A Domain-Specific Language (DSL) for Integrating Neuronal Networks in Robot Control

Georg Hinkel
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
hinkel@fzi.de

Henning Groenda
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
groenda@fzi.de

Lorenzo Vannucci
The BioRobotics Institute at Scuola Superiore Sant'Anna
viale Rinaldo Piaggio 34
56025 Pontedera, Italy
l.vannucci@sssup.it

Oliver Denninger
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
denniger@fzi.de

Nino Cauli
The BioRobotics Institute at Scuola Superiore Sant'Anna
viale Rinaldo Piaggio 34
56025 Pontedera, Italy
n.cauli@sssup.it

Stefan Ulbrich
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe, Germany
sulbrich@fzi.de

ABSTRACT

Although robotics has made progress with respect to adaptability and interaction in natural environments, it cannot match the capabilities of biological systems. A promising approach to solve this problem is to create biologically plausible robot controllers that use detailed neuronal networks. However, this approach yields a large gap between the neuronal network and its connection to the robot on the one side and the technical implementation on the other.

Existing approaches neglect bridging this gap between disciplines and their focus on different abstractions layers but manually hand-craft the simulations. This makes the tight technical integration cumbersome and error-prone impairing round-trip validation and academic advancements.

Our approach maps the problem to model-driven engineering techniques and defines a domain-specific language (DSL) for integrating biologically plausible Neuronal Networks in robot control algorithms. It provides different levels of abstraction and sets an interface standard for integration.

Our approach is implemented in the Neuro-Robotics Platform (NRP) of the Human Brain Project (HBP[1]). Its practical applicability is validated in a minimalist experiment inspired by the Braitenberg vehicles based on the simulation of a four-wheeled Husky[2] robot controlled by a neuronal network.

[1]http://www.humanbrainproject.eu
[2]http://www.clearpathrobotics.com/husky/

MORSE/VAO '15, July 21 2015, L'Aquila, Italy

ACM ISBN 978-1-4503-3614-7/15/07.
DOI: http://dx.doi.org/10.1145/2802059.2802060

CCS Concepts

•Software and its engineering → Model-driven software engineering; Domain specific languages; •Computing methodologies → *Control methods; Neural networks; Bio-inspired approaches;*

Keywords

Neurorobotics, Domain-Specific Languages, Model-Driven Engineering

1. INTRODUCTION

A goal in many robotics application in natural environments is the exposition of a maximally natural behavior. In nature, such behavior is encoded in biological neuronal networks which are subject to research of neuro-physiologists who in turn may not have a strong background in computer science. In particular, 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 connecting neuronal network models.

Knowledge on how neuronal networks are connected to muscles in biology (and analogical actuators in robotics) is extremely valuable for both neurophysiology and robotics. Furthermore, the simulation of neuronal networks connected to robots offers a method to validate models of neurophysiology and thus pushes the understanding of how this is achieved in biology.

As an example, the bio-inspired walking machine LAURON V [1] is inspired by the stick insect Carausius morosus with four joints per leg, depicted in Figure 1. In this setting, the integrated simulation of LAURON V connected to a neuronal network on the one hand yields a promising approach to obtain biologically plausible robot controllers for LAURON V but also gives neuroscientists a way to validate their understanding of the neurophysiology of Carausius morosus.

Raising the particular abstraction level to formulate solutions in the *problem domain* rather than in a technical *implementation domain* is also an important goal of model-driven

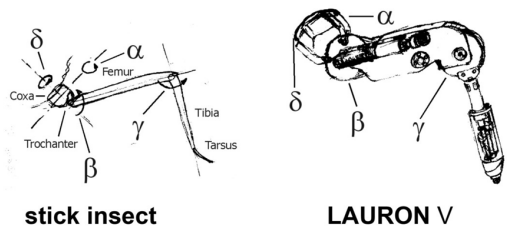

Figure 1: The kinematics of LAURON V compared to the stick insect Carausius morosus

engineering techniques [2]. The terminology and techniques can bridge the gap between robotics and neurophysiology by analogously modeling these connections on a high abstraction level [3].

