Experiences with Model-Driven Engineering in Neurorobotics

Georg Hinkel(✉), Oliver Denninger, Sebastian Krach, and Henning Groenda

Software Engineering, Forschungszentrum Informatik (FZI),
Haid-und-Neu-Straße 10-14,
Karlsruhe, Germany
{hinkel,denninger,krach,groenda}@fzi.de

Abstract. Model-driven engineering (MDE) has been successfully adopted in domains such as automation or embedded systems. However, in many other domains, MDE is rarely applied. In this paper, we describe our experiences of applying MDE techniques in the domain of neurorobotics – a combination of neuroscience and robotics, studying the embodiment of autonomous neural systems. In particular, we participated in the development of the Neurorobotics Platform (NRP) – an online platform for describing and running neurorobotic experiments by coupling brain and robot simulations. We explain why MDE was chosen and discuss conceptual and technical challenges, such as inconsistent understanding of models, focus of the development and platform-barriers.

1 Introduction

The field of neurorobotics uses insights from neuroscience to build robot controllers using neural networks. Of particular interest is the combination of biologically plausible spiking neural networks with robots. This enables neurophysiologists to study how brains can be connected to bodies, neuroscientists to study brain models in the real world and robotic scientists to perform locomotion or perception tasks – which are hard to solve with classical robot controllers – using the neural networks' ability to learn and adapt.

Building spiking neural networks is increasingly understood by domain experts. Robotics has a long experience of modelling robots and building robot controllers. However, establishing a closed loop between both artifacts – this means transferring sensor information from a robot to a brain and control information from the brain back to the robot – is still an open question. Few scientists know both neural network simulation and robotics well enough in order to perform adequate experiments.

As a consequence, most existing experiments in neurorobotics are handcrafted simulation scripts, able to perform only a tightly defined experiment without variations. Such scripts may easily get obsolete when the interface of the components from either domain changes.

Therefore, it is necessary to abstract from the technical implementation details and allow neuroscientists to describe the interconnection between neural

© Springer International Publishing Switzerland 2016
A. Wąsowski and H. Lönn (Eds.): ECMFA 2016, LNCS 9764, pp. 217–228, 2016.
DOI: 10.1007/978-3-319-42061-5_14

networks and robots in a formal model. The simulations gain flexibility as common operations such as pausing, stopping, resetting or interacting with the simulation can be implemented once, based on the formal model. Flexibility in accessing the model is crucial as users want to build and run experiments interactively as well as non-interactively. The interactive style is well-known to robotics where experiments are built iteratively with visualization close to real-time. In contrast, neural network simulation experiments are typically run as batch jobs.

Raising the abstraction level in order to limit the description of a system to domain concepts rather than implementation details is also one of the major goals of model-driven engineering (MDE). Hence, we have adopted MDE techniques in the development of the Neurorobotics-Platform (NRP), an integrated simulation platform to allow neuroscientists to specify a neurorobotics simulation on a high level of abstraction.

In this paper, we present our experiences in applying model-driven techniques in the domain of neurorobotics that we gained during the development of the NRP. We observed inconsistencies in the understanding of models that imply communication problems bringing together experts of the involved matters. A lack of good test concepts for the code generators has made us shift functionality towards the target platform and keep code generators as small as necessary. For the choice of generators or any other tools, we faced a platform barrier. Our agile Scrum development process seemed incompatible with the upfront initial effort implied by the model-driven software development approach we took.

The remainder of this paper is structured as follows: Sect. 2 briefly introduces the NRP. Section 3 discusses the potential advantages offered by MDE in neurorobotics. Section 4 details on the lessons learned during the development of the NRP. Finally, Sect. 5 concludes the paper.

2 The Neurorobotics Platform in a Nutshell

The Neurorobotics Platform (NRP) is developed as part of the Human Brain Project[1] to run coupled neuronal and robotics simulations in an interactive platform. Whereas there are multiple neuronal simulators (e.g. Neuron [1], NEST [2]), robotics and world simulations (e.g. Gazebo [3]), the NRP aims to offer a platform uniting the two fields. A core part of the NRP is the Closed-Loop-Engine (CLE) that allows to specify the data exchange between the brain simulation and the robot in a programmatic manner and orchestrates the simulations.

