Transfer Functions Architecture

State: Draft

Last Edited: 01.09.2014 by Georg Hinkel

1. Purpose

The transfer functions describe how spikes from the neuronal simulation are to be translated into commands for a robot and how the sensor data from the robot should be translated back to spikes. This translation is specified by the users of the Neurorobotics project, which are neuroscientists. Thus, a framework is provided to make the specification of these transfer functions as easy as possible for the users and abstracting away as much as possible technical details. This framework is referred to as Transfer Functions framework (TF framework). The architecture of this framework is the subject of this document.

2. Functional Requirements

The neurons of the Nest brain simulator occasionally send spikes, messages with no data content except for the neuron that sent it and a timestamp. Furthermore, the Nest brain simulator is clocked, i.e. spikes can only be sent in the speed ratio of the brain simulation. However, a neuron does not have to spike in every timeframe but can also decide not to spike.

The transfer functions transfer this spike data into a continuous data stream for the robot, i.e. for every robot topic, a message is sent for every timestep even though no neuron that is needed by the corresponding transfer function has spiked. As a consequence, all transfer functions must be evaluated in every timestep.

A transfer function may access spikes of arbitrary many neurons and the spikes of a neuron may be used by arbitrary many transfer functions. Consecutive neurons can be grouped together in arrays to make the specification of transfer functions easier, but do not have to. A transfer function may require spikes of both grouped and single neurons.

The transfer functions create (or read, depending on the direction) multiple typed robot topics. The user (i.e. the neuroscientist) is responsible to ensure that the data sent to particular robot topics matches the type of the topic. This may be checked additionally by a compiler, but this feature is not mandatory and is depending on the specification language of the transfer functions.

A common requirement for all transfer functions is that they may hold a state. Furthermore, the state should be sharable among multiple transfer functions to allow maximal flexibility of the framework. The transfer functions should have a specified execution order to get an intuitive understanding of side effects.

Specifically for the transfer function from the robot sensors to the neuronal simulator, the TF framework should establish the least possible amount of assumptions on the spike generation schemes, i.e. there should be no restriction to the patterns in which spikes are sent to the neurons. However, neuroscientists should be able to specify common spike generation schemes like frequencies, i.e. to send spikes to a particular neuron in a frequency that is possibly higher than the incoming rate of sensor data.

3. Technical Requirements

In the neuroscience community, the programming language Python has gained a good popularity. Thus, it is important to allow neuroscientists to specify TFs in Python. Further considerations to use C++ were abandoned at the moment, as the simulator is also controlled using Python. Furthermore,

we can use CPython to convert Python code to C if want to reduce the performance impact of using Python.

4. Architecture Overview

The Transfer Functions are a part of the Closed Loop Engine (CLE) and are situated between the neuronal simulator (“the brain”) and Gazebo as our world simulation engine (WSE). Thus, they have an interface to the neuronal simulator at the one hand and an interface to Gazebo at the other hand. However, to ensure that we can test the transfer functions separately, the communication both to the neuronal simulator and to Gazebo is hidden behind interfaces and the functionality is provided to the TF framework by adapters.

Figure 1 shows the architecture of the CLE. The transfer functions are the important component in the middle that transfers data between the two simulators, being the neuronal simulator at the one side and the world simulation engine at the other. Both Simulators are accessed through adapters that represent their functionality and can be replaced with appropriate mocks when necessary. The transfer functions are controlled by the Closed Loop Controller which is responsible to orchestrate the transfer functions with the simulations. The ROS adapter implements the Robot Communication Adapter interface via publishing and subscribing ROS topics, whereas the PyNN Brain Adapter fulfills its interface by delegating the calls to Nest via the PyNN interface. This common interface allows us to use either Nest, SpiNNaker, Neuron or a mock.