In this paper, we present our approach of applying techniques and processes from the field of model-driven engineering for integrating neuronal networks with robot control. In particular, we created a Domain-Specific Language (DSL) in Python. The applicability of the concept is then validated by the simulation of a minimalist experiment inspired by Braitenberg vehicles [4]. This experiment is also used throughout the paper as a running example.

The paper is structured as follows: Section 2 provides an overview of our approach and its implementation in the Neuro-Robotics Platform. In this section, we also map our approach to the model-driven process by Völter and Stahl [2]. Section 3 introduces the experiment which we use as our running example. Sections 4 to 6 describe the approach according to the mapping introduced in Section 2. Section 4 explains the internal Python DSL to specify the connection of neuronal and robot simulations. Section 5 discusses the usage of a formal model with a higher abstraction level. Section 6 presents the transformation from the formal model to the Python DSL. Section 7 presents simulation results of our approach implemented in the Neuro-Robotics Platform (NRP). Finally, Section 8 lists related work before Section 9 concludes the paper and provides an outlook on future research.

2. THE NEURO-ROBOTICS PLATFORM

A round-trip validation of Neuronal Network algorithms controlling a robot in a virtual or real environment requires a solid evaluation platform covering all disciplines. We use the Neuro-Robotics platform (NRP) developed in the scope of the Human Brain Project (HBP) as technical basis. The platform simulate neuronal network models connected to a simulated robot through Transfer Functions (TFs) which translate neuronal activity into the desired control signals for robots.

The NRP consists of the following key components:

Neuronal (Brain) Simulator:.

To simulate the neuronal networks, the neuronal simulator NEST [5] is used. This simulator was designed to run within a distributed and parallel environment. This is especially important given the ultimate goal to simulate the human brain with about 10^{11} neurons and 10^{15} synapses. On the other hand, we access the neuronal simulator through the PyNN [6] interface so we are able to exchange the neuronal simulator by other implementations such as SpiNNaker [7] which runs on specialized neuromorphic hardware.

World Simulator:.

To physically simulate the robots and their environment, the Gazebo simulator[8] is used communicating to the simulated robot through the Robot Operating System (ROS)[9]. In particular, we use the asynchronous event-based communication through ROS topics. This allows identifying parts of the robot only by its topic address and type. Using ROS as middleware also yields the possibility to easily exchange the simulated robot by a physical counterpart.

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 that describe a function together with specifications on how the parameters are connected either to neurons of the neuronal network or to topics of the simulated robot. These transfer functions can be specified both through an internal domain-specific language (DSL [10]) in Python and through an XML format. We target to create these XML representations by means of a graphical designer in future releases.

While these components are static in the sense that they are reused for every simulation run on the platform, other parts of the simulation code are dependent on the given setup, i.e. the simulated robot, the neuronal network and the connection in between.

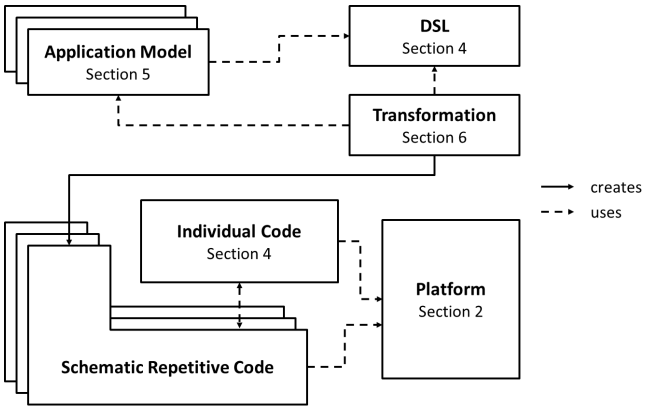

Figure 2: Model-driven software development as proposed by Völter and Stahl [2]

To assemble the simulation, we use an approach similar to the artifacts of model-driven software development as proposed by Völter and Stahl [2]. An overview is shown in Figure 2 which also depicts the Sections in which the artifacts are described. The platform are the key components of the NRP. The schematic repetitive code is the connection between the neuronal network and the simulated robot that is described by a DSL and a model. Unlike the original proposal by Völter and Stahl, we use two domain-specific languages on two different abstraction levels to take account for different expertise among neuroscientists. As a consequence, the low-level DSL appears twice in the diagram of Figure 2, as

a DSL per se and as individual code from the perspective of the high-level DSL.