The key concept of the NRP is offering scientists an easy access to a simulation platform using a state-of-the-art web interface. Scientists are relieved from the burdensome installation process of scientific simulation software and are able to leverage large-scale computing resources. Furthermore, support for monitoring and visualizing the spiking activity of the neurons or joint states of the robot is offered as well as the camera image perceived by the robot.

[1] https://www.humanbrainproject.eu/.

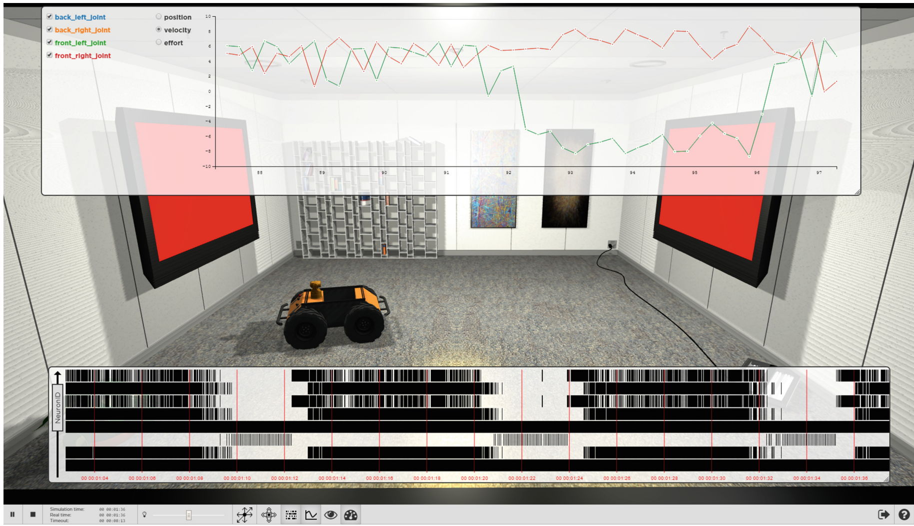

Fig. 1. Screenshot showing a Braitenberg vehicle inspired experiment in the Neurorobotics Platform (NRP). Upon perception of red color in the camera image, the robot moves forward, otherwise it keeps turning on the spot. A plot at the top shows velocity of the robot wheels while a plot at the bottom shows spiking activity of neurons. (Color figure online)

To give an impression on how the platform looks like, a screenshot of an experiment inspired by Braitenberg vehicles [4] using a Husky[2] robot is depicted in Fig. 1.

The different users of the NRP are depicted in Fig. 2. The NRP basically targets three science communities with overlapping fields of interest: *neuroscientists*, *neurophysicists* and *roboticists*. Neuroscientists are able to visualize brain models through embodiment. Neurophysiologists leverage the coupling mechanism of both simulations to analyze or validate models on signal transmission between perception, brain activity and motor control. Roboticists are able to validate and compare the performance of neuronal control compared to classic robot control approaches.

The platform aims to be usable by scientists with little programming knowledge. Based on templates, users can instantiate new experiments and adapt them at runtime using techniques familiar to the respective communities. Data interchange between a simulated brain and a robot is transcribed as so-called *Transfer Functions* using an internal Python-based DSL [5]. We use the Python programming language as it is generally accepted by neuroscientists and actively used for specifying brain models.