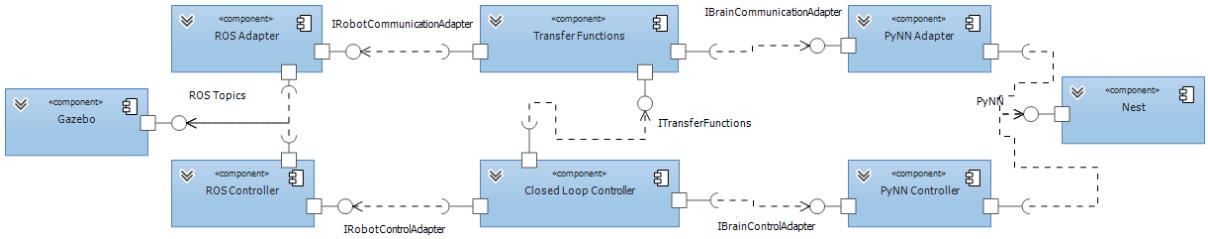

Figure 1: Architecture of the Closed Loop Engine

The interfaces relevant for the above architecture of the CLE are shown in Figure 2. The transfer functions provide three methods that initialize them and call the transfer functions in either direction.

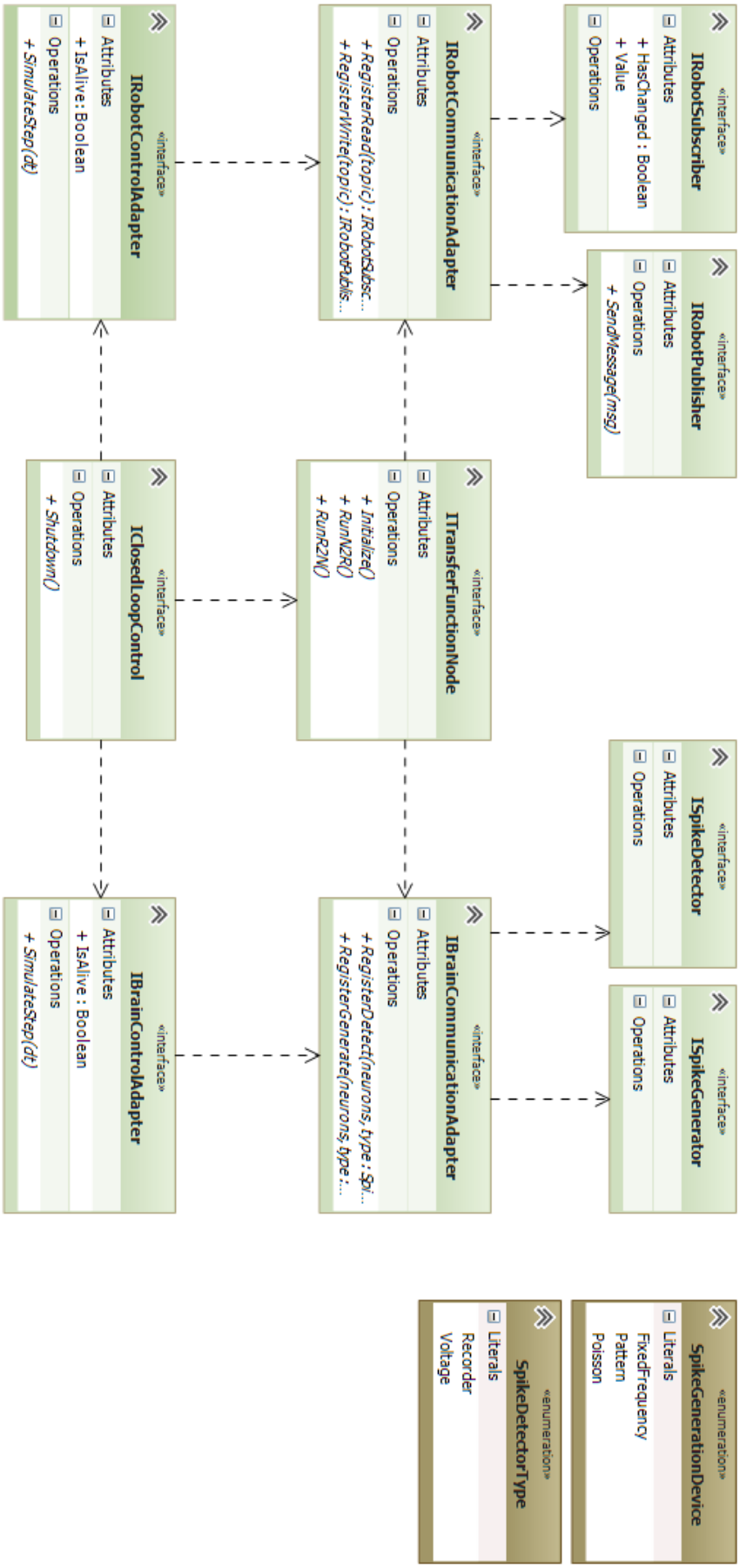

Figure 2: Interfaces of the Closed Loop Engine

On the other hand, the transfer functions will rely on the interfaces for the robot communication and the communication to the neuronal simulator. Both communication adapters provide an interface to initialize them register that data will eventually be retrieved or sent. For this purpose, the transfer functions call the appropriate adapter to create a communication object that can be used to send or receive data. Appropriate interfaces for such communication objects exist for both robot and neuronal simulator side.

On the side of the neuronal simulator, the interface is dependent on the used device type, whereas the interface for the robot adapter is fixed. This is due to the fact that all robot properties are exposed through topics, whereas the data from the neuronal simulation are most efficiently fetched using devices. That is, the neuronal simulation is instrumented with devices that run within the neuronal simulator and get executed by the neuronal simulator for every timestep of the neuronal simulator. These devices can either record the spikes in a certain way or are allowed to issue spikes. Typically, these devices have a low configuration overhead, a frequency-based spike generator for instance only needs the frequency in which to issue spikes.

Thus, the architecture for the TF framework foresees to communicate with the neuronal simulator only through such devices in order to minimize the communication overhead and maximize performance.

An important requirement for the TF framework is that only those spikes should be transferred to the TF nodes that the TF node needs, for the robot topics likewise. To lower the amount of specification that we need from the user, this information should be extracted from the specification of the transfer functions.

