Volr: A declarative interface language for neural computation

Experimental neural systems modeling with Jupyter Notebooks

Christian Pehle
Kirchoff-Institute for Physics
University of Heidelberg

Jens Egholm Pedersen
Department of Computer Science
University of Copenhagen

Contact Information:
Universitetsparken 5
2100 København Ø

+45 25122752
xtp778@alumni.ku.dk

Abstract

For inexperienced users, there are significant barriers to entry to the computational platforms and tools developed by the Human Brain Project (HBP). Despite the efforts done to adopt common abstractions and interface libraries like PyNN, it is time-consuming and complicated to construct and simulate spiking neural networks. Even more so if the experiment involves learning. This poster addresses the growing need for consistent and reproducible semantics within simulated and accelerated neural experiments. We present an intermediate compiler, that bridges the inconsistencies of the execution environments by offering a stable abstraction, along with a machine learning library for spiking neural networks. The library and compiler currently supports NEST, BrainScaleS as well as regular artificial neural network models.

The work presented here carries the potential to significantly reduce experiment iteration times and drastically cut costs, to the benefit of the research community as a whole. Future plans are to develop a strongly typed calculus for network connectivity and plasticity, paving the way for high level abstractions for learning, including the learning-to-learn paradigm. Additionally, we aim to support SpiNNaker and the second generation BrainScaleS hardware platform.

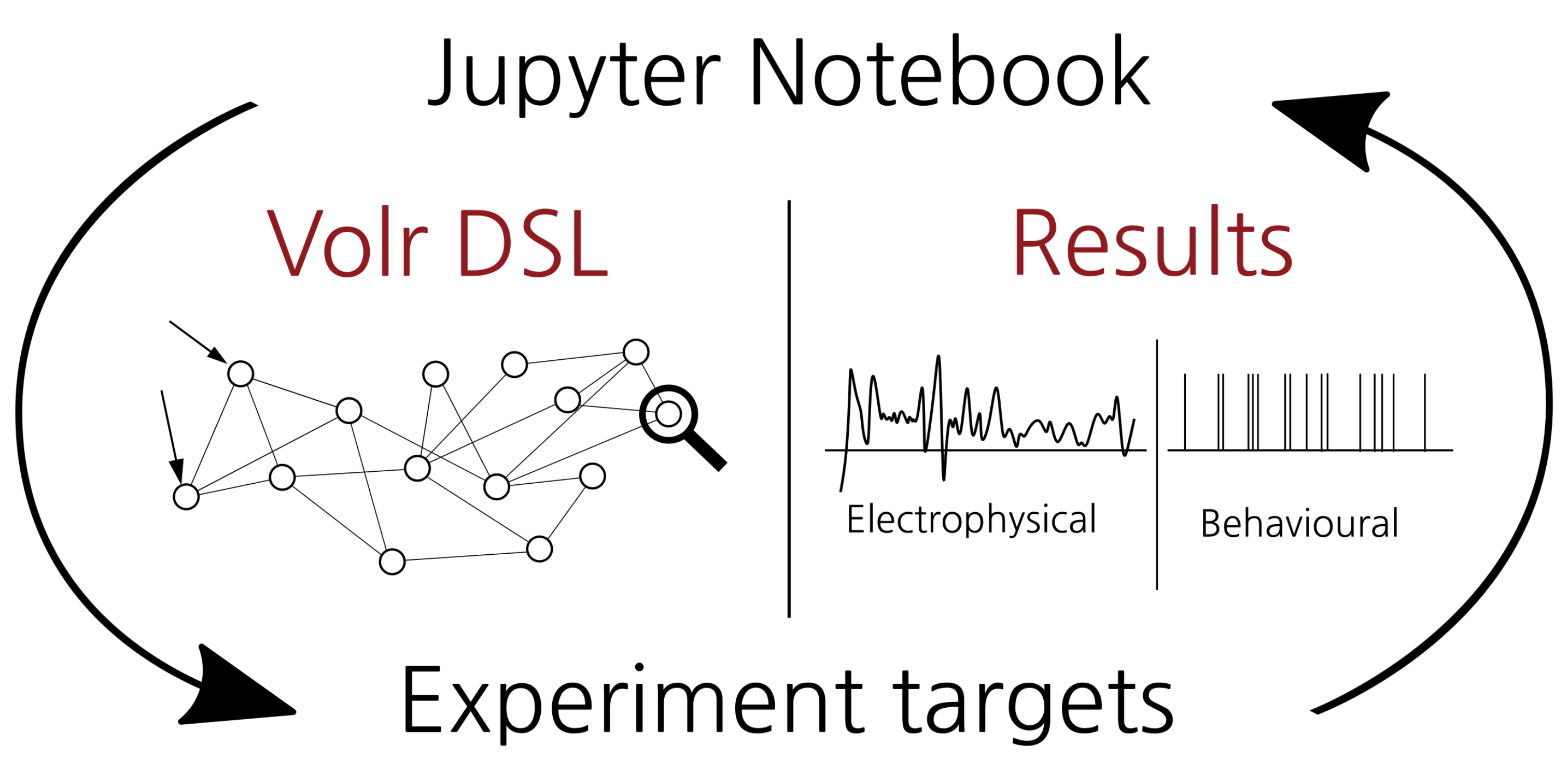

Figure 1: The workflow of an experiment, from the modeling in a Jupyter notebook, through the evaluation and training on one or more platforms, to the retrieval of experiment results for subsequent analysis. From the perspective of the user everything happens directly in the notebook.

