RWTH Aachen University
Software Engineering Group

Extension of the NEST Modelling Language to the SpiNNaker Architecture

Master Thesis

presented by
Schmidt, Levin

1st Examiner: Prof. Dr. A. Morrison

2nd Examiner: Prof. Dr. B. Rumpe

Advisors: Charl Linssen, Pooja Babu

The present work was submitted to the Chair of Software Engineering

Aachen, September 10, 2023

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Kurzfassung

Die Modellierungssprache NESTML (NEST Modelling Language) wurde als simple Sprache zum Modellieren von Neuronen und Synapsen entwickelt. Sie ermöglicht das generieren von performantem Programmcode für unterschiedliche Endpunkte, mit einem Hauptfocus auf NEST (Neural Simulation Tool) Simulator . NEST ist ein simulator für gepulste neuronale Netze welche in den Neurowissenschaften Anwendung finden um das Verhalten unterschiedlicher Netzwerke und Modelle zu beobachten. Diese Arbeit beschäftigt sich mit der Erweiterung von NESTML auf die neuromorphe Hardwarearchitektur SpiNNaker (Spiking Neural Network Architecture). Es wird eine Übersicht über die Konzepte welche für SpiNNaker und NESTML verwendet wurden gegeben. Darauf folgt eine Analyse und Validierung mit dem Vergleich zum NEST Simulator.

Abstract

The NEST Modeling Language (NESTML) was developed to be an easy-to-use modeling language for neuron and synapse models. It allows for generating performant code aimed at different target APIs, with the main focus on the Neural Simulation Tool (NEST) simulator. NEST is a simulator for spiking neural networks used in neuroscience to explore the behavior of different networks and models. This thesis explores the extension process for NESTML to the neuromorphic hardware design SpiNNaker (Spiking Neural Network Architecture). It will give an overview on the concepts applied in SpiNNaker and NESTML, followed by the analysis and validation of the extension process, by comparing it to the NEST simulator.

Contents

Chapter 1

Introduction

Despite all efforts, the brain remains still not fully understood. The first research regarding the brain can be dated back to ancient Egypt in 1700 B.C. [vMSB10] studying brain-related injuries and to the philosophers of old Greece, first suggesting that the brain is the center of intelligence instead of the heart [CR07]. Today, we know that the brain is a complex network of different cells, with the neurons alone amounting to 86 billion [ACG+09]. Still, we are missing some of the details of how intelligence emerges from this network. Before the invention of the computer, experiments were carried out on living subjects (in vivo) or in a dish (in vitro), both still relevant today. However, computational neuroscience offers some advantages to traditional ways. Experimenting on the living brain is difficult because of its fragility, associated costs, and ethical conflicts regarding animal welfare [ASP22]. Furthermore, these experiments suffer from the high variability of the environment and the differences in each subject. Observing the state of the brain is also problematic because of the missing control and the scale. High-resolution imaging enables recording a few cells in detail or the general activity of larger brain areas but is then missing the resolution to record the detailed states of the cells[Ker08].
For a more exact analysis of the cell structure, we can rely on in vitro experiments. However, it was shown that the functionality of the brain is not encoded in a single cell but in the network and its activity, therefore limiting the insights we can gain from these experiments.
With the rise of computers also emerged the field of computational neuroscience. Experiments in this area often rely on simulations that can model the behavior of the cells and the network. These simulations offer control over the network parameters, data and cell states, allowing a closer study of the network compared to in vivo experiments and reproducibility. Simulating these highly parallel networks demands efficient simulator solutions. Software simulators like NEST [GD07] run on general-purpose processing units, with NEST supporting everything from MacBooks up to supercomputers. NEST simulates Spiking Neural Networks (SNN) from the network perspective, not focusing on replicating the exact behavior of the neuron but instead giving insights into the network. [wwwc] Other solutions use neuromorphic hardware that resembles the structure of the brain for a more efficient simulation (e.g. SpiNNaker [FB20]). While the development of SpiNNaker was also focused on SNNs, it can also be used for other problems that benefit from its highly parallel design. The neuromorphic hardware is built from a network of microprocessors executing individual nodes, which, in the case of SNNs, are the neuron and synapse models. Similar to NEST, SpiNNaker is focused on simulating the network behavior. SpiNNaker integrates with PyNN [DBE+09], a Python [VRD09] library for

SNNs, which provides a universal interface to describe networks for supported simulators while the implementation of the network and models (neuron and synapse) is handled by the simulator developers.
Either of these simulators requires new neuron and synapse models to be implemented in the target simulators programming language. This makes it necessary for the neuroscientist to also be proficient in computer science to produce correct and efficient models. For this reason, modeling languages were developed that act as an interface between the neuroscientist and computer scientist (e.g. NeuroML[CCGC$^+$14], NESTML[LBB$^+$23]). They feature a simple syntax that follows known neuroscience concepts to simplify defining new models while the computer scientist handles the implementation. An example is the NEST Modeling Language (NESTML [LBB$^+$23]), which was originally developed for the NEST simulator. Its syntax features concepts similar to Python, a programming language already widely used in neuroscience [MBD$^+$15]. Using NESTML also is interesting for other simulators, as it is not bound to NEST. Its software architecture allows for defining additional target simulators. Because of the advantages of neuromorphic hardware over software-based solutions regarding speed and efficiency, we chose SpiNNaker as an interesting target. This leads to the research question of this paper:

Research Question How can NESTML be extended to the SpiNNaker neuromorphic architecture?

To answer this, we identify an extension process that provides insights into how new targets can be chosen, which components are essential for a new implementation and how the implementation can be verified. We describe how each of these steps was applied for SpiNNaker to illustrate methods and goals for each. We start by providing the fundamentals necessary for the project in chapter 2, to have then a deeper look into the concepts of NESTML (chapter 2.3) and SpiNNaker (chapter 2.5). In chapter 3, we present the extension process and the validation results. At the end, we'll discuss the results of the thesis in chapter 4 and close with a recap of the presented information in .

Chapter 2

Fundamentals

2.1 Domain Knowledge: Neuroscience

To get an understanding of the brain, even though just superficial, the sections in this chapter will briefly introduce the cells and activities in the brain that make up its functionality. This overview is by no means complete, leaving out cells like the glia and biochemical specifics. Even the cells covered here are generalizations of what is found in the brain, as there are many different variations and small-scale behaviors that would go over the top. What is presented in this chapter should provide essential knowledge to understand the abstractions used in the design of NESTML in chapter 2.3 and the similarities to concepts of neuromorphic computing (chapter 2.4). For a deeper look into the brain's structure, see "The Brain" by Watson et al.[WKP11].

2.1.1 Neuron

A neuron is a special type of cell found in the brain. Like in other cells, its body, the soma, contains the cell's genetic information. The special thing about neurons are the extensions from the soma, which establish connections to other neurons. These are distinguished into two categories. Dendrites form dendritic spines, to which an axon (the second type) can connect.[1] The membrane of the dendritic spikes features selective ion channels controlled by receptors. These can be externally activated, leading to a rise or decrease in the membrane potential with regard to the extern potential(see chapter 2.1.2). Surpassing a threshold in the potential leads to a short, significantly larger rise in potential, called an action potential or spike, propagating through the whole neuron, but especially through the axon. At the end, the axon splits into the axon terminals, where the action potential activates the release of chemicals called neurotransmitters. These are the basis for the communication between the neurons controlled by the synapse. [WKP11]

[1] The current state of research proved the brain to be more complex than this, showing that dendrodendritic [RSRB66], axoaxonic [Gra59] and axosomatic[WC82] synapses also exist. For simplicity, this notion is omitted in this introduction

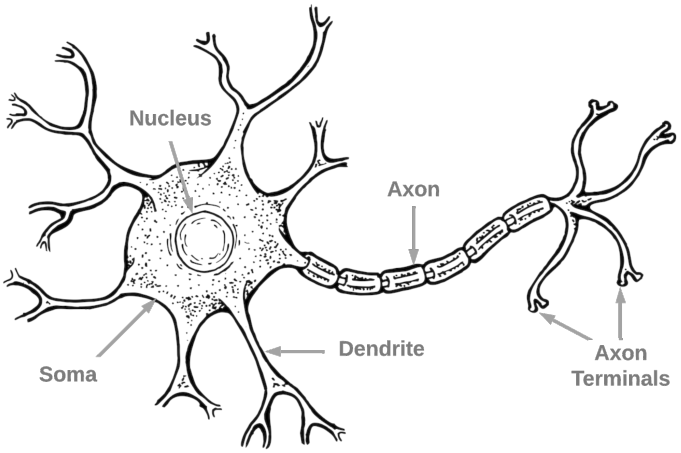

Figure 2.1: Conceptual depiction of a neuron

2.1.2 Synapse

Unlike neurons, synapses are not cells but an umbrella term to describe the processes observed at the connection of two neurons. Connections between neurons are directional, with the emitting neuron referred to as the pre-synaptic neuron and the receiving neuron as post-synaptic. The neurotransmitters released in the pre-synaptic neuron are the communication medium. The axon terminal contains synaptic vesicles that enclose the neurotransmitters and allow them to be released to the outside of the cell by a process called exocytosis [Zuc96]. The dendritic spines of the post-synaptic neuron are spatially close to the axon terminal but form a small gap, the synaptic cleft. The released neurotransmitters from the axon terminal travel to the membrane of the dendritic spine, briefly docking to receptors on the membrane before they are broken down by enzymes in the synaptic cleft. This happens on a timescale of milliseconds. The activation of the receptors leads to the opening of one type of ionic channel, which either increase (sodium) or decrease (chloride) the membrane potential. According to Dale's law, only one type of neurotransmitter is released from the pre-synaptic neuron, opening the same ion channel.[EFK54] Which type of connection is formed underlies a dynamic process called plasticity. [WKP11]