Thus, a specification of a transfer function in accordance to our architecture consists of three parts, where we distinguish between two different kinds of transfer functions. The first kind, Neuron2Robot, transfers spikes from the neuronal simulator to topics for the WSE. The other kind, Robot2Neuron transfers data in the other way round.

Each of these transfer function specifies the data sources from which it receives data. For Neuron2Robot, this is neuronal simulator devices (a transfer function may receive input from multiple devices), whereas for Robot2Neuron it is robot topics, where again multiple topics are allowed.

The second part of the transfer function is the specification of the result, i.e. the specification to which output the data should be sent. For a Neuron2Robot transfer function, this is a robot topic. Additionally, the transfer function may specify that it needs to send data to other robot topics as well. This distinction is necessary as we wish to optimize the more common case that a transfer function results in a message for exactly one robot topic. In this case, the return value of the actual function (the third part of a transfer function) is sent to the robot through the framework. To other topics can be written through framework calls. The same holds for the Robot2Neuron transfer functions.

The third part of a transfer function is the actual function, i.e. the specification how the robot topics and device configuration (either for sending or receiving data) is connected. This specification can be done through arbitrary Python code. The data source is fed into the function as a formal parameter, whereas the return value is taken and sent to the target sink, i.e. the targeted robot topic or device. Further communication is possible through communication objects directly.

5. Transfer Function Framework Architecture

5.1. Initialization

The Initialization of a TF node is straight forward. In the initialization, the TF node is supposed to initialize the adapters for both robot and the neuronal simulator. The initialization also includes the

setup and registration of the transfer functions, i.e. to connect the transfer functions with their necessary communication objects.