3. THE BRAITENBERG VEHICLE EXPERIMENT

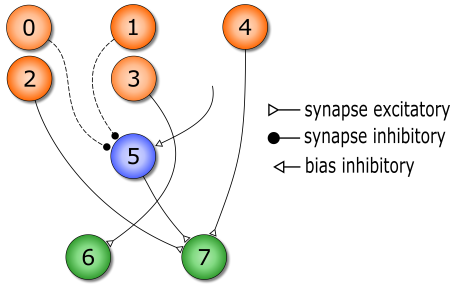

Figure 3: The neuronal network for the simple Braitenberg experiment

As an example experiment to launch on the NRP, we use an experiment inspired by the Braitenberg vehicles [4]. There, a four-wheeled Husky robot equipped with a camera is simulated in a virtual room. A simplistic neuronal network consisting of just 8 neurons recognizes red colors and lets the robot move towards the red color. In the neuronal network, shown in Figure 3, the five neurons in orange (numbers 0 to 4) are bundled in a population that represent the sensors of the network. They receive the input signal through Poisson generators generating spikes at a rate depending on how many red pixels have been detected in the robot's camera image that were classified as red pixels. This classification is done through a library function categorizing the pixels according to the HSV color model. This information is propagated through the network until the voltage of the actor neurons 6 and 7 (in green) is used as voltage on the left and right wheel motors of the robot.

4. A PYTHON DSL FOR TRANSFER FUNCTIONS

Transfer Functions can be specified using the Transfer Functions framework, which effectively offers an internal DSL hosted in Python. We use Python mainly because there is an API both for the neuronal simulations and also for the robotics side since ROS provides a Python API. As a consequence, Python is popular both among robotics and neuroscience users. Given the research results from Meyerovich [11] that suggest that developers do not like to change their primary language, we wanted to make the barrier for neuroscience users as low as possible and therefore created a Python API. This API allows to specify Transfer Functions by decorating usual Python functions, giving it the flavor of an internal domain-specific language (DSL) [10].

4.1 Transfer Functions Neuron to Robot

This section describes the information flow from the neuronal network to the robot. In the example, the voltage of the actor neurons is to be transmitted to the robot. But as the robot simulation requires to specify movement of the robot in terms of angular and linear progression, the voltages 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 voltages must be scaled to achieve good results.

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

Listing 1: Transfer Function from neurons to the robot in the Python DSL

This connection can be specified in our Python DSL as shown in Listing 1. Line 1 simply imports the Transfer Function framework into the current script. Line 2 imports the robot message type. Lines 4-10 are an example of a Transfer Function translating the voltage of actor neurons into robot commands. This is done by a Python function with a set of decorators. The decorator `@nrp.Neuron2Robot` in line 6 marks the function as a Transfer Function from the neuronal network towards the robot and automatically registers it at a singleton component instance to manage the Transfer Functions. Furthermore, the decorator specifies what the platform should do with the function's 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 and 5 specify how the parameters of the function should be mapped to the neuronal network. In this case, the parameters should be connected to two single neurons of the *actors* population through a leaky integration algorithm. That is, the spikes coming from the neuronal network are simply integrated in a neuron with an infinite spiking threshold, thus the device effectively measures the voltage of the neurons combined. The first parameter t is reserved for the current simulation time and cannot be remapped through a decorator. All other parameters must be mapped to either robot parts or neurons.

The fact that the decorators turn a usual function into a transfer function gives the API the flavor of a domain-specific language as the underlying method is no longer accessible to the outside. In particular, the name `linear_twist` in Listing 1 is resolved in Python to an object representing the Transfer Function instead of the method underneath. Once the framework is initialized and the connections such as specified in Lines 4-6 are established, the connected devices can be accessed directly, e.g. through `linear_twist.left_wheel_neuron`. As a valuable side effect, this enables users of the platform that like to use automated tests to test their Transfer Functions when connecting them to mockups of the simulations.