Introduction

Despite the pioneering work of simulator APIs for neural experiments, inexperienced users are faced with considerable obstacles when experimenting with the HBP-related computational platforms. Running experiments includes learning an interface language like PyNN, working with simulation-dependent peculiarities, digitising analog experiments and performing complicated pre- and post-processing of large amounts of data.

A compiler built on a strongly-typed network abstractions has been built to unify the heterogeneous execution environments towards the user. To interface the compiler, a domain-specific language (DSL) has been developed, that provides stable and testable semantics. The generated networks are trained in a Tensorflow-like learning framework. Because of the support for multiple backends, model parameters can be copied into other models, allowing non-plastic platforms like BrainScaleS to indirectly benefit from the training. Finally, a Jupyter Notebook environment is provided to hide the irrelevant compiler details for the user, while simplifying the compilation and execution of experiments. The project accelerates the workflow when simulating neural networks, thereby increasing the research output and widening the audience for neural simulators.

The prototype is designed to facilitate the entire life cycle of an experiment: generation/preprocessing of data, neural network topology and properties, experimental setup and finally data extraction and post-processing (see Figure 1).

Documentation is available at: `https://volr.readthedocs.io`

1. Execution targets

The current prototype supports three simulation targets: NEST (through the NEST API), BrainScaleS for spiking neural networks (SNN) (through PyNN) and artificial neural networks (ANN) with GPU acceleration (through Futhark [4]).

The DSL maintains an intermediate representation of the experiments, including all the necessary details to reproduce the setup. The intermediate representation simplifies the integration with execution targets, currently by through the efforts of Futhark, PyNN and the NEST API (see Figure 3). However other targets would be interesting to examine, such as SpiNNaker, TensorFlow, Intel's Loihi chip and the next generation BrainScaleS 2 system from Heidelberg.

a) Compile and train a model

```
# Compile the code
dsl = 'Net 2 4'
model = Volr.compile(dsl)
# Train the model
results = model.train(x, y)

```

b) Inject trained weights

```
# Extract model weights
params = model.parameters
# Inject into new model
fitted_model =
  Volr.compile(dsl, params)
```

Figure 2: Examples for a) constructing and training a Volr model and b) extracting model parameters and injecting them into a new, fitted model.

2. Fast experiment feedback loop with Jupyter Notebooks

Jupyter Notebooks are increasingly popular in data processing and data analysis domains. Exploiting this paradigm lets researchers quickly map hypotheses to experiments, with the support of the vast Python ecosystem.

After installing the environment it takes less than a minute to setup an experiment and extract the first results. Figure 2 illustrates how a model can be built and trained. By building on open technologies like Python and Jupyter Notebooks, the experiments are straight-forward to share and expand upon.

3. A framework for learning

A machine learning library has been developed to train the networks in NEST and Futhark. Because of the platform-independent topologies, trained parameters can be transferred from one model to another (see Figure 2.

At the core of project is the domain-specific language (DSL) Volr, that models neural network topology as well as simulator-specific properties. The inner representational structure allows for abstract analyses, by understanding networks as first-class citizens subject to function composition and lambda calculus. These *algebraic* properties could pave the way for the application of techniques such as reverse differentiation, dependent types and semantic error checking.