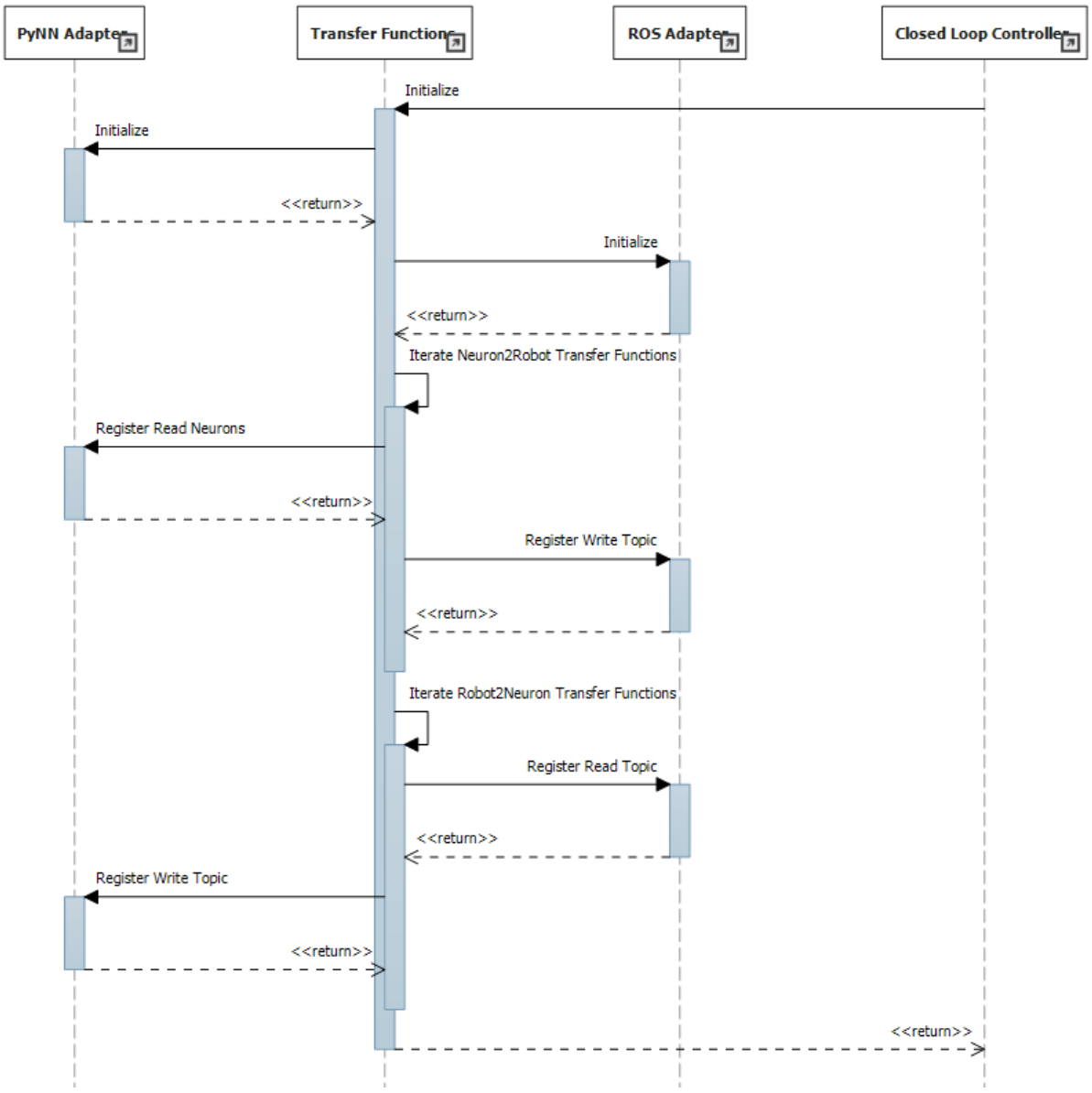

Figure 3: Initialization of a TF node

The sequence of the initialization is shown in Figure 3. The initialization of the TF node is triggered from the closed loop controller, either through in-process communication or remotely. The TF node then makes sure that the dependent adapters are initialized and creates the communication objects necessary for the transfer functions.

5.2. Running the transfer functions

Due to the requirement to run transfer functions even in the case that no neuron accessed by a transfer function spiked, we need the TF framework to run clocked synchronized with the Nest brain simulation, but with a lower time resolution than the neuronal simulator. Responsible for this synchronization is the closed loop controller, which calls the transfer functions.

As the spikes from a neuron can be accessed by multiple transfer functions, the current state of a device from the neuronal simulator is cached in the TF node. The same holds for robot topics, where incoming messages are buffered to be processed by transfer functions separately.

The sequence diagram for a TF node implementing transfer functions in both directions is shown in Figure 4. At any time, the TF node may receive incoming data from the robot via callbacks of subscribed ROS topics. These calls simply update the cached current status of the robot sensors.

