Transfer Function to allow other parameters to be mapped. If no appropriate placeholder exists, an error message is thrown.

```
1 @nrp.MapVariable("var", initial_value=0)
2 @nrp.Neuron2Robot()
3 def my_transfer_function(t, var):
4     pass
```

Listing 2 A minimalist parameter mapping in PyTF

The configuration for mapping specifications is passed as arguments to the decorator representing the parameter mapping. As an example, Listing 2 shows the definition of a parameter mapping to a local variable. Here, the additional configuration of the parameter mapping consists of the initial value for that variable (that is also applied after a reset) and optionally the variables scope, omitted in Listing 2.

The device mappings are most interesting since they allow the most detailed configuration. In particular, they contain a specification to which neurons a device should be connected as a function selecting the neurons for a given model of the neural network. However, as we do not want our users to bother with the details of lambda functions where this is not strictly required, we created a small API to allow them to write such functions as if they were operating on an assumed neural network model directly.

To specify multiple neuron selectors, a list of neuron selectors must be passed into the neural network constructor. In PyTF, we support a mapping operator that construct such lists of neuron selectors based on a lambda function and a concatenation operator to express more complex patterns. These operators make use of the knowledge that neuron selectors must not be nested deeper than in one list (i.e. it is not permitted to specify a list of a list of neuron selectors for a device) and flatten these lists when required.

4 The Neuro-Robotics Platform (NRP)

A round-trip experimentation and validation of neuronal network algorithms controlling a robot in a virtual or real environment requires a solid evaluation platform covering all disciplines. In the scope of the Human Brain Project (HBP)[5], we therefore developed such a platform, the NRP. In the following, we introduce the key components of the NRP simulation backend, describe its architecture and explain the data synchronization between the simulations.

4.1 Overview

The NRP consists of the following key components:

4.1.1 Neural Network Simulator:

To simulate the neural networks, the neuronal simulator NEST [11] is used. This simulator was designed to run on distributed systems utilizing parallel resources. This is especially important given the size of biological spiking neuronal networks such as the human brain with approximately 10^{11} neurons and 10^{15} synapses. However, we are also working on an integration of the neuronal network simulator SpiNNaker [17] which runs on specialized neuromorphic hardware. For users, the choice of the neuronal network is transparent as neuronal networks can be simulated in PyNN [6], an abstraction layer that supports both simulators.

4.1.2 World Simulator:

To simulate the physics of the robots and their environment, the Gazebo simulator [18] is used. For communication with the simulated robot, we use the Robot Operating System (ROS)[26] as a middleware. The platform uses the asynchronous event-based communication through ROS topics. This allows identifying parts of the robot by its topic address and type. Using ROS as middleware also yields the possibility to easily exchange the simulated robot by its physical counterpart.

4.1.3 Closed Loop Engine:

The component connecting both simulators is the Closed Loop Engine (CLE) developed in the scope of the HBP. The CLE orchestrates the brain simulation, world simulation and the data transfer. The data transfer is handled through Transfer Functions (cf. Section 3). As Transfer Functions can take information from a simulation or insert stimuli, a closed loop between the simulations is established.

[5] `http://www.humanbrainproject.eu`

4.2 Architecture of a Simulation

During simulation, the code to run the simulation can be described through components as sketched in the UML Component Diagram of Figure 4.

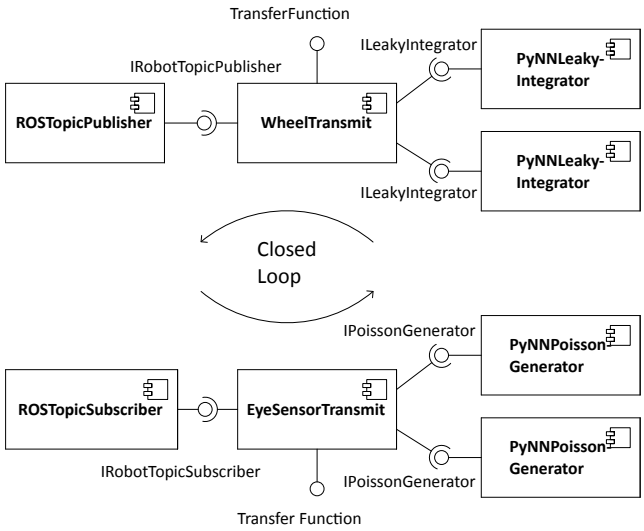

Figure 4 Transfer Function components and their communication adapters in a running simulation of the standard image processing experiment

In the diagram of Figure 4, we assume Transfer Functions for a standard image processing experiment, represented by the components `WheelTransmit` to transmit the voltage from actor neurons to the robot and `EyeSensorTransmit` to transmit camera images to the neural network. They provide an interface to the simulation kernel as a Transfer Function and are thus referred to in the remainder as Transfer Function components. These components are in the middle of the diagram and require interfaces according to their communication needs. For example, the `ILeakyIntegrator` interface specifies a voltmeter to be injected into some neurons so that the Transfer Function component `WheelTransmit` can access their current voltages.

Since these communication needs are hidden behind an interface, the Transfer Function components are independent of simulator implementations on either side. We refer to the components realizing the communication of a Transfer Function component with either simulator as connector components. All these connector components have a configuration such as the robot topics or the neurons that they should be connected to and how. A Transfer Function component may be connected to multiple connector component instances. Each connector component instance is responsible for the connection of a certain group of neurons according to the components configuration.

The connector components on the left side realize the communication with the world simulator. The messages are either directed towards the robot control via `ROSTopicPublisher` or towards the neuronal simulator via `ROSTopicSubscriber`. Internally, these connectors forward the request via ROS to Gazebo.