4.2 Transfer Functions Robot to Neuron

In the opposite direction, we take a camera image from the world simulator, detect red colors and use the results as stimuli for the neuronal network. In this setting, the classification is currently implemented in a library function. This library function computes for an image the ratio of red pixels in the left half of the image, the ratio of red pixels in

the right half of the image and the ratio of pixels classified as not red in the whole image. These ratios are propagated to the input sensor neurons by means of Poisson generators that create spikes according to a Poisson distribution with a rate depending on the detected red color.

```
1  @nrp.MapRobotSubscriber("camera", Topic('/husky/camera', sensor_msgs.
       msg.Image))
2  @nrp.MapSpikeSource("red_left_eye", nrp.brain.sensors[0, 2], nrp.
       poisson)
3  @nrp.MapSpikeSource("red_right_eye", nrp.brain.sensors[1, 3], nrp.
       poisson)
4  @nrp.MapSpikeSource("green_blue_eye", nrp.brain.sensors[4], nrp.
       poisson)
5  @nrp.Robot2Neuron()
6  def eye_sensor_transmit(t, camera, red_left_eye, red_right_eye,
       green_blue_eye):
7      image_results = nrp.tf_lib.detect_red(camera.value)
8
9      red_left_eye.rate = 1000.0 * image_results.leftred
10     red_right_eye.rate = 1000.0 * image_results.rightred
11     green_blue_eye.rate = 1000.0 * image_results.greenblue
```

Listing 2: Transfer Function from a camera image to neuron spikes

The implementation of this connection is shown in Listing 2. The decorator `@nrp.Robot2Neuron` in line 5 marks the function as a Transfer Function from the world simulation to the neuronal network. Line 1 is responsible to map the *camera* parameter to a subscriber on the camera topic of the robot. In lines 2-4, we create three Poisson generators that will issue spikes according to a Poisson distribution. We use Poisson generators since alternative spike sources generating spikes in a fixed frequency are more affected by time resolution. The method body itself is not restricted in any way, so we have no limitation in the expressiveness of the language.

5. APPLICATION MODELS FOR BRAIN AND BODY INTEGRATION

To further simplify the specification of such connections and to enhance validation, we are aiming for a graphical designer, the Brain Interface and Body Integrator (BIBI). Although our requirements demand us to support arbitrary Transfer Functions, in many cases these Transfer Functions simply consist of predefined functional building blocks stacked together. Thus, the full execution semantic of Python is not necessary in these cases and a metamodel of stacked functional building blocks suffices. On the other side, a formal model has a couple of advantages such as model validation and the definition of an abstract syntax to create an editor for it.

In particular, model validation is a particularly important concern. Early validation of Transfer Functions increases locality and reduces the time to error reports. Both have been identified in Software Engineering as success factors for efficient failure management.

An illustration how models at different levels of abstraction are used in the NRP is shown in Figure 4. The user may choose between two ways how to specify the integration of brain and body, either through the Python DSL we have presented in Section 4 or through a graphical designer. The transfer functions directly contained in the BIBI Model are then transformed into the Python DSL by means of a model-to-text transformation. This allows a migration path if the expressiveness of the formal model does not suffice as users can then continue to work with the underlying Python DSL

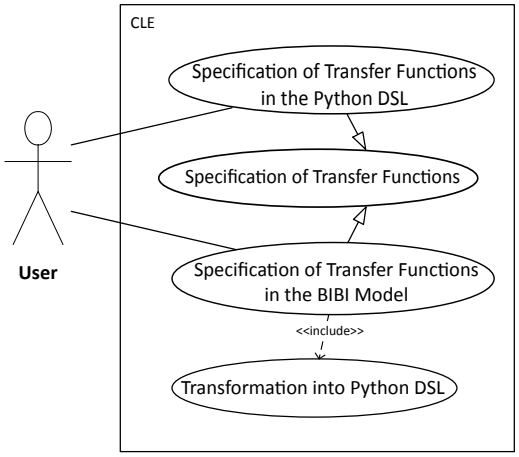

Figure 4: Usage of different abstraction models

directly. As the BIBI Model supports to reference Python files, both ways to specify Transfer Function may be used simultaneously while only one model interpreter is necessary.