2.1.3 Plasticity

Plasticity describes the dynamic process in the brain that enables the structuring of the neural network and the formation of memories by altering existing synapses on different time scales. Changes to the synapse alter the strength of the connection, meaning that the influence of the pre-synaptic neuron on the membrane potential of the post-synaptic neuron is increased (potentiation) or decreased (depression) by emitting more or fewer neurotransmitters. This acts on different time scales due to different underlying mechanisms, categorized into short-term plasticity (up to a few minutes) and long-term plasticity for long-lasting effects. [CM08]

The mechanisms underlying short-term plasticity are thought to come from the effects of temporally close action potentials. Remaining neurotransmitters in the synaptic cleft and

chemicals in the axon terminal can lead to a higher concentration of neurotransmitters, increasing the amount of ions absorbed by the post-synaptic neuron and, therefore, a bigger effect on the membrane potential. On the other hand, previous activation can lead to a depletion of neurotransmitters in the axon terminal, reducing the effect on the post-synaptic neuron. [CM08]

Long-term plasticity is based on the proposal of Hebbian learning [Heb05]. According to this rule, a connection is potentiated when the action potential generated in the pre-synaptic neuron leads to a temporally close action potential in the post-synaptic neuron. This rule is extended to Spike-Timing Dependent Plasticity (STDP) to allow depression when the post-synaptic neuron is active independent of the pre-synaptic neuron. Additionally, the time between the pre- and post-spike is relevant, with smaller differences leading to a higher impact on the connection strength. [CM08]

2.1.4 Computational Neuroscience

In computational neuroscience, the structure of the brain is often abstracted in the form of Spiking Neural Networks (SNN). In contrast to Artificial Neural Networks (ANN) more common in computer science, they follow biology more closely to get insights into how the functionality of the brain emerges. The basis of a SNN is a directional graph, where vertices represent neurons and edges the synapse. Action potentials are modeled as spike events routed from the pre-synaptic neuron through the synapse to the post-synaptic neuron. The connection strength is abstracted in the form of weight, describing the spike's impact on the post-synaptic neuron that the synapse adds when routing the spike. The connection strength is evolved dynamically, for example, by STDP update rules.
A common abstraction for the neuron is the point neuron, which combines the biophysical structure and behavior of the dendrites and axons into a single mathematical formulation. Alternatively, the neuron can be modeled in compartments, for example, to extract the dendrite behavior from the point neuron. This allows connections to use different dendritic behavior, a mechanism observed in the real neuron.[GFCA21]
Execution of the SNN relies on simulation strategies. These describe how the dynamics of the models are evolved. In a time-based or synchronous simulation, the models are executed periodically, while event-driven (asynchronous) simulations execute the model based on a received event. Combining both is also possible and is referred to as hybrid simulation. An example is described in [MMG$^+$05], which is aimed at the requirements of spiking neural networks. Many synapse models only update their state on pre-synaptic spike events, which are infrequent, so a time-driven update scheme would be inefficient, especially considering the number of synapses in neural networks. A fully event-driven approach is also not efficient because neurons potentially receive a high number of spike events. Furthermore, this simulation method is incompatible with some concepts used in neurons, like a continuous post-synaptic current. For these reasons, the authors of [MMG$^+$05] propose to use time-driven simulation for neuron models while using an event-driven strategy for synapse models.[MMG$^+$05]

2.2 Domain Specific Language

General Purpose Languages (GPL), for example C++ [Str00] and Python [VRD09], are designed to be generally applicable to any problem, not offering syntactic concepts that directly relate to the problem. Starting with these languages is a hard task that takes some time to get familiar with. This includes learning their grammar, but also how to use the provided syntax like control-flow statements. Since the value of computers is recognized in many research domains, this became a problem for the researchers. Either a decent knowledge of programming is necessary or they have to rely on people outside their domain to implement a solution to their problem. Learning programming hinders progress, as apart from learning, the functionality and correctness of the program need to be verified. When relying on programmers outside the domain, the communication of the problem is a hindrance and the turnaround time for the code is longer. These problems led to the development of Domain Specific Languages (DSL) that provide domain-specific syntax and limited expressiveness, simplifying the process of learning the language for the domain expert.
In the design process of a DSL, the programmer closely works together with domain experts to design the syntax around notations and concepts already used. This domain analysis ensures that the DSL is applicable in the domain and the learning process is simplified. The result is a grammar that combines all identified syntax expressions into rules describing valid DSL scripts. This also allows to restrict the expressiveness of the DSL when for example loops are not included in that grammar. This may mean that the DSL is not Turing-complete.
Furthermore, the programming paradigm is defined through the grammar. The imperative paradigm is often found in GPLs, where the logic of the program is laid out line-by-line, allowing for simpler optimization but hiding the intent of the program. An alternative is the declarative paradigm, where the programmer declares the intended outcome while the actual implementation is hidden. This approach is less error-prone and the intended result of the program is simpler to extract, but optimization is more difficult.
Converting the DSL script into a computer-readable representation is necessary for the script to be executed. Because of the custom grammar, this can't be done with tools already existing for other languages, but implementing this custom for the DSL is a complex task. Instead, a toolchain is implemented that translates the DSL script into valid code in a target GPL. Three types can be distinguished based on how the DSL is integrated into the target GPL. [Fow10][p.27]

Internal DSL An internal DSL uses a subset of the target GPL's syntax, limiting the grammar to valid expressions in the GPL. The implementation is provided as an Application Programming Interface (API), which limits the expressiveness and can also be designed to feature method-chaining to achieve a declarative programming style.[Fow10][p.60]

External DSL The grammar of the external DSL can be designed freely without considering the syntax of the base language. DSL scripts are structured text files where either existing structured text (e.g. XML, JSON) or a custom structure can be used. The latter leads to more development overhead than existing solutions, as these already provide tools to parse the text. However, custom solutions may be better suited to the domain. [Fow10]

Work-bench DSL Language workbenches are a framework in which the DSL can be created. They support the development process by providing additional tools for the development, like graphical design interfaces. An example for this is MontiCore [RHK21]. [Fow10]

2.2.1 Toolchain

Some common tools need to be implemented for the DSL, which will be presented in this chapter(also see Fig.2.2). The starting point is the grammar, which describes how a valid script is structured. Based on this the lexer and parser are defined.
The lexer is a tool to convert the strings in the input script into tokens. These tokens are containers with two attributes, the content and the type. While the content is the string it contains, the type identifies the syntactic component that is represented by the string. This way, the following syntactic analysis done by the parser is independent of the naming of the components.
The parser takes the tokenized program as input and constructs a tree that represents the relations between the tokens based on the grammar, called the Abstract Syntax Tree (AST). Each expression in the program is represented by an abstract model in the tree, with the program as the root. The children of the expression nodes are the defined subexpressions, for example a function call. Terminal nodes of the tree are literals like datatypes and operators. To support the building of the syntax tree, a symbol table is necessary to reference expressions that are not in the same subtree, e.g. variable definitions. When an identifier is parsed, it is added to this table under that name. From here on, when mentioning parsing we also imply that lexing takes place because of the close relation of these processes.[Fow10]
Because the syntax of the DSL represents domain-specific concepts, it is a kind of abstract model of the domain. To focus this even more, the DSL can implement a semantic model generated from the AST and designed with abstract objects of the domain. This way, the information can be separated from the syntax and better accessible during execution.
How the DSL program is executed can differ between the types of DSLs. Internal DSLs integrate with the program of the host GPL so the is executed at runtime. This can also be achieved with external and work-bench DSLs when the toolchain is compiled into a standalone program that takes the DSL script as input and executes it in the same runtime environment. A similar method is found in GPLs known as interpreting, the toolchain is then referred to as an interpreter. This implies that the DSL can only be translated to the GPL the toolchain is implemented in.
An alternative to this is code generation, where the DSL script is translated into a valid program for the target GPL and executed independently from the toolchain. This process resembles compiling used in different GPLs. The advantage of this is, that multiple target GPLs can be supported without reimplementing the tools, and the generated program is free of the additional overhead of the toolchain. However, another tool needs to be added to the toolchain to support this, namely the code generator, which translates the internal representation into a valid program in the target GPL.
Code generation can either be based on templates or rules. Templates are a combination of static code in the target GPL and template expressions which are executed by the templating engine. This generates a string output based on the internal representation which is injected as static code replacing the template expression. Rule-based code generation relies on rules that describe how components of the internal representation are translated into the target GPL. [Jö13]