On the right side of Figure 4, the Transfer Functions access multiple component instances to connect to the neuronal simulator. In Figure 4, this is realized in the connector component instances of `PyNNLeakyIntegrator` and `PyNNPoissonGenerator`. The different kinds of connector components to the neural network have different interfaces since there are multiple access patterns different to just sending or receiving messages. Whereas a leaky integrator collects information from the neural network, the Poisson generator inserts stimuli.

The Transfer Functions contained in Figure 4 established a closed loop between the neural network and the robot. Whereas the `WheelTransmit` collects information from the neural network and publishes information to the physics simulation through the connector components, `EyeSensorTransmit` establishes a connection in the opposite direction.

The entire architecture of a simulation instance such as presented in Figure 4 is specific to the experiment setup. The component types of the connector components such as `ROSTopicPublisher` or `PyNNPoissonGenerator` are fixed as they reflect the methods to access a running simulation. The Transfer Function components `WheelTransmit` and `EyeSensorTransmit` on the other hand are specific to the physiology between the neural network and the robot. In particular, the physiology is subject to change across multiple experiments and to be specified by the user.

4.3 Architecture at design time

To support the dynamic instantiation of such architectures for a particular simulation, we have implemented a framework. The architecture of this framework is presented in this section.

Despite supporting arbitrary simulations, an important design goal is to make the architecture as independent as possible from the simulator implementations. To achieve this, both of the simulators are encapsulated by two different components, one to manage the communication with the simulator (*-Adapter*) and another component to control the simulation (*-Controller*). We establish this separation 1) to separate the concerns of controlling a simulation and accessing its data and 2) because the control of the simulator could be deployed on another machine then the actual data trans-

fer, furthermore, 3) there may be multiple instances realizing the data transfer as opposed to a single instance controlling the simulation.

On the other hand, the choice of an adapter component is of course dependent on the choice of the controller component as both have to refer to the same simulation.

An overview of the architecture is depicted as a UML Component Diagram in Figure 5.

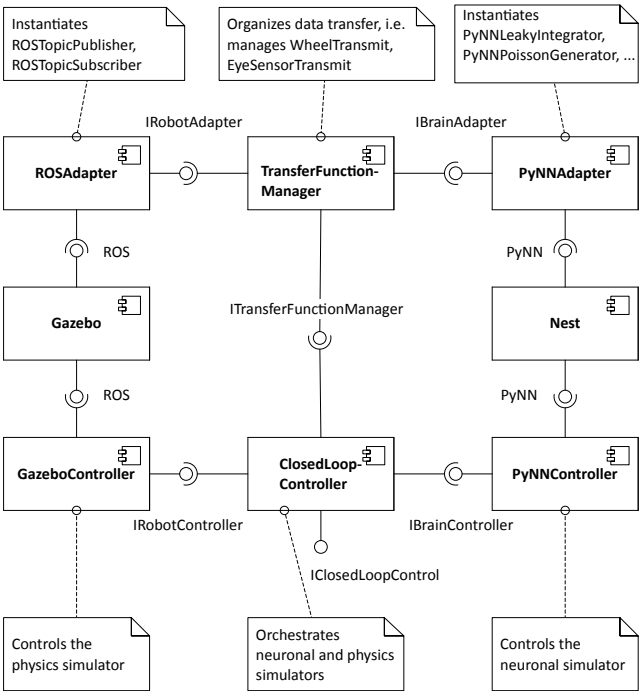

Figure 5 The components of the simulation backend in the NRP

While Figure 4 shows the components in a running simulation instance, Figure 5 depicts the framework architecture at design-time. When initializing a simulation, the components in Figure 5 instantiate the components of Figure 4 according to the experiment setup.

The component accessed from the frontend is the `ClosedLoopController` (CLC). It provides services on a high abstraction level such as initializing, starting, pausing or resetting the simulation and therefore is the control cockpit of the simulation.

The components `NEST` and `Gazebo` represent the neural and world simulators presented in Section 4.1 that are connected to the CLC through the Python interface PyNN or through ROS topics, via respective controller components `PyNNController` and `GazeboController`. Depicting the simulators as components in Figure 5 is not entirely accurate as they are no units of deployment. In particular, both the neuronal simulator `Nest` and also the physics simulator `Gazebo` are complex distributed systems themselves and internally consist of many components. However, we stick to the representation as components for simplicity.

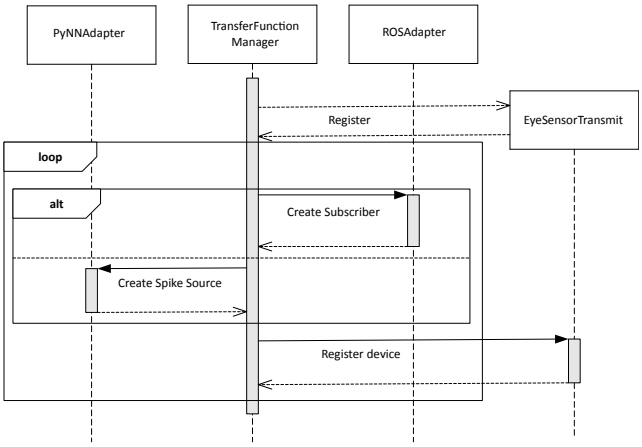

Figure 6 Transfer Function Initialization

The initialization itself is done in the `TransferFunctionManager` (TFM) component as depicted exemplary for the `EyeSensorTransmit` Transfer Function in Figure 6. When initializing the simulation, this component gets the Transfer Function components in the simulation, yet unconnected to connector components. In the simulation sketched in Figure 4, these are `WheelTransmit` and `EyeSensorTransmit`. It then requests connector components from the adapter components `ROSAdapter` and `PyNNAdapter` such that each required interface of each Transfer Function component is connected to an appropriate connector component. After the initialization, the TFM offers services to the CLC to execute the Transfer Functions and retrieve status information about them.