An overview of the general structure of the BIBI Model is depicted in Figure 5. The model is implemented in an XML Schema conforming to the XMI standard since it is available on more technical platforms than modeling environments such as Ecore. Our approach uses generateDS[3] to automatically generate Python classes to read the models. The use of XML Schema limits stable cross-references of elements, which are not part of the containment hierarchy. The design accordingly prevents those cross-references where possible.

The main assets of these Transfer Functions in the BIBI Model are the specification of used channels, i.e. the connections they have to either the simulated robot (topic channels) or the neuronal network through devices (device channels). The device channels augment the neuronal simulation with a device, an intermediate object necessary since the neuronal network typically runs in a higher time resolution than the Transfer Function. Their task is to distribute input values from a Transfer Function or aggregate values from the neuronal network. Typical examples of devices include voltmeters, current generators or Poisson generators.

Channels can either be read from or written to, according whether the channel is used as input or output from the respective simulator. Further, Transfer Functions can also specify local variables that act both as input and output within the Transfer Function. Whether a channel is read from or written to is determined by whether it has an expression assigned to it as its body or not.

For such expressions, we have a simple expression language limited to the prospected needs of Transfer Functions including arithmetic operations, minima and maxima. However, the language also contains an element to include library calls, ensuring its extensibility.

The additional channel configuration is different for device channels and robot channels. For a topic channel, we assume that all connectible parts of the robot are accessible through named and typed topics where the default implementation are ROS topics.

The configuration of device channels is depicted in Figure

[3]https://pypi.python.org/pypi/generateDS

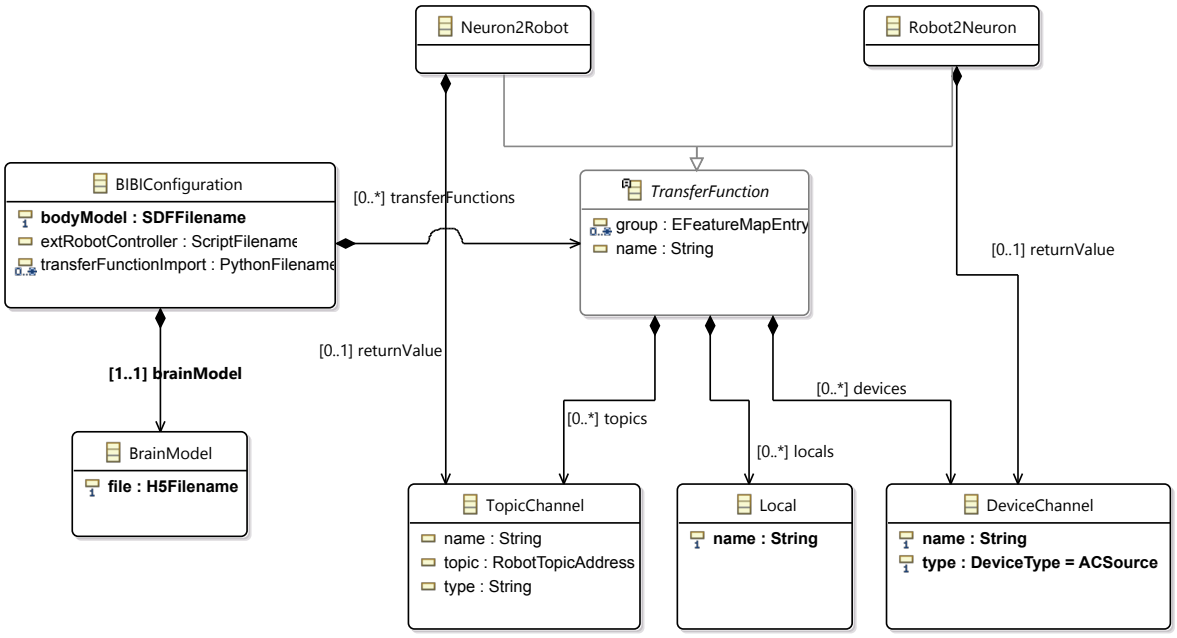

Figure 5: An overview on the structure of the BIBI Model

6. Besides an expression what values should be assigned to the device, the device channel also incorporates a neuron selector and the device type such as a leaky integrator or a Poisson generator. This neuron selector selects the neurons the device should be connected to. Currently we support the selection of a whole population of neurons in the neuronal network, a range of neurons or a list of neurons. Future neuronal network models we intend to use also specify a notion of location for neurons. Based on this location, more elaborate selectors will include a biologically plausible selection by a neurons physical location.