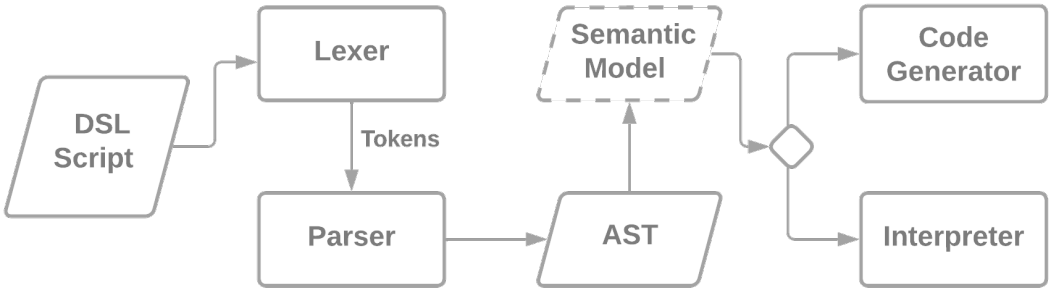

Figure 2.2: Overview of the tools used for DSLs

2.3 NEST Modelling Language

The NEST Modelling Language (NESTML) is a domain specific language in the domain of neuroscience. It was originally implemented for the Neural Simulation Tool (NEST) as an intuitive interface to describe neuron and synapse models. NEST is focused on simulating network dynamics, using spiking neuronal networks and point neurons in the simulation. The simulation itself follows a hybrid strategy. [wwwc] Before NESTML was developed, neuron and synapse models had to be implemented and maintained manually, which required programming knowledge in C++ and the implementation specifics of NEST.
MontiCore [RHK21] was chosen for the implementation of NESTML. The tool provides support in the design of the DSL and generates the necessary toolchain. The supported language for this is Java [AGH05].[PRB+16] Java is not a widely spread language in neuroscience [MBD+15], which causes problems when the generated tooling needs some manual changes or extensions. This led to the conversion of the NESTML toolchain from Java to Python [PRP+18], a language found more often in neuroscience[MBD+15]. With the goal in mind to extend NESTML to SpiNNaker, the following chapters provide a detailed look into the toolchain of NESTML.

2.3.1 Abstractions

NESTML follows the same abstractions that are used for NEST because of its original link to the simulator. Because NESTML only focuses on the definition of cell models and not the network structure, the relevant abstractions that need to be considered are the use of point neurons and the hybrid simulation strategy. The use of point-neurons restricts the type of model that needs to be definable in NESTMl to neurons and synapses. The cell behavior is modeled by differential equations which are evolved during simulation, with NESTML providing the syntax for the hybrid-simulation described in [MMG+05]

2.3.2 Frontend

Grammar

The grammar for NESTML was defined using ANTLR [Par13], a tool that provides syntax to describe a grammar, from which it can generate further tooling (see chapter 2.3.2).

This chapter will not go into the details of the grammar definition, instead explaining the syntax concepts described by the grammar.
NESTML was designed with similar concepts to Python because of its prevalence in computational neuroscience. Python does not use curly brackets to define a code block. Instead, the start of a block is marked by a colon, after which the code needs to be indented by a common number of spaces to assign the statement to the code block. NESTML follows this design concept and also implements standard data types (integer, boolean), extended by physical data types with units. [PRB+16] In figure 2.3 is an example file written in NESTML, implementing every syntax concept. At the beginning of a new implementation stands the keyword `neuron` or `synapse` followed by the name of the model and a colon ①, after which each new block belonging to the model needs to be indented. In the model body, the variables and dynamics are defined by using code blocks provided by NESTML. There are different types of blocks for variables and dynamics of the model. The variable blocks separate variables in different groups of behavior, but the syntax is the same. It starts with the variable name, followed by the datatype, which could be a primitive type or a unit symbol. The right side sets the value and also specifies the unit if necessary because the units of assignments are checked. Starting with the `state` block ② in which the variables are defined that represent the dynamic state of the model. These are complemented by the variables defined in the `parameters` block ③, which don't change in the course of the simulation. Variables that don't describe the cell model, but are relevant for the simulation, can be set in the `internals`-block ④. The dynamics of the model are defined in the form of ordinary differential equations (ODE), inline expressions and kernels in the `equations` block⑤. Inline expressions are marked with the `inline` keyword, kernels use the `kernel` keyword. The ODEs can have different orders, which are indicated by the number of ' (tick) added to the variable name. For the formulation of the ODEs and other functions, the user can reference variables and functions declared in the script or predefined in NESTML. The manual definition of functions is possible in separate `function` blocks ⑥.
Coming to the more neuroscience-specific blocks, there is the `input` block ⑦ and `output` block, ⑧ which define the input- and output ports of the model and which information can be received or emitted. For input ports expecting spikes, the type of spike can also be defined by the keywords `excitatory` and `inhibitory`. The output of the model can be a `spike` or `continuous`.
How the state of the model evolves can be defined in two different dynamic blocks that differ between a time-based and event-based execution. Neurons are restricted to the time-based `update`-block ⑩ while synapse can also make use of the event-based `onReceive`-block (⑨) that can be implemented for each input. The syntax in both blocks allows to use of control-flow statements (e.g. if-then-else) to change the model's dynamics based on the current state. The ODEs defined in the `equation`-block can be referenced by the predefined `integrate_odes` function. This is necessary as the ODEs need to be integrated before they can be used. This is done with the help of the ODE-toolbox [LBME21] which generates a fitting numerical solver.

Parser & Lexer

ANTLR can generate a lexer and parser from grammar written in the syntax it provides. This reduces the risk of errors and makes grammar changes faster to implement. The produced parse tree is agnostic of the domain, so the generated objects are missing methods

```
synapse model_name:                                                    ①
    state:                                                             ②
        var1 real = 0
        var2 mV = par1
    parameters:                                                        ③
        par1 mV = -70 mV
        par2 ms = 2 ms
    internals:                                                         ④
        internal1 integer = steps(par2)
    equations:                                                         ⑤
        kernel kernel1 = exp(-t / par2)
        inline inline1 pA = convolve(kernel1, input1)
        var2' = var2 + inline1
    function reset(temp1 mV) mV:                                       ⑥
        return temp1
    input:                                                             ⑦
        input1 pA <- excitatory spike
        input2 pA <- continuous
    output:                                                            ⑧
        spike
    onReceive(input1):                                                 ⑨
        deliver_spike(var1, internal1)
    update:                                                            ⑩
        if var2 < 0:
        integrate_odes()
        else:
            var2 = reset(par1)
```

Figure 2.3: Composed file showing NESTML syntax

to extract the necessary information. This is solved by the `ASTBuilderVisitor`, which post-processes the generated tree into a domain-specific AST.

Abstract Syntax Tree

The AST that was built by the `ASTBuilderVisitor` uses `ASTNodes` for every node in the tree. Each syntax expression is derived from this component to provide additional functionality that is special to the component, adding some semantics to the AST. NESTML uses the AST as the foundation for code generation, as it does not implement a separate semantic model. The root node is the `ASTNeuonOrSynapse` node, which contains references to all syntactic components defined in this file.

Symbols

Symbols are generated after the AST is finished. It provides the context to the syntax tree by referencing defined variables and functions. The variable's type and the function's return type are given by a `TypeSymbol`, representing a primitive datatype or a `UnitType`. Units in NESTML are managed with Astropy [AA22] which provides scientific units, scaling and unit checking. They are encapsulated in the `UnitType` to control how they are printed during code generation.

2.3.3 Backend

The backend defines how a NESTML script is executed. In the case of NESTML, code generation was chosen to potentially adapt to multiple simulators that don't integrate NESTML's toolchain. Before the code is generated, a pre-processing step is introduced to transform the AST. This is necessary to relay the changes that are applied to the ODEs after they are processed with the analytical solvers returned by the ODE-toolbox. Furthermore, this allows for target-specific transformations. For example, models for NEST are compiled into one binary with a paired synapse, which enables a simple exchange of data between both models at runtime via shared variables. In NEST, this is used for the activity trace of the post-synaptic neuron. Following the model definition, these traces are stored and evolved in the synapse. However, this would lead to each synapse calculating and storing the same post-neuron trace. To prevent this inefficient behavior, the trace and its dynamics are moved to the co-compiled neuron model. This is enabled by the use of a transformer that moves the relevant components from the synapse AST to the neuron AST, making them available during code generation. Furthermore, it adds a reference to the moved trace in the synapse AST.
Code generation is handled with templates by the use of the templating engine Jinja2 [Ron08]. The templates to generate are given as parameters to the code generation process. These are loaded into the templating engine with references to the AST and printers that translate the syntax expression from NESTML to the programming language of the target simulator. The templates implement the static API for the simulator while the model specific information is injected via Jinja2 template statements (see Fig: 2.4). Jinja2 can access Python objects in the AST and call functions to retrieve information. These can be stored in variables for further processing (①) or injected as strings into the templates (⑤). Printers use the latter method, returning string representations of the objects. Static

```
{%- set example = pyObject.getList() %}                    ①
{%- if example != [] %}                                    ②
{%-     for e in example | sort(attribute="name") %}
{%-         include second_example.jinja2 %}               ③
{%-         filter indent(4, True) %}                      ④
{{pyObject.toString()}}                                    ⑤
StaticString                                               ⑥
{%-         endfilter %}
{%-     endfor %}
{%-endif %}
```

Figure 2.4: Example for a Jinja2 template

strings can also be generated as seen in line ⑥. Control-flow statements like those seen in line ② and below allow control about what is generated or looping over lists of data to generate a template for each object. Subtemplates can be included to reduce duplication (③). In NESTML this is used for example to generate if-statements.