The adapter components `ROSAdapter` and `PyNNAdapter` serve as dependency injectors for the communication demands of a Transfer Function. That is, when the TFM requests a leaky integrator for a given neuron such as in Figure 4, the `PyNNAdapter` will create an instance of a connector component type realizing this interface, in this example the `PyNNLeakyIntegrator`, and connect it to the requested neurons. Likewise, the `ROSAdapter` will create a `ROSTopicPublisher` instance when a publisher to the control topic of the robot is required. The TFM will then connect the returned connector component to the Transfer Function component.

4.4 Data Synchronization

When the simulation is run, the CLC orchestrates the simulation in cycles of a fixed length, currently set to 20ms simulation time.

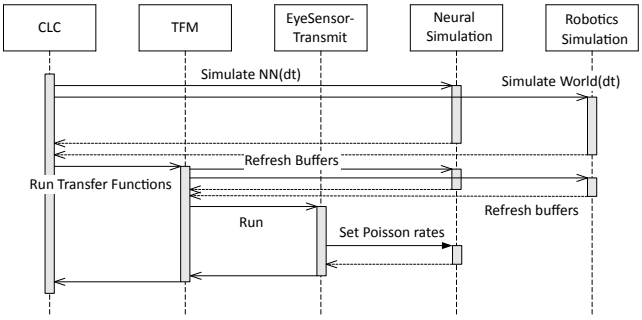

Figure 7 Synchronization of simulations

This cycle is depicted in Figure 7. To save space, we have omitted any controller, adapter or communication components but show the simulations as directly accessed. The CLC first runs both of the simulations in parallel, then it calls the TFM to run the actual data exchange. The TFM holds a list of registered Transfer Functions and thus knows the Transfer Function `EyeSensorTransmit`. But before any Transfer Function is called, the buffers of the adapters are refreshed. This is is necessary for some devices such as leaky integrators to prevent that the devices are updated only once per cycle. Inside the Transfer Function, the access to the device data is very fast as the data is already buffered. The Transfer Function creates its outputs by assigning values to some properties of the used device. This results in a call to the respective simulation, in case of Figure 7, the rates of the Poisson generators are transmitted to the neural network.

As a consequence, it is not possible to access data yielded in the current timestep from the respective other simulation. The only data exchange is done through Transfer Functions, but as they do not run in parallel to the simulations, such data can only be processed in future timesteps. The reason for the sequential execution of Transfer Functions is to avoid race conditions (asynchronously changing parameters of the simulated models causes some simulators to crash), but also to support reproduceability of the experiment results.

5 Case Study: A Braitenberg Vehicle Experiment

In this section, we demonstrate and validate our approach by applying it to a small experiment inspired by the Braitenberg vehicles [5]. We chose this experiment as it is small enough to explain the neural networks involved and to show the code necessary to couple this neural network to a robot. We present the experiment in two versions where the proportions of the neural controller are different. This resembles a typical workflow when transitioning an existing classical robot controller to a neural implementation.

As robot, we use a four-wheeled Husky[6] robot equipped with a camera. This robot is in a virtual room equipped with two screens. The screens are either blue or red and the user can change their color. Eventually, one of the screens is turned red. The robot counterclockwisely turns around until he recognizes the red color and moves towards it.

The implementation can be done relatively easy by using classical image processing methods, for example by iterating through the pixels of the camera image and counting red pixels based on a HSV color model. However, given the results on pattern recognition with neural networks [3], one may want to exchange these classic image processing steps by a neural network in order for a fine granular perception of red colors or to take advantage of neural networks ability to adapt to new situations, i.e. to learn. Conversely, neuroscientists may want to validate their neuro-physiological models in order to check whether they produce valid results.

We therefore take this example as a case study to demonstrate the applicability of our approach. In particular, we migrate the controller for the Husky robot in two steps. In a first step, the identification of red colors is implemented using a standard image processing library, OpenCV. The neural network thus only navigates the robot based on the input stimuli. In a second step, we shift the image processing part into the neural network so that the neural network takes full control over how to detect red pixels.

The neural networks for both steps of this experiment are entirely static. In particular, we did not implement any learning algorithm.

5.1 A Braitenberg Vehicle Controller using Standard Image Processing

As a first step towards a fully neural implementation of a controller for our Husky robot acting as a Braitenberg vehicle, we migrate the implementation of the velocity control into the neural network but use as stimuli the camera images preprocessed using standard image processing. In particular, we use a simplistic spiking neuron network consisting of just 8 neurons getting stimuli

[6] `http://www.clearpathrobotics.com/husky/`

from preprocessed images and letting the robot move towards the red screen. The neural network is adapted from the original network presented by Braitenberg [5]. In the remainder, we refer to this step of the case study experiment simply as standard image processing experiment.

We first present the neural network in Section 5.1.1, present the Transfer Functions to transform spikes from the neural network into robot control signals in Section 5.1.2 and in the opposite direction from the robot camera to stimuli for the neural network in Section 5.1.3.

5.1.1 A Neural Network for Braitenberg Vehicles

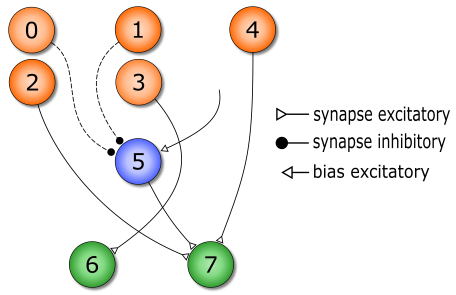

Figure 8 The neural network for the Braitenberg Vehicle experiment with standard image processing

In the neural network for the standard image processing step, depicted in Figure 8, the five neurons in orange (numbers 0 to 4) are bundled in a population that represent the sensors of the network. As an exemplary connection to the Husky robot, these neurons receive the input signal through Poisson generators generating spikes according to a Poisson distribution. The rate of this Poisson distribution depends on how many red pixels have been detected in the camera image. We use Poisson generators since alternative spike sources generating spikes in a fixed frequency are more affected by time resolution. The classification whether a given pixel is red is done through an image processing library function categorizing the pixels according to the HSV color model. This information is propagated through the network so that the membrane potential of the actor neurons 6 and 7 (in green) can be used to control the left and right wheel motors of the robot.