6. TRANSFORMATION

To avoid inconsistencies between artifacts representing the same entities on different levels of abstraction, transformation approaches have proven to be viable. In our case, it is the CLE that runs the Transfer Functions. As we allow the specification of Transfer Functions in two different formats, inconsistencies could arise from different interpretations of how Transfer Functions are connected to the neuronal network or to the robot. To prevent these inconsistencies, we transform the Transfer Functions specified in the formal BIBI model to the Python DSL. That is, Transfer Functions specified in the BIBI Model are transformed into Transfer Functions in the Python DSL. Thus, for the CLE the fact that a formal BIBI model is on top is entirely transparent since it is only based on the Python DSL.

The alternative solution of using a model interpreter does not allow users to understand what is happening under the hood whereas the transformation approach allows users to see the generated Python DSL code for a given Transfer Function created in the BIBI Model. The transformation approach also provides a migration path if the expressiveness of the formal model does no longer suffice.

For consistency reasons, this transformation is also written in Python code where we use Jinja2[4], a templating engine usually used for the generation of HTML files.

```
1  {% for tf in config.transferFunction %}{% if tf.extensiontype_ == '
       Neuron2Robot' %}
2  {% for topic in tf.topic %}{% if is_not_none(topic.body) %}
3    @nrp.MapRobotPublisher("{{topic.name}}", Topic('{{topic.topic}}',
         {{topic.type_}}))){% else %}
4    @nrp.MapRobotSubscriber("{{topic.name}}", Topic('{{topic.topic}}',
         {{topic.type_}}))){% endif %}{% endfor %}{% for dev in tf.
         device %}{% if is_not_none(dev.body) %}
5    @nrp.MapSpikeSource("{{dev.name}}", nrp.brain.{{get_neurons(dev)}},
          nrp.{{get_device_name(dev.type_)}}){% else %}
6    @nrp.MapSpikeSink("{{dev.name}}", nrp.brain.{{get_neurons(dev)}},
         nrp.{{get_device_name(dev.type_)}}){% endif %}{% endfor %}
7    @nrp.Neuron2Robot({% if is_not_none(tf.returnValue) %}Topic('{{tf.
         returnValue.topic}}', {{tf.returnValue.type_}}){% endif %})
8    def {{tf.name}}(t{% for t in tf.topic %}, {{t.name}}{%endfor%}{%
         for dev in tf.device %}, {{dev.name}}{%endfor%}):
9  {% for local in tf.local %}
10     {{local.name}} = {{print_expression(local.body)}}{% endfor %}
11 {% for dev in tf.device %}{% if is_not_none(dev.body) %}
12     {{dev.name}}.{{get_default_property(dev.type_)}} = {{
           print_expression(dev.body)}}{% endif %}{% endfor %}
13 {% for top in tf.topic %}{% if is_not_none(top.body) %}
14     {{top.name}}.send_message({{print_expression(top.body)}}){%
           endif %}{% endfor %}
15 {% if is_not_none(tf.returnValue) %}
16     return {{print_expression(tf.returnValue.body)}}{% endif %}
```

Listing 3: Transformation of the BIBI Model to the Python DSL

An excerpt of this template showing the transformation for Transfer Functions from the neuronal network to the robot is shown in Listing 3. An advantage of the Jinja2 templating engine is that it supports the inclusion of arbitrary Python code. Therefore, more complex predicates can be implemented in normal Python code and can thus be unit tested. This is important given the difficulties of testing model transformations [12].