Figure 3 depicts the Closed-Loop-Engine which implements the core of the NRP. The CLE controls the neuronal and the world simulation, and realizes

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

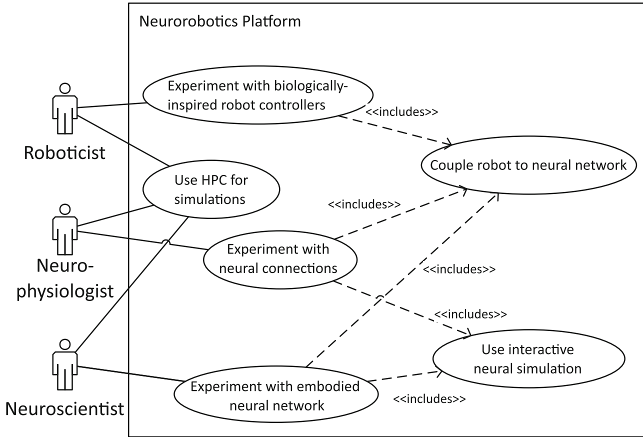

Fig. 2. The actors and their intentions of using the NRP

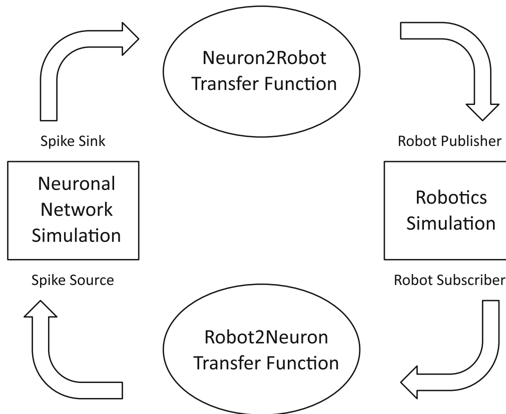

Fig. 3. A closed loop between a robotics simulation and a neural network

a lightweight simulation data interchange mechanism. Neuronal and robotics simulation are iteratively run in parallel for the same amount of time, after which the transfer functions are executed periodically. Communication with the brain is realized through recording and injecting spike data. Interfacing with the robot simulation is done using the middleware (ROS [6]). In order to ensure reproducibility, data exchange is conducted in a deterministic fashion.

Complex brain simulations require large scale computing resources and often exceed the capacities researchers have at hand. Furthermore, effectively leveraging computing resources provided by data centers is only available to neuroscientists with the appropriate competences or support from the computer scientific domain. The NRP provides a unified access to high-performance computing resources of different institutions. The shared infrastructure in particular

offers the possibility to run simulations using specifically designed neuromorphic hardware, which is currently not widely available.

Neuronal simulation in data centers are usually run as non-interactive batch jobs whose execution is scheduled once and runs to completion before reporting results to the neuroscientist. The scientist has no means of influencing the simulation after the job has been scheduled and errors can only be detected by analyzing the results. The NRP is designed to enable interactive manipulations of the neuronal network, the simulated environment or the robot. Thus, erroneous behavior can be detected very quickly.

3 The Potential of Model-Driven Engineering in Neurorobotics

In this section, we discuss why MDE is a suitable approach for developing a simulation platform such as the NRP. As the NRP aims to support the specification of data transfers between neural networks and robots accessible also by users with little programming expertise, the process of assembling the simulation code and its goals are very similar to model-driven software development. While a frequent rationale of model-driven software development (MDSD) is to reduce the development effort, we regard the higher abstraction level even as an enabling technology for users with little programming experience.

The artifacts for a model-driven software development process as introduced by Völter and Stahl [7] are depicted in Fig. 4.

The idea of model-driven software development is to divide the code of an application into three parts, (i) the platform, (ii) schematic repetitive code and (iii) individual code. From the repetitive code, a metamodel is extracted. Application models are then created as instances of this metamodel through a domain-specific language (DSL) and are transformed to the repetitive code by means of a model transformation.