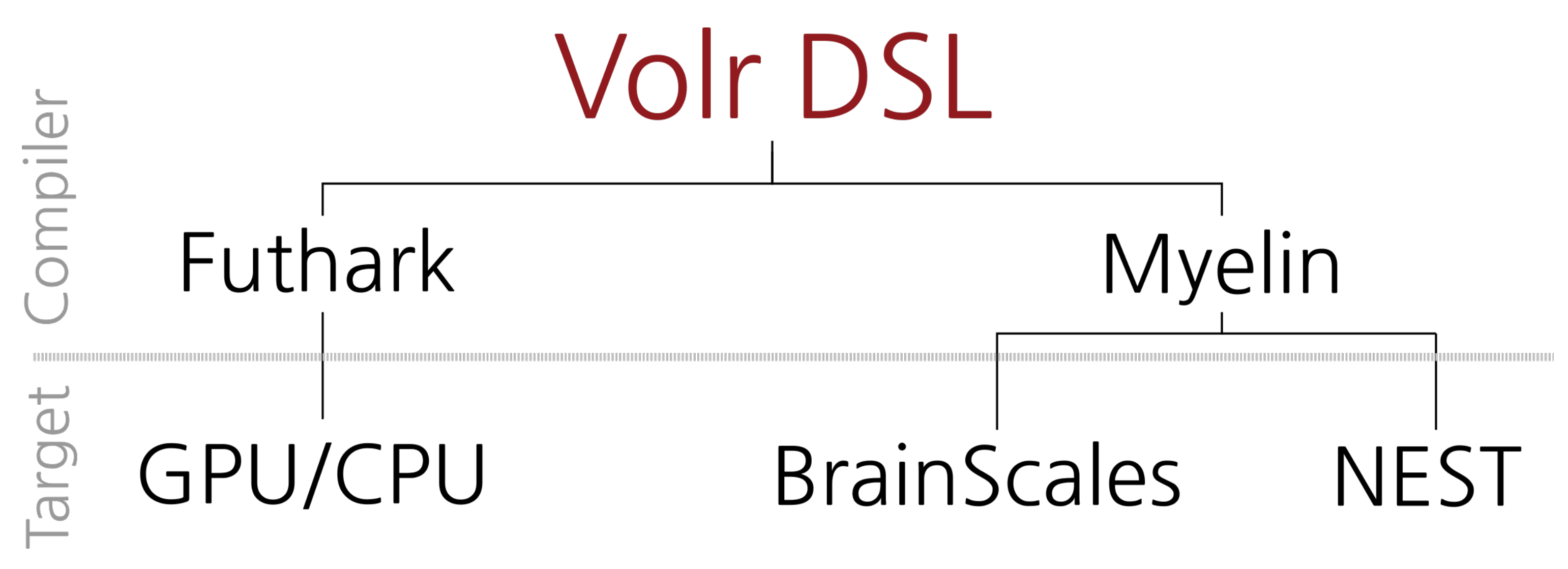

Figure 3: The translation from the DSL to the execution targets. ANN models are interpreted by Futhark which in turn is evaluated on GPU/CPUs. SNN models are interpreted by the internal Myelin compiler that checks for inconsistencies and generates target-specific code through PyNN and the NEST API.

4. Future work

Ongoing work is exploring the domain of more complex cognitive experiments with cognitive tasks modeling, reverse differentiation for ANN and the inclusion of high-level abstractions for learning through learning-to-learn integration between ANN and SNN. In this context it would be interesting to incorporate the methods of [2][5] into a generalizable non-experiment specific framework.

The algebraic properties of the internal model also deserves to be explored. Techniques such as automatic differentiation could permit the fast generation of large and complex SNNs [1].

Conclusions

1. Prototypical integration with Jupyter Notebooks has been shown to allow fast iterations of experiments, with immediate access to already familiar Python tools for large-scale data analysis.
2. The formalised domain-specific language, Volr, has been developed to describe reproducible and consistent neural network experiments for artificial and spiking substrates.
3. Three targets are currently supported: NEST, BrainScales and artificial neural networks through Futhark. Further work is needed to expand existing targets and include platforms such as Intel's Loihi, SpiNNaker and the new DLS chip from Heidelberg, Germany.
4. Ongoing work is improve and exploring the domain of more complex cognitive experiments, including *learning-to-learn*, cognitive tasks modeling and reverse differentiation for ANN.

References

[1] A. Baydin, B. Pearlmutter, and A. Radul. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015.

[2] G. Bellec, D. Salaj, A. Subramoney, R. A. Legenstein, and W. Maass. Long short-term memory and learning-to-learn in networks of spiking neurons. *CoRR*, abs/1803.09574, 2018.

[3] J. Eppler. A Python interface to NEST. *The Neuromorphic Engineer*, 2009.

[4] T. Henriksen, N. Serup, M. Elsman, F. Henglein, and C. Oancea. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. *Proceedings of the 38th ACM SIGPLAN*, pages 556–571, 2017.

[5] M. Innes, D. Barber, T. Besard, J. Bradbury, V. Vhuravy, S. Danisch, A. Edelman, S. Karpinski, J. Malmaud, J. Revels, V. Shash, P. Stenetorp, and D. Yuret. On machine learning and programming languages. `https://julialang.org/blog/2017/12/ml&pl`, 2017.

Acknowledgements

We are grateful for the valuable time and counsel of Sebastian Schmitt and Karlheinz Meier from the Kirchoff-Institute for Physics at the University of Heidelberg, Germany.

We would like to thank Jesper Mogensen and Nicolai Daugaard from the Unit for Cognitive Neuroscience and Martin Elsman from the Department of Computer Science at the University of Copenhagen for their interest and important contributions.