[4]http://jinja.pocoo.org/docs/dev/

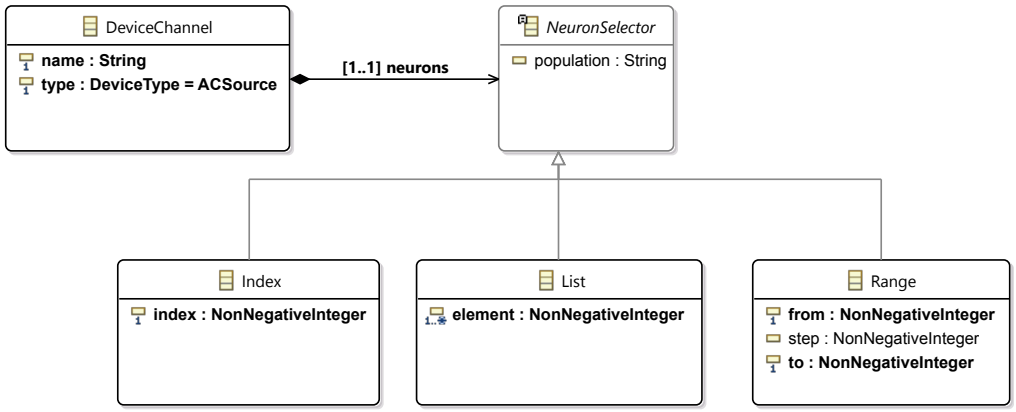

Figure 6: The connection to the neuronal network in the BIBI Configuration model

However, as shown in Listing 3, the code generation template can be difficult to read and thus maintain, particularly if it is a goal that the generated code has a reasonable formatting. This is important to users that want to understand what Python DSL code is generated for a particular BIBI model.

7. THE BRAITENBERG VEHICLE EXPERIMENT ON THE NRP

This section describes the use of the Braitenberg vehicle experiment introduce in Section 3 on the NRP. The Husky robot is simulated in a realistic virtual room equipped with two screens that may be turned red by the user during the simulation. The neuronal network (presented in Section 3) guides the robot to turn unless it sees red color and then moves towards it. Figure 7 shows a screenshot of the simulation of the experiment and a video is publicly available online[5].

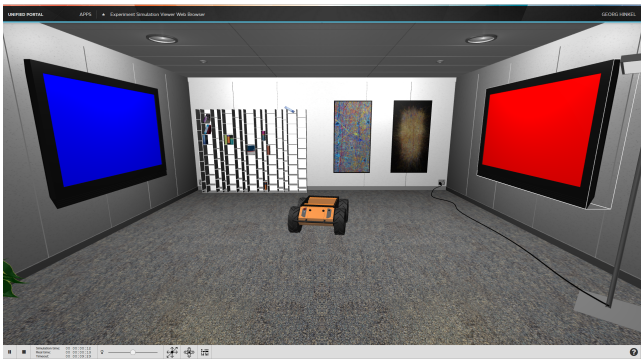

Figure 7: The Braitenberg vehicle simulated in the NRP platform

The platform takes as input an experiment model. This model contains a reference to the virtual room in which the robot is loaded for simulation as well as a BIBI Model. In this case, all Transfer Functions are specified within the BIBI Model. These Transfer Functions are transformed into the Python DSL which is then executed by the CLE.

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

However, so far the platform has only been released within the HBP consortium and we do not have any experience reports on further validation areas such as the expressiveness of both the Python DSL and the BIBI model and leave such analysis for future work.

8. RELATED WORK

Despite software playing a basic role in implementing the functionality of robotics systems, most robotics software systems are still hand-crafted. A model-based approach is commonly accepted as more suitable to cover the control or mechanical design aspects. In the last years, a migration from a code-driven approach towards a more flexible model-based one has started to emerge[13], [14].

But even though approaches to simulate a neuronal network connected used to control robots can be traced back at least until the early nineties [15], we could not identify domain specific languages designed to support this connection for neuro-scientist and covering biological inspired neuronal networks. However, there are some DSLs that describe either the neuronal network simulation [16], [17] or robotics [18], our language is the first to describe their interplay on a high abstraction level.

In the field of neuroscience, an example of such a language can be found in the NEURON simulator [16], [17], whose network models can be described in a DSL based on Hoc [19]. Furthermore, current research projects such as NESTML[6] are trying to create domain-specific languages for the NEST simulator [5].