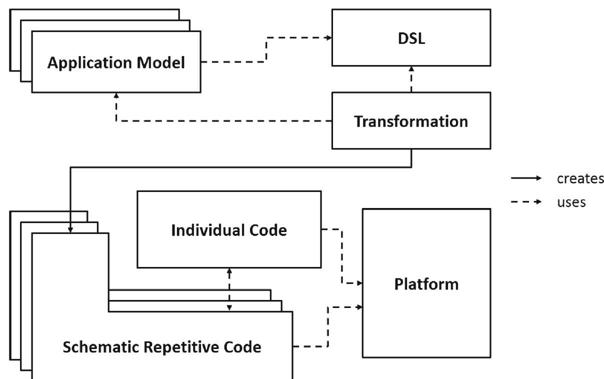

Fig. 4. Artifacts of model-driven software development as defined by Völter and Stahl [7]

The advantage is that the application model is very focused on domain concepts whereas technical implementation details are encoded in either the platform, the model transformation or the individual code. A goal of MDSD is clearly to keep the individual code as small as possible. The application models are thus usually specified in DSLs, either textual or graphical.

Understanding an experiment as an application, this model is also well suited to describe neurorobotics simulations of these experiments. Here, the ability to describe simulations in terms of domain concepts independent of technical implementation details allows also neuroscientists without a strong experience in programming languages to design and run experiments. At the same time, the technical implementation details can be exchanged in order to support new simulators. This is important as there is currently a multitude of neural network simulators available.

Another important aspect is the validation of experiments. As the involved neural networks become large, neurorobotics simulations require a large amount of resources. Additionally, especially in a web-based simulation platform accessible through the internet, simulations are run with an identity of the service host rather than the user. This raises security concerns in order to avoid the NRP to execute malicious code. Here, formal models can help by validating the models, whereas the code ultimately executed is generated and therefore can be trusted.

Finally, especially in neurorobotics, large assets of the experiments are likely to be reused. Hardly any neuroscientist will create both the robot, the neural network and its connection but rather reuse existing robots and potentially also neural networks and couple them in an experiment. This requires an introspection of both neural networks and robots for the neuroscientists to understand details of these artifacts such as e.g. the topology of the network or the kinematic of the robot.

These analyses, validations for security and potential mistakes as well as introspection, can be done independently of a running simulator if the experiment is available as a formal model. In this sense, the problems are similar to embedded systems, where formal models allow a verification, before a transformation converts them to integrated circuits.

We applied techniques and tools from model-driven engineering. In particular, we created formal metamodels to specify the connection between neural networks and robots, designed a DSL for it and created a transformation to generate the code to simulate an experiment [5].

4 Lessons Learned

In this section, we summarize our experiences applying model-driven techniques to the development of the NRP. These descriptions cover the domain analysis, development, tools, development process and project-internal communication. The selection of experiences presented here are the outcome of a brainstorming session reflecting on our experiences that we though could be interesting for others.

4.1 Inconsistent Understanding of Models

Though the notion of models has been clearly defined as early as 1973 by Stachowiak [8] and models are omnipresent in both robotics and neuroscience, there is a diversity of opinions how models should be implemented. Furthermore, as models always have a purpose, modeling standards of the same physical entities exist that incorporate abstractions for different usage scenarios.

In robotics, many modeling standards have been established. In the NRP, we came along Collada to describe the robots' appearance as a 3D-mesh, SDF, used in the Gazebo robotics simulator and the Unified Robot Description Format (URDF), used in the middleware ROS. While all of these standards are based on XML, neither of them complies to the XMI standard usually used in MDE. This has the consequence that tools such as generated parsers based on the XML Schema are not usable as the implied object-oriented class structure does not properly reflect the model. The framing XML language at least allows to reuse some functionality when writing parsers and allows some degree of validation.

Reusing these existing modeling standards seems appealing as this means that existing models can be easily reused, raising acceptance among users. However, one has to be very careful selecting the right model as it is hard to revoke this decision once the modeling standard used turns out to miss important aspects. For example, the SDF standard is used for simulation only and does not allow to inspect a robot model in terms of how it can be controlled. This is because the model contains references to packages that encapsulate the control channels offered to a potential neural control architecture. From such a description, it is close to impossible to restore a mapping of joints to robot topics, as this information is buried in controllers, only available in compiled form.