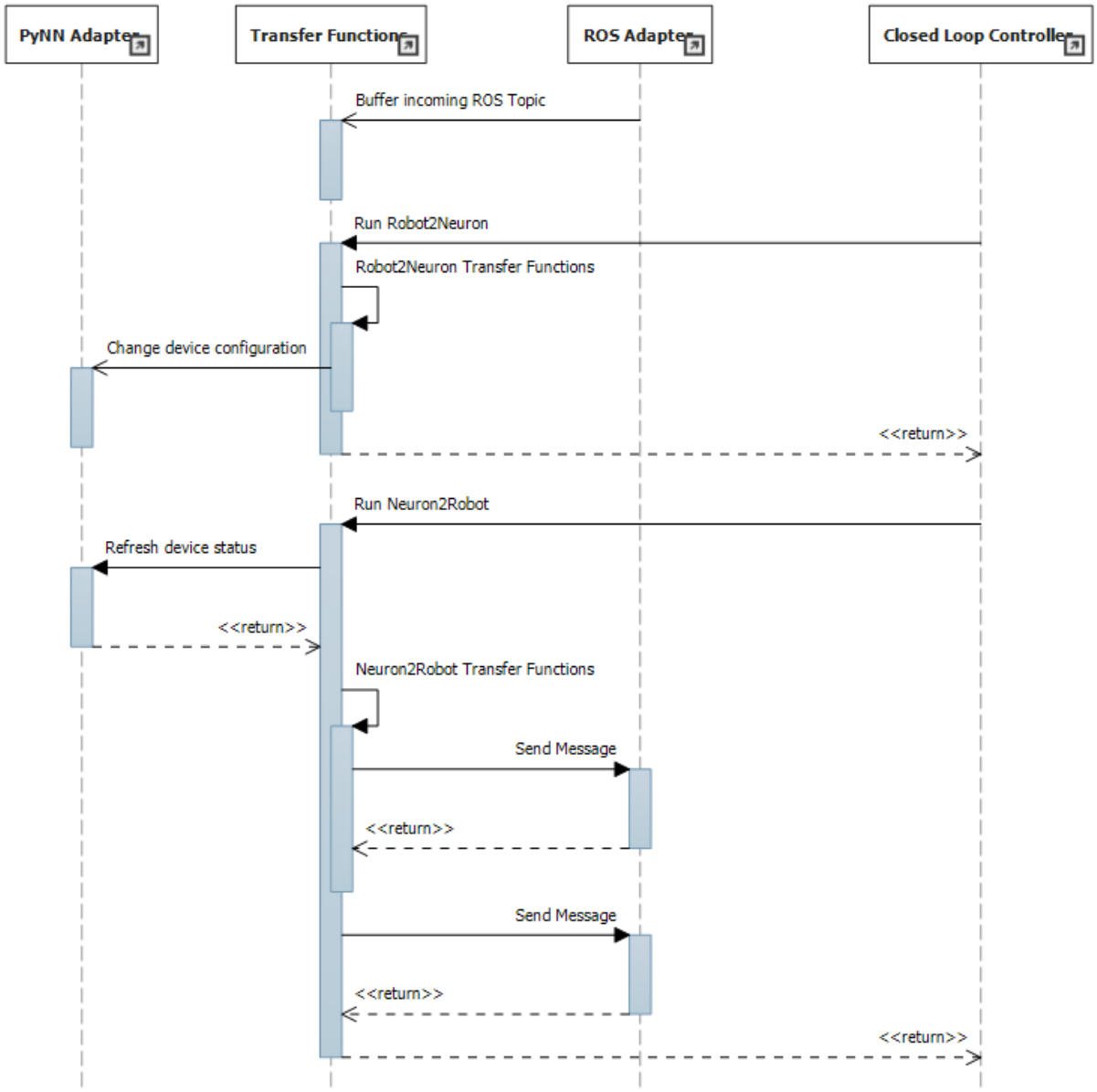

Figure 4: Iteration of a TF node

Eventually, the closed loop controller calls the TF node to either run all transfer functions transferring data from the neuronal simulator to the WSE or vice versa. These calls may also be done in parallel. Thus, the methods from the TF node to run either transfer functions must be thread-safe.

Eventually, the closed loop controller calls the TF node to call all transfer functions that transfer robot messages to spikes for the neuronal simulator. In this case, the TF node runs all Robot2Neuron transfer functions. Within the body of these functions, the device status may be updated. This device is given to the transfer function also as a formal parameter. As data source, the cached values from the buffered robot messages are used.