From a robotic point of view, several works have been proposed, covering some specific aspects of robotic software systems [20]–[22]. Nordmann et al. have published a list of DSLs in robotics[7] and created a survey of these [18]. Most of these languages utilize the knowledge of a particular sub-domain of robotics to create an abstract syntax and a DSL for it. From this DSL, the entire robot controller or at least large parts of it can be generated. This is different to our approach where we assume the robot controller to be already existing,

[6]http://www.jara.org/en/research/jara-hpc/research/details/seed-funds-zuk2/nestml-eine-modellierungssprache-fuer-biologisch-realistische-neuronen-und-synapsenmodelle-fuer-nest/

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

implemented as a neuronal network that has to be connected to the robot.

9. CONCLUSION AND FUTURE WORK

In this paper, we have presented an approach to bridge the semantic gap involved in the specification of a coupling between a neuronal network and a simulated robot. We have presented both a textual DSL in Python for users with a computer science background and also a meta-model for Transfer Functions on a high abstraction level. This meta-model paves the way for graphical languages that allow to specify such connections for users with little or no computer science background. Furthermore, it fosters the validation of these models. Through a transformation approach, these two abstraction levels can be mixed.

In the short-term, we plan to create an editor for the graphical DSL easing the specification for the brain and body integration. The Python DSL has just been released together with the NRP within the HBP and we are looking forward to comments and suggestion how to further improve the applicability according to our users needs. In the mid-term, we are going to implement analyses and constraint checks to ensure that Transfer Functions reference valid input and output of Brain and Body.

Acknowledgment

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).

References

[1] A. Roennau, G. Heppner, M. Nowicki, and R. Dillmann, "LAURON V: a versatile six-legged walking robot with advanced maneuverability," in *Advanced Intelligent Mechatronics (AIM), 2014 IEEE/ASME International Conference on*, IEEE, 2014, pp. 82–87.

[2] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, and K. Czarnecki, *Model-Driven Software Development: Technology*. 2006.

[3] P. Trojanek, "Model-driven engineering approach to design and implementation of robot control system," 2013.

[4] V. Braitenberg, *Vehicles: Experiments in synthetic psychology*. MIT press, 1986.

[5] H. Plesser, J. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig, "Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers," in *Euro-Par 2007 Parallel Processing*, ser. LNCS, vol. 4641, Springer Berlin Heidelberg, 2007, pp. 672–681.

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

[7] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, "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*, IEEE, 2008, pp. 2849–2856.

[8] N. Koenig and A. Howard, "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*, IEEE, vol. 3, 2004, pp. 2149–2154.

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, 2009, p. 5.

[10] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[11] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, ACM, 2013, pp. 1–18.

[12] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Communications of the ACM*, vol. 53, no. 6, pp. 139–143, 2010.

[13] C. Schlegel, T. Haßler, A. Lotz, and A. Steck, "Robotic software systems: from code-driven to model-driven designs," in *Advanced Robotics, 2009. ICAR 2009. International Conference on*, IEEE, 2009, pp. 1–8.

[14] C. Atkinson, R. Gerbig, K. Markert, M. Zrianina, A. Egurnov, and F. Kajzar, "Towards a Deep, Domain Specific Modeling Framework for Robot Applications," pp. 1–12.

[15] D. A. Pomerleau, "Neural network perception for mobile robot guidance," PhD thesis, DTIC Document, 1992.

[16] M. Hines, "A program for simulation of nerve equations with branching geometries," *International journal of biomedical computing*, vol. 24, no. 1, pp. 55–68, 1989.

[17] A. P. Davison, M. L. Hines, and E. Muller, "Trends in programming languages for neuroscience simulations," *Frontiers in neuroscience*, vol. 3, no. 3, p. 374, 2009.

[18] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Simulation, Modeling, and Programming for Autonomous Robots*, Springer, 2014, pp. 195–206.

[19] B. W. Kernighan and R. Pike, *The Unix programming environment*. Prentice-Hall Englewood Cliffs, NJ, 1984, vol. 270.

[20] M. Frigerio, J. Buchli, and D. G. Caldwell, "A domain specific language for kinematic models and fast implementations of robot dynamics algorithms," 2013.

[21] M. Bordignon, U. P. Schultz, and K. Stoy, "Model-based kinematics generation for modular mechatronic toolkits," in *ACM SIGPLAN Notices*, ACM, vol. 46, 2010, pp. 157–166.

[22] D. Di Ruscio, I. Malavolta, and P. Pelliccione, "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.