In the domain of neuroscience, the situation is very different. Here, an important question is the level of abstraction, a neuron is interpreted. While some approaches investigate the connection of entire regions of a neuronal network consisting each of hundreds or thousands of neurons, other approaches investigate the compartments inside a single neuron. Existing formal approaches to model neuronal networks such as NineML [9] or NeuroML [10] try to combine these diverse levels of abstraction and provide a uniform format to catalog knowledge. However, this makes them unusable for simulation purposes as there is no simulator that spans all these levels of abstraction. Most simulators we have been facing in the development of the NRP simulate neural networks at the level of point-neurons, meaning that each neuron is considered to be an opaque box whose behavior is described by a set of differential equations. These neuron models can be modeled in a formal way [11].

When it comes to simulating neural networks, the common format to pass in a model of a neural network to a simulator is a Python script that creates the model inside the simulator. To raise the compatibility between the simulators, there is an abstraction layer PyNN [12] which provides an interface that can be used to create the neural network independently of the used simulator, such that network scripts can be used with multiple simulators with only few modifications.

This very low degree of formalization implies some challenges as it makes it very hard to inspect neural networks without loading them into a simulator. This applies even to very simple analyses such as checking whether a given neuron or population exists at all. One of the reasons is that currently there are many neural networks that use a multitude of control structures and loops when creating the network. Hence, any formal model that does not allow such control structures has a risk of a low acceptance. If control structures are supported, there is a risk to end up with a formal metamodel that does not add much to the Python syntax. Furthermore, because Python scripts are so popular, this raises expectations that any new approach also supports them. As a result and to avoid an enforced language adoption [13], we created an internal DSL in Python [5], allowing to specify the coupling between a neural network and a robot in Python, but without the requirement that a simulator is already loaded. A formal metamodel on top of this internal DSL has not yet gained acceptance. We hope this will change when we can provide a graphical editor for it.

These very different understandings of models imply a communication challenge when bringing together experts of different fields. While for some in the team, the term model implicitly means the robots mesh, the term is bound to Python scripts or neuron models when speaking to others. Especially in the neuroscience domain, the level of formalization is currently very low, which makes it very hard to establish formal modeling standards.

As a possible reason, both in robotics and in neuroscience, models are often validated through simulation or even execution. A model is considered valid if its execution completes without failures and produces the intended behavior. However, it is difficult to specify when such a behavior is valid. This is different for the coupled simulation of both a robot and a neural network where one would like to ensure that for a given connection between the simulators, both the involved neurons or neuron populations as well as the involved sensors and actuators exist and have the expected format.

4.2 Focus the Platform, Not the Generator

Following a model-driven approach as depicted in Fig. 4, a very important question is whether a given functionality should be implemented in the target platform or the generator, if both are developed in the same team. Applying the approach of Völter and Stahl [7], we started with creating a reference simulation script after creating an initial version of the platform. We then extracted a code generator that would generate exactly this simulation script based on a given reference model.

However, this soon turned out to cause problems in the quality assurance. Generators are very hard to test in terms of unit testing. While an integration test is desirable, the resources necessary to run these makes it infeasible to cover large parts of the code through integration tests and make unit tests inevitably required, but ensuring that the produced simulation scripts are correct is complex and far beyond a usual unit test. Furthermore, we lack the tools to measure whether the code generator complied to our goal of 90 % coverage by unit tests.

Therefore, we only tested the generated output for a few example models by comparing with a predefined expectation and further created usual unit tests for any functionality called from the generator. However, this leads to highly fragile unit tests. Many changes in the target platform had an influence onto the code generator and as a result, the test cases had to be adjusted. Thus, developers started to simply copying all the generated output for the changed generator to the test oracle and peer-reviewers spent less attention as the code was only generated.

Therefore, we eventually decided to minimize the amount of code that is being generated and tried to drag as much code as we could into the target platform as Python modules. As a result, this code is subject to our continuous integration infrastructure and thus, many code checks are automatically performed by static code analysis that would otherwise have to be done by tests. The generator now only generates artifacts that are very hard to create by non-generating approaches such as expressions composed from model elements. While this could also be done through model interpretation at runtime, it would mean a less clear syntax (which is made visible to the user and therefore important) and degrade the performance.