2.4 Neuromorphic Hardware

The idea of building a computer imitating the brain was already discussed in the early years of computers. Von Neumann, also known for today's computer architecture, explored the analogies between the brain and computers in 1958 [vN86]. Replicating the brain is interesting because of the adaptability and the low power consumption compared to modern computers, estimated at around 20 W [Jor22]. Today, architectures that resemble the brain are commonly referred to as neuromorphic hardware since Mead used the term in his work about a brain-like analog circuit [Mea90]. The neuromorphic architecture is based on Spiking Neural Networks (SNN), a model of the network found in the brain. Neurons and synapses are abstracted into mathematical models that exchange information as spike events, which resemble the action potential generated by the neuron. SNNs are a commonly used abstraction in computational neuroscience, explaining the interest in neuromorphic hardware for simulation, leading to the development of neuromorphic hardware often targeted at neuroscience ([PBC+22], [FB20]).
However, with the resurgence of Artificial Neural Networks (ANN) as a solution for different problems, like natural language processing([Ope23]), interest outside of neuroscience is on the rise. ANNs are similar to SNNs, abstracting neurons and synapses into mathematical models, with models for ANNs being more problem-specific. Another difference is found in the communication strategy. Instead of spike events, communication in ANNs is modeled with floating point numbers with fixed timing. However, it was shown that models implemented for ANNs can be transformed to work in SNN models [RLH+17]. From the computational point of view, either of these models is unsuitable to be executed on von Neumann machines, as modern computers generally rely on the von Neumann architecture (with some extensions), which has an architectural flaw called the von Neumann bottleneck [Bac78]. A von Neumann computer uses a data storage and a central processing unit (CPU), which communicate via a bus. For the CPU to process data, it must first be received from the data storage, leading to a dependence on the data transmission rate of the bus, blocking the CPU in the meantime. This problem becomes increasingly more relevant when using parallel processors accessing the same memory. The neuromorphic architecture solves this problem as there is no separate data storage. Instead, data is encoded through plasticity mechanisms like Spike-Timing Dependent Plasticity (STDP) replicating the formation of memories.
Neuromorphic hardware is based around many highly parallel computing nodes, each executing a small subset of neuron and synapse models from the whole network population. Like in [Mea90], these can be analog circuits that behave similarly to dynamics seen for synapses and neurons or simple digital processors. The latter can be based on the von Neumann architecture, as the nodes don't share any data, reducing the impact of the von Neumann bottleneck. Intel's Loihi [DSL+18] uses a custom processor, while SpiNNaker [FB20] uses ARM microprocessors.
In computer science, Graphic Processing Units (GPU) are often the tool of choice, featuring a high number of cores [2]. However, they use a shared memory and are significantly less efficient [3] compared to existing neuromorphic hardware. [vARS+18]
Of course, neuromorphic hardware features its own problems. Analog circuits use a fixed design and connections are wired so they can't be programmed to fit the problem. Furthermore, integrating analog systems with modern computer hardware needs an additional step to convert between the analog and digital domain. Digital solutions have to employ

[2] 16000 CUDA-Cores in the NVIDIA GeForce RTX 4090 [wwwd]
[3] 450 W for the NVIDIA GeForce RTX 4090 [wwwd]

a strategy to distribute and manage the potentially large number of spike events. They also lose some of their efficiency advantages [vARS+18].

2.5 SpiNNaker

SpiNNaker [FB20] is the brainchild of Steve Furber, with the first public mention in 1998. It is a neuromorphic architecture based on ARM microprocessors connected into a network. The project emerged from the research question of efficiently integrating associative memory in Very Large Scale Integration (VLSI) architectures. After some research, it was clear that the architecture would come down to a neuromorphic architecture. Earlier implementations mainly focused on analog implementations of neurons and synapses combined with digital communication, However, Furber's previous involvement with ARM and asynchronous digital circuits led to a design based on ARM microprocessors implementing the model and network details in software. As mentioned in the introduction to neuromorphic computing, handling many communication messages is a challenge. Following the biological example is hard to achieve, as to be generally applicable would mean building a physical connection between each neuron, which apart from spatial problems also hinders scalability. This was previously solved by using a bus system with, compared to biological spikes, short digital messages encoding a unique address of the source neuron as the spike. Because of the short pulse, spikes generated simultaneously can be serialized without breaking the concepts found in nature. This protocol is called Address Event Representation (AER[Mah92]). Each spike is a broadcast message on the bus, where each neuron determines if it should receive the spike through the address of the source neuron.[Mah92]
As this solution uses a single bus, the scalability of the system is dependent on the bus throughput. This let the team around SpiNNaker to switch from a bus system to a packet-switched network, where packets can be routed to specified targets. Instead of being bound to the fixed specification of bus throughput, this now also allows scaling the network capacity by adding more routers. They call this Multicast Packet-Switched AER. With this architecture, they were able to achieve a system with 1 million processors called SpiNNaker1M. All information about SpiNNaker presented in this chapter comes from the book "SpiNNaker: A Spiking Neural Network Architecture" [FB20].

2.5.1 Hardware

After developing the architecture, it now comes to implementing the concepts in hardware. The base block for the SpiNNaker system is a custom Complementary Metal Oxide Semiconductor (CMOS) chip housing 18 cores of the type ARM968, chosen because of its power efficiency. It is important to note that the microprocessor does not include a floating point unit. One of these cores is set aside for monitoring the state of the chip. The cores are complemented by a router with 24 ports, of which 18 are connected to the processors and six are used to connect to other chips, building a triangular mesh structure. Four types of packets are defined for communication in the network(Table 2.1).

Memory The processors and router need to be paired with appropriate memory for fast operation. The routing table is stored in a combination of Ternary Content-Addressable

Name	Purpose
Multicast	Used for spikes, can be routed to multiple targets
Point-to-Point	Inter-chip communication, e.g. requesting data from other chips
Nearest Neighbout	Communication with directly connected chips, used during boot
Fixed Route	Additional data for host

Table 2.1: Network packets used in SpiNNaker

Memory (TCAM) and Random Access Memory (RAM). Like in Content Addressable Memory (CAM), the address of the routing configuration in the RAM is retrieved based on the binary key (in this case, the source neuron address) by matching it to the keys stored in the TCAM. The keys stored in the TCAM are extended by a mask that allows to set bits of the key as wildcards, either matching to any input or none. Each processor

Key	Mask	Function
0	0	Match any
0	1	Match 0
1	0	Match none
1	1	Match 1

Table 2.2: Matching strategy in TCAM memory

has access to its local Tightly-Coupled Memory (TCM), one for the code (Instruction TCM) and one Data TCM, both based on Static RAM (SRAM). The DTCM is used to store the stack and data working set. Neither stores the synaptic weights, as there are too many to be stored in SRAM. Instead, each chip provides a shared Synchronous Dynamic RAM (SDRAM) that can be accessed by each processor through a Direct Memory Access (DMA) unit, offloading the overhead of retrieving the data from the microprocessor to a separate chip. The connection to the SDRAM is done through an internal Network-on-Chip (NOC), which connects all processors and the router.