5.1.2 Transmitting Membrane Potentials to Motor Commands

This section describes the information flow from the neural network to the robot. In the Braitenberg experiment, the membrane potential (i.e. the voltage) of the actor neurons encodes the movement of the robot. But as the underlying Husky controller requires to specify movement of the robot in terms of angular and linear progression, the voltages of the actor neurons must be converted by means of arithmetic transformation. In particular, the minimum of both voltages forms the linear progression while their difference results in the angular progression. Furthermore, the resulting movement commands must be scaled to achieve good results.

```
1  import hbp_nrp_cle.tf_framework as nrp
2  from geometry_msgs.msg import Vector3, Twist
3
4  @nrp.MapSpikeSink("left_wheel_neuron",
5     nrp.brain.actors[0], nrp.leaky_integrator_alpha)
6  @nrp.MapSpikeSink("right_wheel_neuron",
7     nrp.brain.actors[1], nrp.leaky_integrator_alpha)
8  @nrp.Neuron2Robot(Topic('/husky/cmd_vel', Twist))
9  def wheel_transmit(t, left_wheel_neuron, right_wheel_neuron):
10     linear = Vector3(20 * min(left_wheel_neuron.voltage,
           right_wheel_neuron.voltage), 0, 0)
11     angular = Vector3(0, 0, 100 * (right_wheel_neuron.voltage -
           left_wheel_neuron.voltage))
12     return Twist(linear=linear, angular=angular)
```

Listing 3 Transfer Function from neurons to the robot in the Python DSL

An implementation of this Transfer Function in our Python DSL is shown in Listing 3. Line 1 simply imports the Transfer Functions framework into the current script. Line 2 imports the ROS Topic types needed for the communication with the robot. Lines 4-12 form the Transfer Function translating the voltage of actor neurons into robot commands.

The function `wheel_transmit` is turned into a Transfer Function from the neural simulator towards the robot simulator by the decorator `@nrp.Neuron2Robot` in line 8. The decorator automatically registers this function at the TFM which will ensure that it is connected to the necessary connector components. Furthermore, the decorator specifies the connector component that will receive the functions return value. In the example, the return value is sent to the robot using the ROS topic `/husky/cmd_vel`. The decorators in lines 4 to 7 specify how the input parameters of the function should be mapped to the neural network. In this case, the parameters are connected to two single neurons of the *actors* population through a leaky integration algorithm. The first parameter t of a Transfer Function is always the current simulation time and cannot be remapped, whereas all other parameters must be mapped to either robot topics or neurons.

The body of the original Python function in lines 10-12 is not affected by the Python DSL and is allowed to contain arbitrary Python code. In this Transfer Function, we manually construct the Control messages to control the Husky's velocity.

Additional details of the device connection to the neural network such as the specification of weights or

delays are not required in this case as the default values suffice.

5.1.3 Transmitting Processed Images to the Neural Network

We now describe the opposite direction, i.e. the processing of camera images to stimuli for the neural network. A camera image is taken from the world simulator, red colors are detected by a call to OpenCV and the results are used as stimuli for the neural network (cf. Section 5.1).

```
1  import cv2
2
3  @nrp.MapRobotSubscriber("camera", Topic('/husky/camera', sensor_msgs.
       msg.Image))
4  @nrp.MapSpikeSource("red_left_eye",
5     nrp.brain.sensors[0, 2], nrp.poisson)
6  @nrp.MapSpikeSource("red_right_eye",
7     nrp.brain.sensors[1, 3], nrp.poisson)
8  @nrp.MapSpikeSource("green_blue_eye",
9     nrp.brain.sensors[4], nrp.poisson)
10 @nrp.Robot2Neuron()
11 def eye_sensor_transmit(t, camera, red_left_eye, red_right_eye,
       green_blue_eye):
12    cv_image = bridge.imgmsg_to_cv2(camera.value, "rgb8")
13    image_results = process(cv_image)
14
15    red_left_eye.rate = 1000.0 * image_results.leftred
16    red_right_eye.rate = 1000.0 * image_results.rightred
17    green_blue_eye.rate = 1000.0 * image_results.greenblue
```

Listing 4 Transfer Function from a camera image to neuron spikes

The implementation of this Transfer Function is depicted in Listing 4 where we omitted the import statements. Line 3 is responsible to map the *camera* parameter to a subscriber to the camera topic of the robot. In lines 4-9, the parameters `red_left_eye`, `red_right_eye` and `green_blue_eye` are mapped to Poisson generators for the respective neurons. The decorator `@nrp.Robot2Neuron` in line 10 marks the function as a Transfer Function from the world simulation to the neural network.

The body of the original Python function simply then processes the camera image using standard image processing libraries such as in particular OpenCV in line 11. The results from this process are then used as inputs for the Poisson generators in lines 13-15.

5.2 A Braitenberg Vehicle Controller using Neural Image Processing

Striving to perform as many tasks as possible through neural networks, the standard image processing version of the experiment can be extended by shifting the detection of red colors to the neural network. In the standard image processing setup, the neural network can only react on the processed images which limits the applicability of any learning based on new incoming images to the preprocessing results. However, one would rather want that the neural network can learn based on the entire image, e.g. to enhance pattern recognition.