Conversely, the closed loop adapter may also call the TF node to run the transfer functions from the neuronal simulator to robot messages. For this, the TF node will call the brain adapter to refresh all cached device states. Then, the Neuron2Robot transfer functions are called. They take the current

cached device status as input and return some value or the default value None. The return value of these transfer functions is sent to the predefined robot message by the TF framework. However, Neuron2Robot transfer functions are also allowed to send messages on their own within their function body.

Concluding, all transfer functions are evaluated at every simulation loop of the Closed Loop Engine, but possibly on a lower time resolution than the neuronal simulator. They operate on buffered data from the robot sensors, but also get a flag indicating whether these values have changed since the last loop. Transfer functions are evaluated in order of their appearance in the specification so that side effects happen in an intuitive manner. Sophisticated spike generation patterns are subject of further helper classes.

The patterns for the spike generators are supported by custom spike generator classes, which are implemented separately and called within the transfer function. This design makes it easier to import other spike generators as neuroscience discovers new spike generation patterns.

6. Specification Prototypes

This section introduces prototypes for the user specification of transfer functions. A prototype for a Python implementation is provided as well as a prototype for C++. The last subsection shows the suggested API for both prototypes.

6.1. Python Prototype

```
__author__ = 'GeorgHinkel'

import NeuroboticsFramework as nrp
from husky import Husky

# Transfer functions may in general have some state
# in this case, we reserve memory for a global double variable
right_arm_v = 0

# The annotation Neuron2Robot registers this function as a transfer function
# As the parameter neuron0 is not explicitly mapped, the framework will assume a mapping
# to the neuron with the GID neuron0
neurons@nrp.Neuron2Robot(Husky.RightArm.pose)
def right_arm(t, neuron0):
    global right_arm_v
    if neuron0:
        right_arm_v = 1
    else:
        right_arm_v /= 2
    return right_arm_v

# Here is a another transfer function from neurons to robot messages
# This time, the neuron parameter is explicitly mapped to an array of
@nrp.MapNeuronParameter("neuron2", nrp.spikes("neuron2", 10))
@nrp.Neuron2Robot(Husky.LeftArm.twist)
def left_arm_tw(t, neuron1, neuron2):
    if neuron1:
        if neuron2[0]:
            return 0
        else:
            return 1
```

```
    else:
        if neuron2[1]:
            return 0.75
        else:
            return 0.25


# Here is an example of a transfer function mapping robot sensor data to spikes
# As the image processing is a common task, this is done through a specialized
# device in the neuronal simulator. However, this device might not be mapped to
# physical Nest device, but do some processing internally and use a less specialized
# device type internally
@nrp.MapRobotParameter("camera", Husky.Eye.camera)
@nrp.MapNeuronParameter("camera_device", nrp.CameraSpikeGenerator(200, 300))
@nrp.Robot2Neuron(nrp.spikes("spike45", 200000))
def transform_camera(t, camera, camera_device):
    if camera.changed:
        camera_device.update_image(camera.value)

if __name__ == "__main__":
    nrp.initialize()
```

This prototype has an implementation that when executed prints the list of all available transfer functions as well as their mappings to ROS topics and spikes. The prototype here also relies on an imported husky.py script that should be generated from the ROS information model of the robot. This script specifies the ROS topics referenced from within the TF script.

The output of the script is as follows:

```
<function right_arm at 0x2081af0> transfers to robot /husky1/joint325/pose : float () using [spike0(1)]

<function left_arm_tw at 0x2081e20> transfers to robot /husky1/leftArm/twist : float () using [spike1(1), spike2(10)]

<function transform_camera at 0x20e4490> transfers to robot spike45(200000) () using [/husky1/sensors/camera1 : list]
```

This information could be used e.g. to generate a MUSIC configuration file. With different command line arguments, it can start a ROS node, subscribe or publish the ROS topics used by the transfer functions and run the node according to the specification in Section 5.

The prototype does not contain any information regarding the used technologies, i.e. there is nothing in the script that depends on MUSIC, ROS or even Nest. All communication channels can be exchanged by modifying the implementation of the TF framework without changing the implied API of the prototype.