An artifact that has helped a lot minimizing the generated code is our internal Python DSL PyTF [5]. Despite a syntax familiar to Python developers (which is why we created the language in the first place), it also helped us minimize the code generator transforming our formal model into PyTF. PyTF itself can then be executed directly.

4.3 Model-Driven Tooling Based on Java Platform

As of today, many of the tools available to support model-driven engineering are still based on Java, more specifically on the Eclipse Modelling Framework. The components that we used in the NRP are based on C++ and Python. As we also decided to create a Python interface for users to specify their connection between selected neural networks and robots, the simulation backend is also based on Python to avoid inter-process communication.

However, this decision puts a platform barrier that hampers the adoption of model-driven techniques since most available tools cannot be used, unless the involved developers install a Java IDE next to their Python or C++ environment. As suggested by Meyerovich [13], many developers do not like the idea of adopting a new language which yields a strong argument against the introduction of such tools. Indeed, our experience was that many developers tried to avoid Java as much as they could.

Therefore, the more pragmatic solution for us was to use XML technology. That is, we created the formal metamodel as an XML Schema. Though this has the drawback that XML has no direct support for typed cross-references (only ID and IDREF), this can be circumvented by designing the metamodel appropriately. In favor of XML, like most languages, Python includes tools to generate parsers based on a XML schema so that we can easily load and save models. Furthermore, XML Schema allows a basic validation of instance documents.

This validation is not as powerful as OCL and harder to specify but suffices for many applications. More advanced validation, such as checks whether a reference to a particular neuron or robot sensor is valid, has to be done separately anyhow since the data is not available in a formal model (cf. Sect. 4.1).

A promising approach was to use EMF tools to generate the XML Schema from an Ecore metamodel, but this turned out to be not a long-term solution. EMF by far does not export all validations done on models into the XML Schema. As we tend to include as much validation as we can, this quickly meant that we maintained the XML Schema manually. However, as long as the Schema complies to the XMI standard, the interoperability to modeling frameworks such as EMF is still given as these frameworks also use XMI.

This interoperability is important, as it allows to use editor technologies such as e.g. XText[3] or Sirius[4] and consume the created models in other languages. However, in our case, this is difficult as we need our editors to be web-based.

To generate code, we used Jinja2[5], normally used to generate HTML pages, though we generate Python code. This allows to use template-based code generation without additional effort. Nevertheless, these templates do not comply with static code analysis and as a result, the acceptance of these code generators among the developers is low. As a result, we have reduced the code generation to a minimum, meaning that we do not generate anything else than our Python DSL.

4.4 Customer Value of Model-Driven Artifacts in a Scrum Process

Though set up as a research project, the NRP is developed according to a distributed Scrum process in sprints of three weeks length. Model-driven software development introduces a high initial effort to set up the metamodel, transformations and editors. This seemingly contradicts the idea of Scrum where all items of the backlog should be user stories that add some value to the user.

On the other hand, it is the user that specifies models of a simulation in the final platform. Therefore, the typical development artifacts of a classical model-driven project are in fact parts of the ready-made platform and therefore do add a value to the user, hence fit into the format of a user story. For example, metamodels are created with a user story similar to "As a user, I want to have a clear specification how x is defined". This gives the developer one sprint to create an initial metamodel and possibly generate a visualization of it to show that to the (expert) user. Further documentation of the metamodel, accessible also to the non-expert user as well as other artifacts such as a DSL on top, generators or editors are then developed in subsequent sprints.

Slightly more problematic are evolution scenarios as there is a risk that new features are not introduced into all artifacts simultaneously. Therefore, the metamodel may diverge from subsequent artifacts such as the DSL. As there are few

[3] https://eclipse.org/Xtext/.
[4] http://www.eclipse.org/sirius/.
[5] http://jinja.pocoo.org/.

tools to detect this in a dynamic language such as Python, we rely on code-reviews. Though such evolution scenarios appear more often in agile methods, this problem is not limited to them as it is unlikely to have an optimal metamodel at the first attempt.