While in the standard image processing version of the experiment, only the result of the image processing is transmitted to the neural network, in the extended step we transmit the entire camera image from the robot to the neural network. Only the rescaling of the image to a resolution appropriate for the neural network is left to the transfer function.

As a consequence, subsequent steps to improve the capabilities of the neural network in terms of pattern matching can be implemented without having to change the Transfer Function as the Transfer Function only describes the interface from the classical controller (i.e. the camera in the robot) to the neural network.

5.2.1 A Neural Network Extension for Image Processing

Thus, compared to the standard image processing version of the experiment, the neural image processing version yields the requirement to extract stimuli from an array such as a camera image. These stimuli are then to be transmitted to a whole range of neurons.

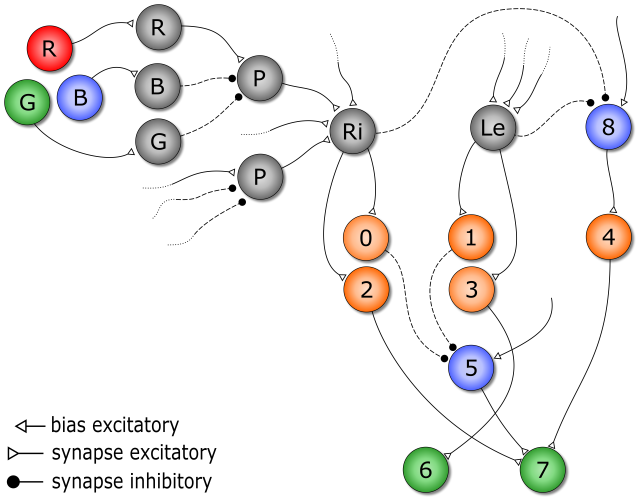

Figure 9 Sketch of the neural network for a Braitenberg Vehicle experiment with neural image processing

The example neural network for recognizing red colors is sketched in Figure 9. For a 40x30 pixel image, it contains approximately $5,000$ neurons. Each pixel is processed by a neuron P. The pixels of a half image are all connected to the neuron populations Ri or Le that represent how much red color can be seen on the right or left half image, respectively. Each pixel neuron P is connected to three Poisson generators that spike

according to the red, green and blue color channels of the corresponding pixel.

While the neural network in this extended case is much larger than in the standard image processing version, it is still to be considered a very small neural network. This is particularly because the Husky robot that the experiment is using only contains two degrees of freedom.

5.2.2 Transmitting Raw Images to the Neural Network

The connect this neural network to the robot controller, we need to insert stimuli for the entire image that needs to be transmitted. We do this by transmitting each RGB color value separately to the neural network as this network contains a node for each color channel of each pixel through a Poisson generator (cf. Fig. 9). However, for an image resolution of just 30×40 pixels, this amounts to a connection of $30 \times 40 \times 3 = 3600$ Poisson generators. As Python has a limitation to 256 positional parameters, it is not possible to create a Transfer Function with 3600 parameters and it would not be convenient, either. Thus, we use device groups.

```
1  @nrp.MapRobotSubscriber("camera", Topic('/husky/camera', sensor_msgs.
       msg.Image))
2  @nrp.MapSpikeSource("red_generators",
3     nrp.map_neurons(1200, lambda i: nrp.brain.sensors[i]),
4     nrp.poisson, weights=0.0075, target='excitatory')
5  @nrp.MapSpikeSource("green_generators",
6     nrp.map_neurons(1200, lambda i: nrp.brain.sensors[i]),
7     nrp.poisson, weights=0.00375, target='inhibitory')
8  @nrp.MapSpikeSource("blue_generators",
9     nrp.map_neurons(1200, lambda i: nrp.brain.sensors[i]),
10    nrp.poisson, weights=0.00375, target='inhibitory')
11 @nrp.Robot2Neuron()
12 def eye_sensor_transmit(t, camera, red_generators, green_generators,
       blue_generators):
13    image_results = tf_lib.get_color_values(image=camera.value, width
          =40, height=30)
14
15    red_generators.rate = 250.0 * image_results.red
16    green_generators.rate = 250.0 * image_results.green
17    blue_generators.rate = 250.0 * image_results.blue
```

Listing 5 Transfer Function from a camera image to Poisson rates for each pixel

The code for the Transfer Function to transmit the images from the camera to the neural network using device groups is shown in Listing 5. Similar to Listing 4 from the standard image processing experiment, it contains a Python function in lines 12-17 that is marked as a Transfer Function using the `@nrp.Robot2Neuron` decorator in line 11.

The device group specification is contained in lines 2-10. The `map_neurons` function is used to specify that the parameters should be used to multiple neurons using a device group. This function takes an index set as parameter and a lambda function how an index is tied to a neuron. Lines 3, 6 and 9 specify that three groups of 1200 Poisson generators should be created, each Poisson generator connected to exactly one neuron that has the same index inside the *sensors* population. Whereas this connection is excitatory for the red values of a pixel, the synapses for the Poisson generators responsible for green and blue values are inhibitory.

In the function body of Listing 5, the library call in line 13 splits the image into three arrays with the pixel values according to the given channels. The arrays are implemented as **NumPy**[7] arrays that support arithmetic operations like the scaling of the resulting vectors in lines 15-17. The rescaled vectors are then assigned as rates to the Poisson generators. The device group internally reconfigures the rate of each Poisson generator device in this group.

5.3 Simulation of the Braitenberg Vehicle Experiment in the NRP

To validate that our neural controller produces the correct outputs, we run the Husky robot in a simulated environment, i.e. a realistic virtual room equipped with two screens that may be turned red by the user during the simulation. The simulation uses the Transfer Functions introduced in Sections 5.1 and 5.2. In both versions, the Husky successfully finds the red color and moves towards it. Figure 10 shows a screenshot of the simulation of the standard image processing version of the experiment and a video is publicly available online[8].