Boards With the goal in mind to build a 1 million core machine, multiple iterations of boards were developed. This started with the proof-of-concept boards SpiNN-1 and SpiNN-2. SpiNN-3 was then the first development platform distributed to other institutions. It features four SpiNNaker chips and an Ethernet adapter to connect to a host machine, which is controlled by one of the chips referred to as the Ethernet chip that relays the information to the other chips. The connection between multiple boards is only available in a limited manner through two custom inter-chip connectors. Because of the low power usage of the microprocessor, a single chip only uses around 1W of power, which is also reflected in the board's power consumption using a 5W power supply. Further iterations were aimed for production, with SpiNN-4 as the test platform and SpiNN-5 as the finished production board. Both boards feature 48 SpiNNaker chips and an Ethernet port to connect to the host. Other external devices can be connected through a custom connector port or three SATA connectors. In total nine SATA connectors are on the board. The remaining six are used for connecting to other boards. Three Field Programmable Gate Arrays (FPGA) control the inter-board communication, which are set up as seen in figure 2.5. For the SpiNNaker1M, the 1 million core machine, 1200 SpiNN-5 boards are connected in 10 cabinets, needing a 100kW power supply.[FB20]

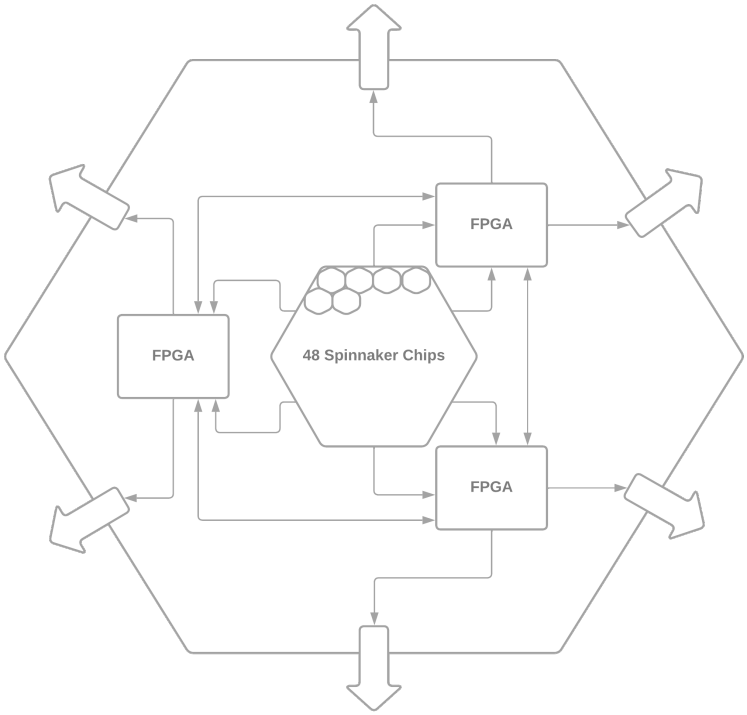

Figure 2.5: Layout of the FPGAs for inter-board connections on the SpiNN-5 board (adapted from [FB20][Fig.3.7])

2.5.2 Software

SpiNNTools SpiNNaker provides a software stack written in C [KR06] called SpiNNTools [RBD+19]. It controls the boot process, loading the simulation data and extracting the results. Each chip stores the boot program in the Read Only Memory (ROM). On startup, it runs checks of the chip and its cores and selects a working processor as the monitor core. The monitoring processors run the SpiNNaker Control and Monitor Program (SCAMP). It enables control over the chip, like flashing the simulation code from the host and communication with the other chips' monitoring cores. As only one chip on each board can communicate with the host (Ethernet chip), SCAMP is distributed to other chips on the board through nearest neighbor packets. Now that the chips can communicate, the set network is mapped out, recording defective cores.2.7.[RBD+19]

After this setup process, the other cores are initialized by writing the user code to the core's ITCM, which includes the SpiNNaker Application Runtime Kernel (SARK) and an event-driven operating system called SpiN1API. SARK provides a hardware abstraction layer with functionality for the DMA, network and interrupts. This is used by SpiN1API to control the runtime. It relies on an interrupt-driven design to leverage the low power consumption of the ARM processor's sleep mode. The interrupt events can be linked to user-defined callbacks that handle the event and can, for example, be triggered by received packets or a timer.[RBD+19]
Apart from the functionality for setting up the SpiNNaker board, there is also an API for implementing graph-like problems provided. For this, it provides data structures

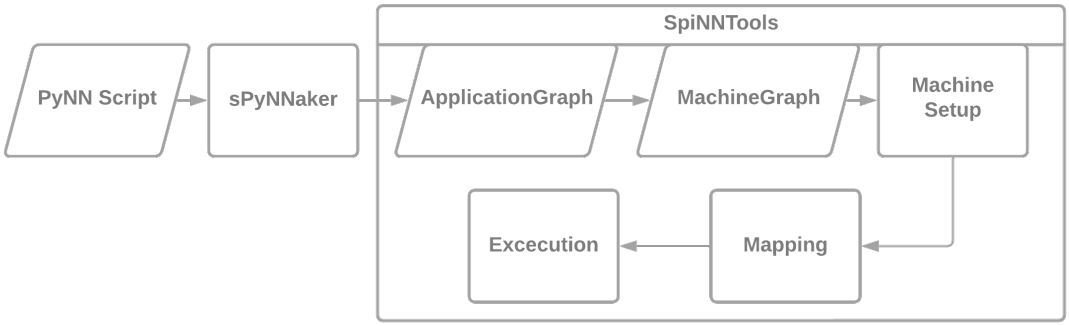

Figure 2.6: Process flow for executing PyNN scripts on SpiNNaker, with the intermidiate data structures

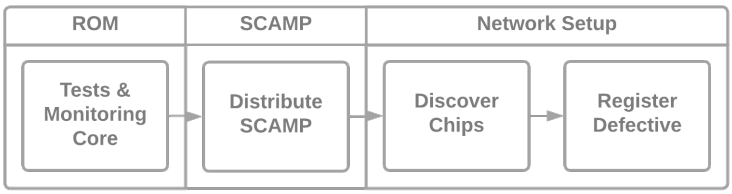

Figure 2.7: Bootprocess for the SpiNNaker board(adapted from [FB20][Fig. 4.2])

to define the graph either on a machine level (`MachineGraph`), where the graph is directly mapped to the machine, or via the `ApplicationGraph` modeling the program components and their communication. As a reference, see Fig 2.8. Structurally, both graph types are similar in design, featuring vertices and directed edges that make up the graph, where the `ApplicationVertex` and `ApplicationEdge` can be converted into multiple `MachineVertex` and `MachineEdge` respectively. Vertices are the computational nodes in this graph and are linked to the specific binaries. In the case of `ApplicationVertices`, these can also be split into `Atoms` (e.g. neuron-synapse model), which can then be mapped to a `MachineVertex` for execution. `MachineVertices` represent processors of the machine and are therefore not splittable. Edges in the graph represent communication paths, which in the case of `MachineEdges` are the connections between the chips representing the triangular mesh. On the other hand, represent `ApplicationEdges` the data exchange between applications. Edges can also be grouped into `OutgoingEdgePartition` modeling the multicast packets. The graph must only be created with one of the types, as the SpiNNTools automatically maintains both types. After creating the graph, the machine is mapped according to the `MachineGraph` and sets up the network components, like routing tables for the defined edges and the routing keys (Fig. 2.9).[RBD+19]

sPyNNaker To simplify the design process for SNNs, the team around SpiNNaker added support for PyNN, handled in the library sPyNNaker [RBB+18]. PyNN can be described as an internal DSL in Python, adapted by many different simulators ([wwwe]). It features a declarative programming approach, with which the network structure can be defined. The network is built from `Populations` that group neurons together. Mod-

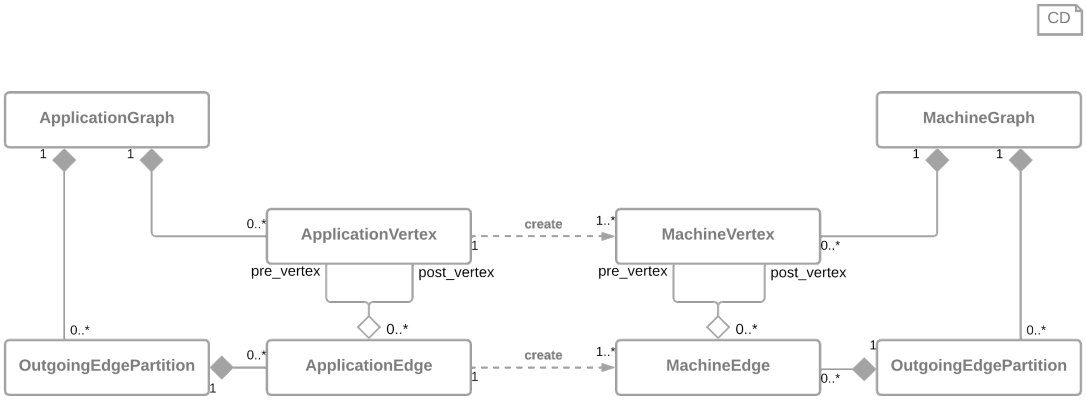

Figure 2.8: Classes of the application and machine graph and the conversion from the first to the latter (adapted from [FB20][Fig. 4.6])

els of neurons are provided with the target simulator, as they also need to be implemented for the target simulator. Instead, PyNN provides an API, that needs to be implemented by the models. Connections between the `Populations` are called `Projections`, which can also be reflexive to the same `Population`. These represent the network synapses, which are again included by the simulator maintainers, in this case, in the sPyNNaker library. Next to the models, sPyNNaker also provides the integration mechanism to SpiNNTools, enabling the translation of `Populations` and `Projections` to an `ApplicationGraph`. `Populations` are mapped to `ApplicationVertices` and `Projections` to `ApplicationEdges`. The synaptic information for each projection is stored in a row datastructure, visualized in Fig. 2.10. Each row describes the connections from one neuron to all post-synaptic neurons, represented by one synapse for each connection in that row. They contain a plastic-, static- and fixed-plastic region. Static synapses are written to the fixed region because their structure is different from plastic synapses. The static synapse is a 32-bit word, storing the connection strength(weight), artificial transmission delay and type (excitatory, inhibitory) for every connected post-synaptic neuron.[RBB+18]
In contrast, plastic synapses are split into a dynamic and a fixed part stored in the respective region. The fixed-plastic region contains delay, type and target neuron id, while the weight is stored in the plastic region for processing reasons. Apart from the weights, a trace of the pre-synaptic activity is stored, which is used for plasticity. It's important to mention that due to PyNN limitations, fixed and plastic synapses can not be mixed. [RBB+18]
For the simulation sPyNNaker uses the idea of hybrid-simulation [MMG+05] described in chapter 2.1.4 that updates neurons with a fixed frequency (timer interrupt) and synapses on events, in this case on a received pre-synaptic spike. Spike events are distributed using the multicast packet.[RBB+18]