6.2. C++ Prototype

The C++ prototype is for the current state of the project obsolete. However, the prototype is still contained in this document in order to document the original ideas in case the performance implications of using Python are too heavy and we switch back to C++. However, changing back to C++ would be difficult, as converting the data between Python and C++ eats most of the benefit from using C++ over using Python. Thus, this should be considered carefully.

As a consequence, the C++ prototype may be outdated and not aligned with the current architecture.

```
/*
 * MyTransferFunctions.cpp
 *
 *  Created on: 11.08.2014
 *      Author: GeorgHinkel
 */

#include "NeuroboticsFramework.h"
#include "husky.h"
#include <string>

#include "ComplexImageProcessing.h"
#include "ObstacleDetection.h"

using namespace Neurobotics;

double left_arm_v = 0;
bool* spike1;
ImageSpikeGenerator camera_spike_generator(300, 200);


double moveLeftArm(const int t, const bool spike0) {
	if (spike0) {
		left_arm_v = 1;
	}else {
		left_arm_v /= 2;
	}
	return left_arm_v;
}


double moveRightArmMuscles(const int t, const bool* right_arm_spike) {
	if (spike1) {
		if (right_arm_spike[1]) {
			return 1;
		}else {
			return 0;
		}
	}else {
		if (right_arm_spike[2]) {
			return 0.75;
		}else {
			return 0.25;
		}
	}
}


bool* sendCameraData(const int t, const CacheValue<char*> sensor) {
	if (sensor.Changed) {
		camera_spike_generator.updateImage(sensor.Value);
	}
	return camera_spike_generator.tick(t);
```

```
}


int main(int argc, char** argv) {

        spike1 = RegisterReadNeuron("spike1");

        TransferNeuronToRobot("spike0", husky::left_arm::elbow::pose, moveLeftArm);
        TransferNeuronsToRobot("right_arm_spike", husky::right_arm::muscles,
moveRightArmMuscles, 10);

        TransferRobotToNeurons(husky::eye::camera, "eye_spikes", sendCameraData, 2000);

        StartNode("MyTransferFunctions", argc, argv);
}
```

The C++ way of specifying transfer functions is rather thought for functions where the performance of the transfer function is important, particular because complex (image) processing is involved, e.g. for obstacle detection. Thus, the specification files should have a possibility to include such libraries.

The intended output of this prototype is a ready-to-run ROS node, i.e. the StartNode entry function of the TF framework will start a ROS node and configure the TF framework from the previous registration methods. The transfer functions represented by the prototype match exactly the functionality of the Python prototype.

Also in this case, the included header file “husky.h” is meant to be generated in advance from the ROS information model of the robot. However, the prototype is currently not supported by an implementation, there is just a header file to make Eclipse show no warnings or error messages.

6.3. Suggested API

The below table lists the public API that is intended with the prototypes. The table lists how to accomplish a certain purpose using one of these prototypes.

Purpose	Python	C++
Register a transfer function to transform spikes into messages for the robot	Neuron2Robot decorator	TransferNeuronsToRobot, TransferNeuronToRobot
Register a transfer function to transform robot messages into spikes	Robot2Neuron decorator	TransferRobotToNeuron, TransferRobotToNeurons
Register that a transfer function will read spikes from a neuron	Inferred by the parameter name of a function, or explicit MapNeuronParameter annotation	One neuron with the registration, others via RegisterReadNeuron
Register that a transfer function will generate spikes to a neuron	Parameters of Robot2Neuron annotation, first neuron(s) is/are connected to function return value	One neuron with registration, others via RegisterWriteNeuron, RegisterWriteNeurons
Register that a transfer function will read robot topics	MapRobotParameter annotation	One robot topic with the registration, others via RegisterReadTopic

Register that a transfer function will write robot topics	Parameters of Neuron2Robot annotation, first topic is connected to return value	One topic with registration, others via RegisterWriteTopic
Send spikes to a neuron	sendSpikes	Via object obtained from RegisterWriteNeuron(s)
Send a message to the robot	sendRobot	Via object obtained from RegisterWriteTopic

For more detail, the prototype implementation with documented methods can be obtained under

git clone https://bbpteam.epfl.ch/reps/user/hinkel/tf-framework.git