The NRP offers some tools for experimenters to validate their experiment. In Figure 10, one can see two tool windows showing a plot of the spike train for the neurons and the plot of the joint velocities. The purpose of these tools is to enable experimenters to retrace what is currently happening during a simulation. In the moment the screenshot was taken, the robot has already had turned towards the red screen and moved towards it until the screen got out of sight. It then turned again until it found the other screen.

With the spike train, we can see that the four neurons with indices 0-3 connected to the Poisson generators to encode the image spike exactly when a red screen is in the robots area of sight. Neuron 4 spikes all the time since there is a considerably large proportion of the image that is not red. When no red color is detected, neuron 5 creates a sparse spike train. This combination is then added to neurons 6 and 7 that forward their information to the respective transfer function which translates these spike trains in motor commands, thus allowing the robot to move.

[7] `http://www.numpy.org/`

[8] `https://www.youtube.com/watch?v=osmkKQb5pTc`

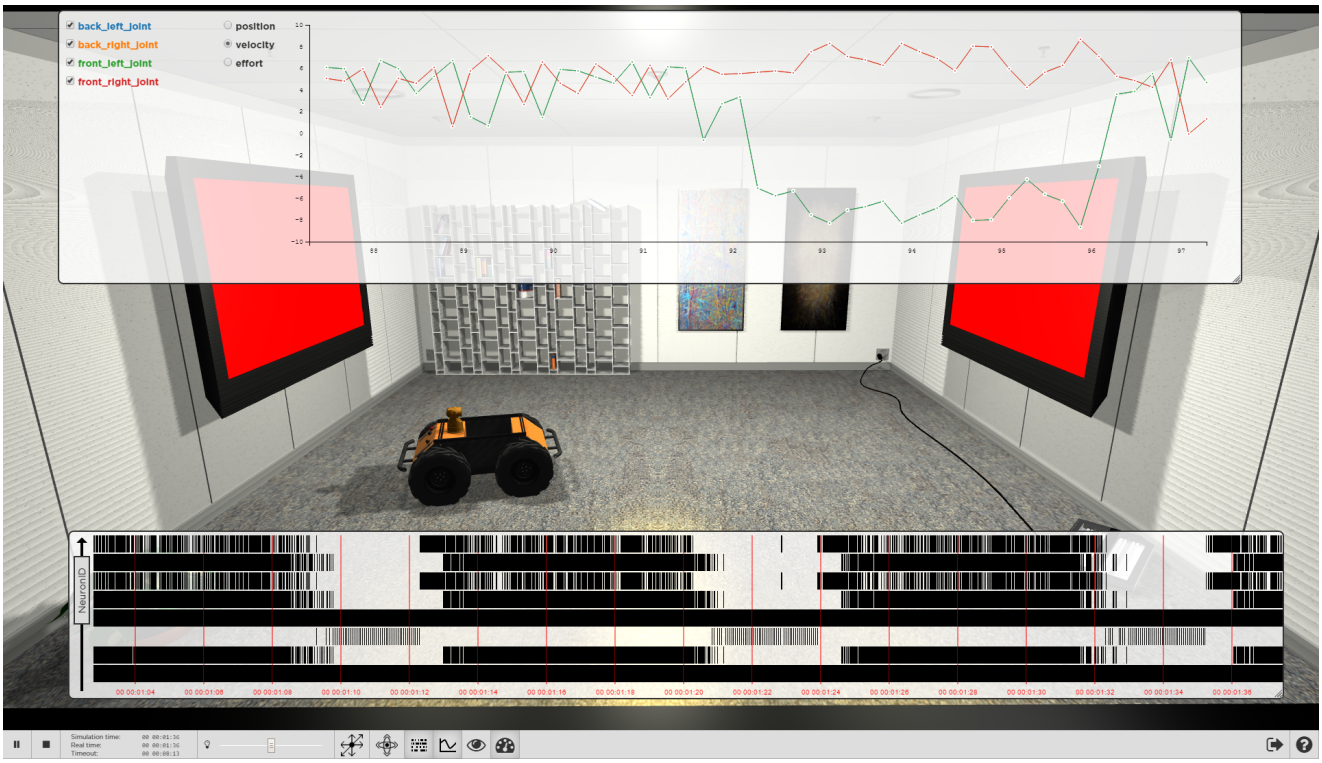

Figure 10 The Braitenberg vehicle experiment using standard image processing simulated in the NRP platform

In the joint plot, this is reflected by graphs plotted for the velocity of the front wheel joints where the plots shows positive velocities for both front wheel joints when the robot moves towards the screen and opposite velocities of different signs when the robots turns.

If a Transfer Functions turned out to produce suboptimal results, the platform also allows to exchange the Transfer Functions during a running simulation. For this, we provide a simple editor with syntax-highlighting, shown in Figure 11. The editor on the left half of this screenshot lists all Transfer Functions currently loaded in the simulation with their specification in PyTF. When a Transfer Function is updated, the old Transfer Function is discarded, releasing the devices where possible and adding the new Transfer Function on the fly.

6 Conclusion and Future Work

In this paper, we have presented an approach to bridge the semantic gap between spiking neural networks and simulated robots. Coupled simulations can be supported with a experiment-agnostic framework architecture that eases the specification of the experiments. This architecture is implemented in a web-based integrated simulation platform that makes it easy for neuroscientists to run experiments validating models of a connection between neural networks and actuators, but also gives roboticists a tool to develop robotics controllers tightly coupled to a spiking neural network. We have presented a textual DSL in Python targeted for neuroscience users with a good knowledge of Python to specify the connection between spiking neural networks and robots for a particular experiment.