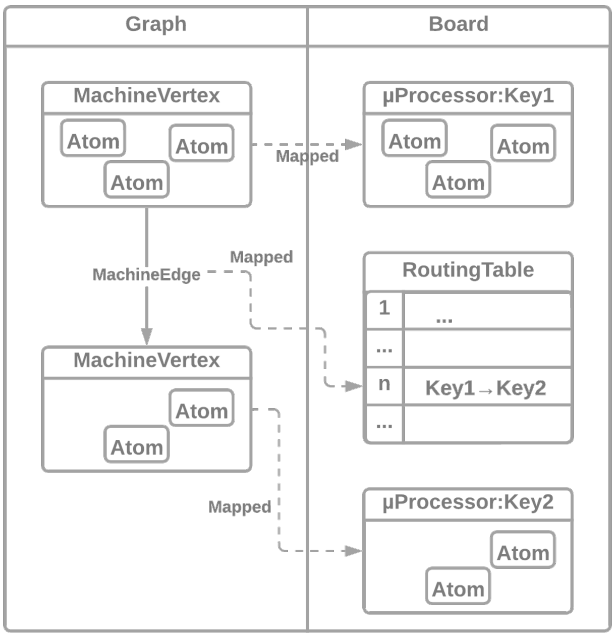

Figure 2.9: Mapping from `MachineGraph` to SpiNNaker board

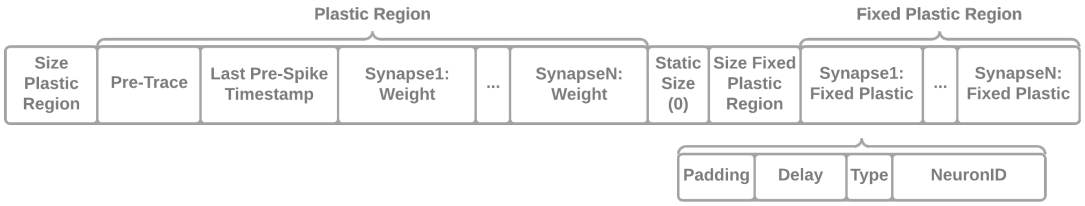

Figure 2.10: Synapse row for a plastic synapse model (adapted from [FB20][Fig.4.20])

Fixed point arithmetic

Because the microprocessor used for SpiNNaker does not provide a floating point unit, this datatype is inefficient and should be avoided.[4] Instead, the SpiNNaker team provides several fixed point datatypes as an extension to the "stdfix" library to conform with the ISO/IEC Standard DTR 18037 norm [ERW07]. The most important for use with SpiNNaker is the `S1615` datatype, a signed 32-bit number matching the 32-bit architecture of the ARM processor. Unlike floating point datatypes, fixed point types are not based on an exponent and mantissa but define a fixed point in the bit representation of the number, separating it into an integer and a fractional part. In the case of `S1615`, this would mean 16-bit integer resolution and 15-bit fractional resolution, with one bit used for the sign. Using this structure allows for efficient execution with integer arithmetic but comes at the cost of precision and range compared to single and double floating point numbers. `S1615`, for example, has a precision of roughly 0.0000305 with a range of `[-65536,`

[4]When used, the FPU is emulated in software, leading to overhead for each operation. For example, multiplication using 26 cycles instead of 1 [wwwf]

65535.99996948242].
This is an example of multiplication for matching fixed point numbers implemented for SpiNNaker, in this case for S1615:

$$\text{mul}((\texttt{S1615}x, \texttt{S1615}y)) = (\text{REAL})((\_\_I64(x) * \_\_I64(y)) >> 15)$$

Before the multiplication, both are cast to a 64-bit wide integer before multiplication. This is due to the effect, that the number of bits before and after the decimal place is double of the value before, so in the case of S1615, the result is a SS3230 which places the decimal point after bit 29. To extract the S1615 value the underflow is removed by shifting down and the overflow is cut by casting back to the S1615 datatype. [wwwa]

Chapter 3

Extension Process

Adding a new target to NESTML can be achieved by writing the necessary transformers, printers and templates for the target simulator. Before the implementation, the compatibility of the target simulator with NESTML needs to be confirmed (chapter 3.1). If this is the case, the focus can be shifted to the implementation. When there are minor differences to the target's syntax tree, these can be fixed with a transformer that allows to alter the AST, making the components available in the correct places or simplifying the access (chapter 3.2). The actual implementation is handled with templates that are based on the target's API and contain static code in the programming language and template statements that are dynamically generated (chapter 3.4). Printers handle this generation, accepting instances from the AST and returning a string representation (chapter 3.3). To confirm the correct implementation, some tests should be carried out compared to a known good implementation. We propose the NEST for this and some tests that provide insight into the functionality in chapter 3.5. The project for this thesis was the extension of NESTML to SpiNNaker, which is therefore chosen as an example throughout this chapter. Still, the process is generally applicable to any new target.

3.1 Compatibility

A starting point to determine compatibility is the supported model type. This describes how the cell is abstracted, which in the case of NESTML as well as SpiNNaker is the point-neuron model presented in chapter 2.1.4. SpiNNaker uses a definition file that integrates different modules, like a threshold or input-type module. This is only code structuring and can be reduced into a single component. This makes it possible to be described with a NESTML script, as splitting of different components is not intended in NESTML.
Apart from the model type, it is also of interest if the simulation method used in the target simulator is supported. Possible is a time-driven, event-driven or hybrid simulation (see chapter 2.1.4). NESTML and SpiNNaker are built around the hybrid simulation method, with NESTML providing the necessary syntax blocks (`update`: time-driven, `onReceive`: event-driven) and SpiN1API using this simulation strategy. From this analysis follows, that SpiNNaker is a possible target for NESTML.

3.2 Transformer

With transformers, we can alter the AST structure if necessary. An example is discussed during the introduction to NESTML in the chapter 2.3.3. SpiNNaker, like NEST, uses binaries of neuron-synapse pairs but the standard implementation keeps the trace of post-synaptic spikes in the synapse model so we decided to keep it that way at the moment. Switching to the concept used in NEST could be implemented as a future improvement. As the structure does not differ from the one used in NESTML, no transformation to the AST is necessary.

3.3 Printer

NESTML already includes a set of printers used for the NEST target and a standalone Python implementation, which can be used as a guideline for the printer implementations required for SpiNNaker. A printer for each syntax element must be defined to represent it in the target language. Some of the work can also be offloaded to the templates. In the implementation for NEST, this is done for loop and control-flow statements. The necessary printers can be grouped into variable- (e.g. variables, datatypes), expression- and function printers (e.g. predefined functions).

3.3.1 Expression Printer

Syntactical structures like loops and control flow statements are handled via templates in NESTML. Therefore, printers in this category are focused on functions and expressions. Expressions are a concatenation of operands, constants, variables and function calls. Operators are the basis for the expressions, describing the interaction between the other components. They are differentiated by the number of operands they act on. Most operators are binary, expecting two operands referred to as the left- and right-hand operands, but NESTML also supports several unary and ternary operators. An essential aspect of operators is the precedence, which in the case of NESTML is the same as in C so no special care needs to be taken. The code for the expression is generated top-down, where each operand can be another expression or a simple expression, which are leafs in the AST, like literals, variables and function calls. They have a static structure and, therefore, don't have children. Literals are handled directly by the `SimpleExpressionPrinter`, as they can be returned verbatim. Variables and functions are handled in separate printers.