Overall, we think that the agile Scrum methodology met our needs creating a model-driven platform very well.

4.5 Missing Baseline for MDE Benefits

Although we already noted the multitude of potential benefits brought by MDE in Sect. 3, quite a number of people in the project are still skeptical on MDE. This is partially due to problems we discussed in the earlier sections, but also because the benefits are not obvious. We see having a formal representation of domain concepts used in the NRP as the key benefit. But as we are developing both the formal metamodel and the target platform on which it is executed in parallel and in the same team, it is hard to distinguish whether the metamodel has influenced the platform development or vice versa.

As a consequence, it is easy to claim that the formal metamodel just formalizes the concepts implemented in the platform anyway. After all, abstractions can also be employed without MDE techniques in place. It is hard to proof this wrong as the usage of proper abstractions does not strictly require MDE. We believe that even if some of the models are supplied very informally, attempts to formalize these models still improved our domain understanding and helped us to ask domain experts the right questions. However, further research is required to analyze and show potential advantages and disadvantages.

5 Conclusion

In this paper, we reported our experience applying MDE in the development of the Neurorobotics Platform, a web-based simulation platform to run experiments coupling spiking neural networks with robots. Though we identified large potential given the overlap of defining and running such an experiment on the one side and model-driven software development on the other, there are a couple of challenges and obstacles that make the application difficult.

We expect that our lessons learned of these challenges can help others who want to apply model-driven techniques in the area of neurorobotics and help to identify obstacles in the future development of the model-driven approach as a whole. An inconsistent understanding of models means that few assumptions can be made on how a model is specified and many models are only available rather informal. Especially in neuroscience, people are used to Python, making it infeasible to use model-driven tools, often based on Java. As the support for validating generated Python code is limited, we limited code generation to the places where it is absolutely necessary. The compatibility with the Scrum process we are following could be solved. However, the overall benefit of MDE was not as clear for other developers, making arguments in favor of it difficult.

Acknowledgment. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreements no. 604102 (Human Brain Project) and 610711 (Cactos).

References

1. Hines, M.L., Carnevale, N.T.: The NEURON simulation environment. Neural Comput. **9**(6), 1179–1209 (1997)
2. Gewaltig, M.-O., Diesmann, M.: NEST (NEural Simulation Tool). Scholarpedia **2**(4), 1430 (2007)
3. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an opensource multi-robot simulator. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), vol. 3, pp. 2149–2154. IEEE (2004)
4. Braitenberg, V.: Vehicles: Experiments in Synthetic Psychology. MIT press, Cambridge (1986)
5. Hinkel, G., Groenda, H., Vannucci, L. et al.: A domain-specific language (DSL) for integrating neuronal networks in robot control. In: Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering (2015)
6. Quigley, M., Conley, K., Gerkey, B. et al.: ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software, vol. 3, p. 5 (2009)
7. Völter, M., Stahl, T.: Model-Driven Software Development. Wiley, New York (2006)
8. Stachowiak, H.: Allgemeine Modelltheorie. Springer, Heidelberg (1973)
9. Raikov, I., Cannon, R., Clewley, R., et al.: NineML: the network interchange for neuroscience modeling language. BMC Neurosci. **12**(Suppl 1), 330 (2011)
10. Gleeson, P., Crook, S., Cannon, R.C., et al.: NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. PLoS Comput. Biol. **6**(6), e1000815 (2010)
11. Plotnikov, D., Blundell, I., Ippen, T. et al.: NESTML: a modeling language for spiking neurons. In: Modellierung (2016, to appear)
12. Davison, A.P., Brüderle, D., Eppler, J.M., et al.: PyNN: a common interface for neuronal network simulators. Front. Neuroinformatics **2**(11), 1–10 (2009)
13. Meyerovich L.A., Rabkin, A.S.: Empirical analysis of programming language adoption. In: Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, pp. 1–18. ACM (2013)