However, not all users may have the necessary programming skills and know Python as good. Thus, a formal language equipped with a graphical editor is under development. With such an editor, we hope to make the coupled simulation of spiking neural networks and robots accessible for a wider audience. Furthermore, we want to develop analyses and constraint checks to ensure that Transfer Functions reference valid input and output of the Brain and Body. As a benefit, we hope to detect design flaws in simulations before we need to allocate sparse resources such as neuromorphic computing platforms.

Acknowledgements The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 604102 (Human Brain Project).

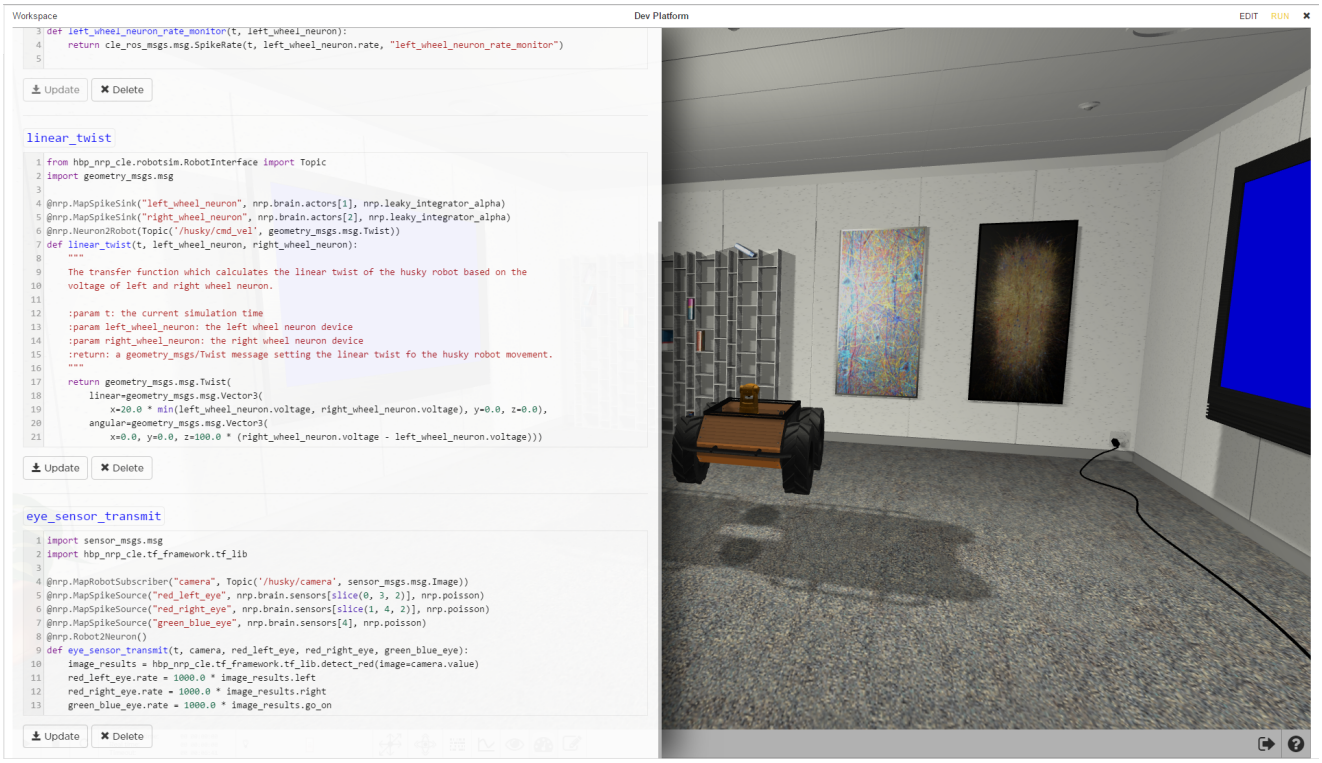

Figure 11 Editing Transfer Functions during a simulation

References

1. Atkinson, C., Gerbig, R., Markert, K., Zrianina, M., Egurnov, A., Kajzar, F.: Towards a Deep, Domain Specific Modeling Framework for Robot Applications. In: U. Assmann, G. Wagner (eds.) Proceedings of the 1st International Workshop on Model-Driven Robot Software Engineering (MORSE), no. 1319 in CEUR Workshop Proceedings, pp. 1–12. Aachen (2014). URL `http://ceur-ws.org/Vol-1319/`
2. Bihlmaier, A., Wörn, H.: Robot Unit Testing. In: Simulation, Modeling, and Programming for Autonomous Robots, pp. 255–266. Springer (2014)
3. Bishop, C.M.: Neural networks for pattern recognition. Oxford university press (1995)
4. Bordignon, M., Schultz, U.P., Stoy, K.: Model-based Kinematics Generation for Modular Mechatronic Toolkits. SIGPLAN Not. **46**(2), 157–166 (2010). DOI 10.1145/1942788.1868318. URL `http://doi.acm.org/10.1145/1942788.1868318`
5. Braitenberg, V.: Vehicles: Experiments in synthetic psychology. MIT press (1986)
6. Davison, A.P., Brüderle, D., Eppler, J.M., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., Yger, P.: Pynn: a common interface for neuronal network simulators. Frontiers in Neuroinformatics **2**(11) (2009). DOI 10.3389/neuro.11.011.2008. URL `http://www.frontiersin.org/neuroinformatics/10.3389/neuro.11.011.2008/abstract`
7. Davison, A.P., Hines, M.L., Muller, E.: Trends in programming languages for neuroscience simulations. Frontiers in neuroscience **3**(3), 374 (2009)
8. Di Ruscio, D., Malavolta, I., Pelliccione, P.: A family of domain-specific languages for specifying civilian missions of multi-robot systems. In: First Workshop on Model-Driven Robot Software Engineering-MORSE (2014)
9. Fowler, M.: Domain-specific languages. Pearson Education (2010)
10. Frigerio, M., Buchli, J., Caldwell, D.G.: A Domain Specific Language for kinematic models and fast implementations of robot dynamics algorithms (2013)
11. Gewaltig, M.O., Diesmann, M.: Nest (neural simulation tool). Scholarpedia **2**(4), 1430 (2007)
12. Gleeson, P., Crook, S., Cannon, R.C., Hines, M.L., Billings, G.O., Farinella, M., Morse, T.M., Davison, A.P., Ray, S., Bhalla, U.S., Barnes, S.R., Dimitrova, Y.D., Silver, R.A.: NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail. PLoS Comput Biol **6**(6), e1000,815 (2010). DOI 10.1371/journal.pcbi.1000815. URL `http://dx.doi.org/10.1371%2Fjournal.pcbi.1000815`
13. Hagras, H., Pounds-Cornish, A., Colley, M., Callaghan, V., Clarke, G.: Evolving spiking neural network controllers for autonomous robots. In: Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on, vol. 5, pp. 4620–4626 Vol.5 (2004). DOI 10.1109/ROBOT.2004.1302446
14. Hines, M.: A program for simulation of nerve equations with branching geometries. International journal of bio-medical computing **24**(1), 55–68 (1989)
15. Hinkel, G., Groenda, H., Vannucci, L., Denninger, O., Cauli, N., Ulbrich, S.: A Domain-Specific Language (DSL) for Integrating Neuronal Networks in Robot Control. In: 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering (2015)
16. Kernighan, B.W., Pike, R.: The Unix programming environment, vol. 270. Prentice-Hall Englewood Cliffs, NJ (1984)
17. Khan, M.M., Lester, D.R., Plana, L.A., Rast, A., Jin, X., Painkras, E., Furber, S.B.: SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor. In: Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on, pp. 2849–2856. IEEE (2008)

18. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, vol. 3, pp. 2149–2154. IEEE (2004)
19. Lier, F., Lütkebohle, I., Wachsmuth, S.: Towards Automated Execution and Evaluation of Simulated Prototype HRI Experiments. In: Proceedings of the 2014 ACM/IEEE International Conference on Human-robot Interaction, HRI '14, pp. 230–231. ACM, New York, NY, USA (2014). DOI 10.1145/2559636.2559841. URL http://doi.acm.org/10.1145/2559636.2559841
20. Meyerovich, L.A., Rabkin, A.S.: Empirical analysis of programming language adoption. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, pp. 1–18. ACM (2013)
21. Moghadam, M., Christensen, D.J., Brandt, D., Schultz, U.P.: Towards Python-based Domain-specific Languages for Self-reconfigurable Modular Robotics Research (2013)
22. Nichols, E., McDaid, L., Siddique, N.: Biologically inspired snn for robot control. Cybernetics, IEEE Transactions on **43**(1), 115–128 (2013)
23. Nordmann, A., Hochgeschwender, N., Wrede, S.: A survey on domain-specific languages in robotics. In: Simulation, Modeling, and Programming for Autonomous Robots, pp. 195–206. Springer (2014)
24. Plotnikov, D., Blundell, I., Ippen, T., Eppler, J.M., Morrison, A., Rumpe, B.: NESTML: a modeling language for spiking neurons. In: Modellierung (2016). Accepted, to appear
25. Pomerleau, D.A.: Neural network perception for mobile robot guidance. Ph.D. thesis, Carnegie Mellon University, Pittsburgh (1992)
26. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source Robot Operating System. In: ICRA workshop on open source software, vol. 3, p. 5 (2009)
27. Raikov, I., Cannon, R., Clewley, R., Cornelis, H., Davison, A., De Schutter, E., Djurfeldt, M., Gleeson, P., Gorchetchnikov, A., Plesser, H., Hill, S., Hines, M., Kriener, B., Le Franc, Y., Lo, C.C., Morrison, A., Muller, E., Ray, S., Schwabe, L., Szatmary, B.: NineML: the network interchange for neuroscience modeling language. BMC Neuroscience **12**(Suppl 1), P330 (2011). DOI 10.1186/1471-2202-12-S1-P330. URL http://www.biomedcentral.com/1471-2202/12/S1/P330
28. Roennau, A., Heppner, G., Nowicki, M., Dillmann, R.: LAURON V: A versatile six-legged walking robot with advanced maneuverability. In: Advanced Intelligent Mechatronics (AIM), 2014 IEEE/ASME International Conference on, pp. 82–87. IEEE (2014)
29. Schlegel, C., Haßler, T., Lotz, A., Steck, A.: Robotic software systems: From code-driven to model-driven designs. In: Advanced Robotics, 2009. ICAR 2009. International Conference on, pp. 1–8. IEEE (2009)
30. Strey, A.: EpsiloNN - A specification language for the efficient parallel simulation of neural networks. In: J. Mira, R. Moreno-Díaz, J. Cabestany (eds.) Biological and Artificial Computation: From Neuroscience to Technology, *Lecture Notes in Computer Science*, vol. 1240, pp. 714–722. Springer Berlin Heidelberg (1997). DOI 10.1007/BFb0032530
31. Vannucci, L., Ambrosano, A., Cauli, N., Albanese, U., Falotico, E., Ulbrich, S., Pfotzer, L., Hinkel, G., Denninger, O., Peppicelli, D., Guyot, L., Von Arnim, A., Deser, S., Maier, P., Dillman, R., Klinker, G., Levi, P., Knoll, A., Gewaltig, M.O., Laschi, C.: A visual tracking model implemented on the iCub robot as a use case for a novel neurorobotic toolkit integrating brain and physics simulation. In: Humanoid Robots (Humanoids), 2015 IEEE-RAS 15th International Conference on, pp. 1179–1184 (2015). DOI 10.1109/HUMANOIDS.2015.7363512