3.3.2 Variable Printer

Variables are composed of a name and a type. Optionally, they can also contain a unit. The variable name is already present as a string and can be printed directly. Type symbols are the primitive datatypes that are defined for NESTML, which therefore need to be represented for SpiNNaker as well, mapping the datatypes to the equivalents in C. Because of the 32-bit architecture used in SpiNNaker, integers are mapped to the `int32_t` type. Special attention also needs to be given to the data types not found in native C, namely `real` and `unit` datatypes. As the name suggests, are `real` variables used to store values from $\mathbb{R}$ and therefore need to be a decimal datatype. With SpiNNaker comes a collection

of custom datatypes, including the `REAL` datatype. Because of the missing FPU on the microprocessor, it is designed as a 32-bit wide signed fixed point decimal, using 16-bit for the natural and 15-bit for the fractional part. The same type is also used for the unit-type variables. Printing the variable name is done by `VariablePrinter`, which switches between how the variable is printed based on what kind of variable it is. It checks if the variable is a known constant, printing the assigned value (e.g. e). Other differentiating factors are based on syntactic notation, like vector (arrays) and buffer variables. Buffers are used for the input ports, which, in the case of SpiNNaker, are stored in an array, where each entry is a dynamic value assigned to one input port. SpiNNaker does not support units, so the values of parameters need to be nondimensionalized beforehand. If the variable has a unit, its magnitude is compared to the expected magnitude and adds the necessary conversion factor to the generated output. An option that can be selected for each printer is the `with_origin`-flag, printing the variable with reference to the NESTML code block it was defined in. When the parameter definitions are generated as part of an object in the target language, this allows referencing that object. C does not support objects, but `structs` can be used in a similar manner.

3.3.3 Function Printer

The function printer first checks if it is a custom function predefined from NESTML. These functions differ from a regular function call so far that they may be expanded to multiple statements. An example is the `emit_spike`-function. Function calls that don't belong to this group are handled like function calls in the target language. If the function doesn't expect any parameters, the function is called with empty parentheses. Otherwise, the parameters are inserted first as placeholders in the form "{n!s}", where `n` is replaced by the place of the argument in the original call. The placeholders are replaced with the actual variable names in the second step. This allows for a more flexible function call structure and reordering of parameters. Next to the more complex predefined functions, NESTML also supports predefined arithmetic functions like `exp`, `log` and `sinh, cosh, tanh`. These can't be translated to standard math functions provided with C because of the missing FPU in SpiNNaker. With SpiNNaker, fixed point versions of `exp` and `log` are provided so that we were able to easily implement the arithmetic functions that can be defined with these, like the mentioned `sinh`, `cosh` and `tanh`.

3.4 Templates

Each target offers a different API that must be reflected in the templates, with static code defining the required function names, structs and definitions. The implementation of this API is generated based on the defined generator expressions written with Jinja2. Apart from the implementation, templates are also used to construct control-flow expressions and other syntax structures defined in NESTML that are not present in the target language. This is done because the bodies of these components are variable and are easier implemented with the generative approach. In the case of SpiNNaker, we were able to recycle most of the already implemented templates for the syntax components, as C and C++, the language of the NEST backend, are similar. The template for neuron and synapse implement different APIs so that they are considered separately in the following chapters (3.4.1, 3.4.2) and will be specifically for SpiNNaker.

3.4.1 Neuron Template

Implementing a new neuron model is documented on the SpiNNaker website [wwwg], using an example repository providing template files to be extended manually. The standard way of building a model is modular, using a base file that implements the SpiNNaker API for neurons and invokes the necessary functions implemented in separate components. But they also provide a template file combining all functionality, which is suitable for use with NESTML. Because SpiNNaker uses PyNN to initialize the models, we need to build a template for the PyNN model as well as the model implementation in C. As mentioned is the PyNN model written in Python and used as an interface to initialize the model written in C. Therefore, it does not contain any model-specific logic. Instead, it defines parameters to be set by the user and internal parameters generated from the NESTML model file. All parameters are stored in a data structure that describes the data type of the parameters. Apart from that, it also provides relevant methods for the execution. For example, the function `binary_name` returns the name of the binary linked to this model. Other essential functions are related to recordable variables. These state variables of the model evolve dynamically and are, therefore, interesting to analyze after the simulation run. Variables specified in these functions can then be recorded during the runtime, reporting their behavior over time. The binary is the compiled C file with all dependencies. As C does not support object-oriented programming, the neuron is modeled with a `struct` that stores all model parameters and methods that act on this model. The `struct` comprises three substructures, storing the input, state and parameter variables, keeping the data separation from the NESTML model. The memory layout of the neuron `struct` is chosen to be the same as the data structure defined in the PyNN model so that the neuron `struct` can be directly initialized from the data copied to the SpiNNaker memory via the pipeline implemented by sPyNNaker and SpiNNTools. It is handled in the `neuron_impl_load_neuron_parameters` method, which has as a parameter the pointer to the memory address where the data is located in memory. A pitfall during the development was the order of the variable names generated, as the order of the variables in the dictionary differed between the generation of the Python and C templates. This matters as the values declared through the PyNN interface are written to the DTCM in an unexpected order, leading wrong initialization of the model parameters. This was solved using the `sort`-function built-in with Jinja2.
The dynamics of the neuron model are implemented in the methods `neuron_impl_add_inputs` and `neuron_impl_do_timestep_update`. Events are triggered when a new spike arrives at one of the input ports, invoking `neuron_impl_add_inputs` to handle the event according to the input port. The value of the event is already scaled to the value associated with the connection strength set by the synapse, so it is currently handled by accumulating an internal variable associated with the input port. For every simulation timestep, the function `neuron_impl_do_timestep_update` is called to update the state of the neuron and calculate the behavior. Here, the accumulative variables of the input ports are evaluated, and the internal state is updated based on what was defined in the NESTML `update`-block. This includes whether this neuron should send out a spike event.

3.4.2 Synapse Template

Same as for the neuron, we also need to implement a PyNN and a C template for the synapse. However, there is no example of extending SpiNNaker by a new synapse disclosed with SpiNNaker, therefore we derived the API from existing implementations.

The model for PyNN is originally composed of two classes, called `weight_dependence` and `timing_dependence`, handling weight-related parameters like limits and weight update dynamics respectively. These components can't be defined separately in NESTML and even are difficult to separate so we merged the API into a single class. Interesting functions of the class are the `write_parameters`, `get_plastic_synaptic_data` and `vertex_executable_suffix` functions. The function `get_plastic_synaptic_data` is part of the initialization of the synapse row, defining the layout of the fixed-plastic region. The static region is disregarded, as plastic and static synapses can't be mixed in a row. We use the same layout, so nothing needs to be changed. They are organized in rows, a visualization can be seen in figure 2.10. Each row describes the connections from one neuron to the post-synaptic neurons, represented by one synapse for each connection in that row. They contain a plastic-, static- and fixed-plastic region. Static synapses are written to the fixed region because their structure is different from plastic synapses. The static synapse is a 32-bit word, storing the connection strength(weight), artificial transmission delay and type (excitatory, inhibitory) for every connected post-synaptic neuron. Because static synapses are already implemented, we won't focus on these
In contrast, plastic synapses are split into a dynamic and a fixed part stored in the respective region. The fixed-plastic region contains delay, type and target neuron id, while the weight is stored in the plastic region for processing reasons. Apart from the weights, a trace of the pre-synaptic activity is stored, which is used for plasticity. It's important to mention that due to PyNN limitations, fixed and plastic synapses can not be mixed. [RBB+18]
For writing the parameters for the synapse dynamics to the machine's memory, the `write_parameters` function is used. It defines the memory layout similar to the data structure used for the neuron.
SpiNNaker only supports models compiled into one binary, implying that the neuron and synapse models must be compiled into one file. To identify the synapse used, the name of the binary is suffixed with the term ¨_with_änd the synapse name. This is reflected by the `vertex_executable_suffix` function, which returns the expected suffix when using the model. The C implementation of the model is influenced by the use of the two memories, the SDRAM for the synapse rows and the DTCM for synapse parameters. To simplify the initialization, the synapse parameters are stored in a `struct` with the same memory layout defined in the `write_parameters` function. Retrieval of the synapse row is managed by SpiN1API, but the data needs to be combined with the synapse parameters. For this, there is the `weight_state_t`-struct that stores a reference to the synapse parameters and the current state of the synapse.
The update of the synapse is triggered when a spike of the pre-synaptic neuron arrives. Post-synaptic spikes that arrived since the previous pre-synaptic spike are buffered with their timestamp and a trace value to calculate the temporal relation to the pre-synaptic spikes(see `synapse_dynamics_process_post_synaptic_event`). The trace value dynamically evolves with every received post-spike. It first is decayed according to the time since the last post-spike and then increased by one. This is computationally more efficient when calculating the depression. Handling the generation of the trace variable and update posed a problem with NESTML. In NESTML, there is no keyword to differentiate between the trace for post-spikes and the also exiting trace for presynaptic spikes. However, the dynamics for both are defined in the `equations`-block but need to be generated in two different functions. The current solution solves this by hard-coding the variable names, which works for the predefined model files in NESTML because of the naming convention, but this is not generally applicable. Possible solutions are discussed in chapter 4.

When a pre-synaptic event arrives, it is handled by the `synapse_dynamics_process_plastic_synapses` function. As mentioned, the pre-synaptic activity is stored in a trace as well, which evolves the same way and encounters the same problem. However, the pre-synaptic spike also triggers the update of the synapse dynamics, which is an implementation of STDP. As a reference of how the post-synaptic spike impact the synapse, see figure 3.1 .The depression of the weight is calculated based on the trace of the last post-synaptic spike, which is decayed beforehand based on the elapsed time. Calculating the potentiation needs more attention, as the same trick with the trace is impossible here, as it is calculated based on the timing of the previous pre-synaptic spike. Instead, we iterate over the list of relevant post-spikes and calculate their impact based on the time between the pre- and post-spike.

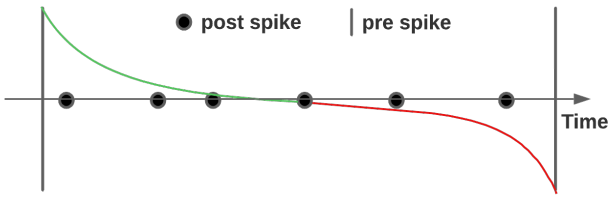

Figure 3.1: STDP window for weight update. Post-synaptic spikes under the green curve (left) lead to potentiation, while post spike under the red curve (right) lead to depression

3.5 Validation

Only the neuron implementation was validated because the synapse is not yet executable. However, in chapter 3.5.1 we propose possible tests for validation. All tests were done with the exponential integrate-and-fire model with post-synaptic current (iaf_psc_exp) [TUM00]. The NESTML script for this model already exists, shown in figure A.1. First, we test the model's behavior on incoming spikes and compare the model's numeric accuracy against the recorded values of NEST as a known good reference. In a second test, we defined a simple network of two neurons, with a static connection from the first neuron to the second (Fig: 3.4) to test if the model behaves as expected when generating a spike. Lastly, we adapted a balanced network, testing the model's behavior in a larger network. For the balanced network, we focus on recording the generated spikes on a network level, disregarding the states of the individual neuron. Interesting values for the other tests are the membrane potential and post-synaptic current to see the effects of a received spike. The effects of a received spike are visible in the (excitatory/inhibitory) synaptic current (`I_kernel_exc__X__exc_spike`)[1] and the membrane potential (`V_m`).

Numeric accuracy For this test, we set up a single neuron connected to an external spike generator which emits on spike at a fixed point in the simulation. The emitter is connected with an excitatory static synapse to the neuron model, which is the generated implementation of the iaf_psc_exp model for SpiNNaker. An incoming spike to the neuron

[1] In the tests shown here, we focused on excitatory connections. Inhibitory connections were tested the same way with the same results and were therefore omitted

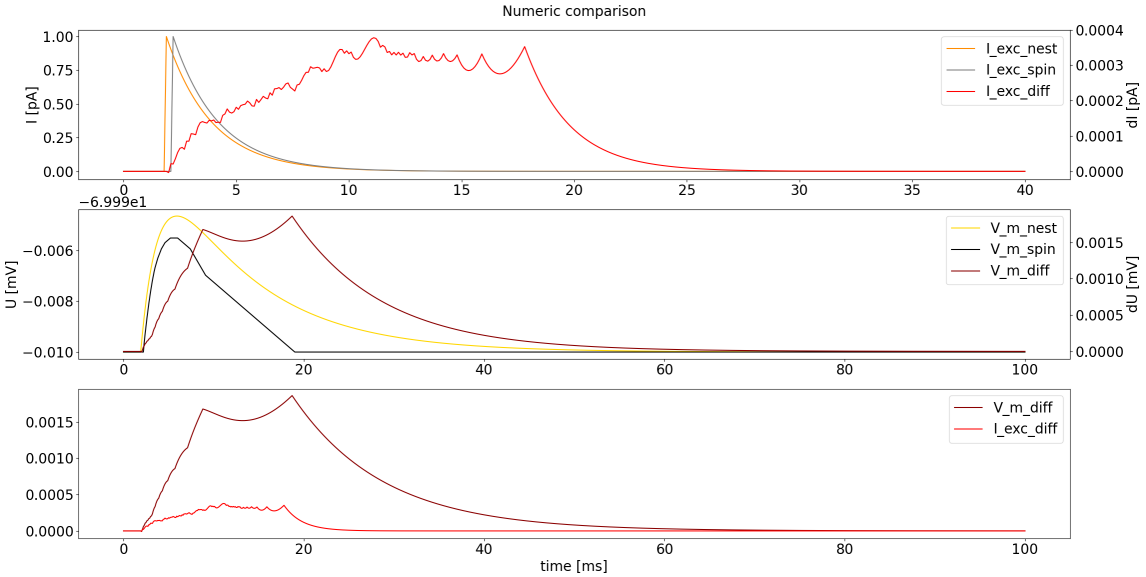

Figure 3.2: Shown here are the recorded values of the excitatory synaptic current (top) and difference (scale on the right), membrane potential (middle) and difference (scale on the right) and a graph composed of both errors between the NEST reference values and SpiNNaker (bottom) for a single neuron. The recorded data was synchronized on the first value above the initial value, which differs by 0.3 ms. All data recorded from SpiNNaker is shifted slightly to the right to reduce overlap and increase readability. The input spike strength was set to 1.

increases the synaptic current equal to the weight of the synapse and decays exponentially over time. The membrane potential integrates the synaptic current, leading to a proportional rise, also decaying exponentially over time. Both values were recorded for SpiNNaker as well as NEST to compare the accuracy of the calculated values.
The recorded data is synchronized on the first increase from the initial value to counter the timing differences when the spike was received between the simulators. Compared to NEST, the spike event in SpiNNaker was delayed by 0.3ms (simulation time). We argue, that this comes from the overhead due to the transmission of the spike and retrieval of the synapse data from the SDRAM. Results from this test are shown in Fig. 3.2. The top and middle graphs show the excitatory synaptic current and the membrane potential, respectively. The difference was calculated by subtracting the synchronized values of SpiNNaker from NEST. Therefore, positive values in the difference show a higher value recorded from NEST. The difference seen in the synaptic current (3.2[top]) can be explained by the lower precision in the SpiNNaker machine due to the use of fixed point numbers and the buildup of this error over time. For the `S1615` datatype in SpiNNaker, the precision is roughly 0.0000305. Problems with this can be observed for small changes, like seen in Fig. 3.2[top] and Fig. 3.2[middle] in the interval from 9.1ms to 19ms, when the values are close to zero and the rate of change is low. Expected would be an exponential curve like recorded from NEST, but instead, we see this linear behavior because the derivation is smaller than the precision `S1615` provides. To support this, we looked at the discrete derivation, giving us the numerical change per simulation step (Fig. 3.3) in this interval, showing that the difference equals the lowest possible with roughly -0.0000305. Furthermore, the membrane potential depends on the synaptic current, so the accumulative error in the synaptic current propagates to the membrane potential, leading to an additional error. In this case,

because the membrane potential is proportional to the excitatory synaptic current, the membrane potential is further decreased compared to NEST, in addition to the errors from the fixed point arithmetic. The team around SpiNNaker argues that these deviations are similar to noise present in the brain and are therefore not a problem [FB20][p.27]. Because these rounding errors are deterministic, as every value is always rounded to the same value, this is not a problem for repeatability.

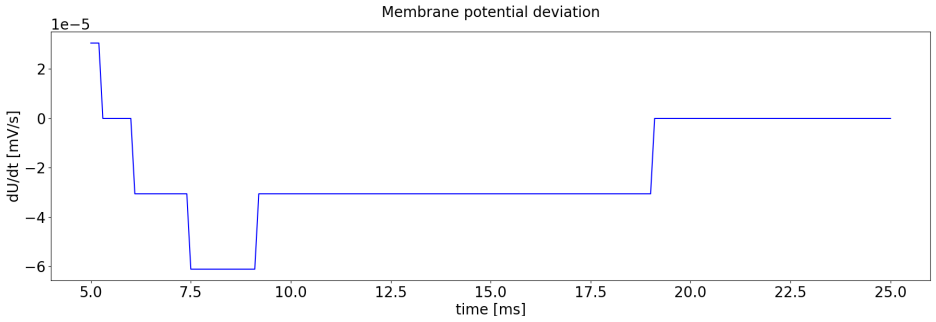

Figure 3.3: Discrete derivative for the membrane potential recorded on SpiNNaker

Figure 3.4: Network layout used to validate correct spike generation in our generated model

Neuron chain In the second test case, two neurons were set up in a chain with an excitatory static synapse (Fig. 3.4). We refer to the first neuron as the emitting and the second as the receiving neuron. The emitting neuron is excited by an external spike generator until the neuron generates a spike. This is received by the receiving neuron. Of interest are the dynamics of the synaptic current and membrane potential in both neurons. The recorded Results for this test case are shown in Fig. 3.5. The emitting neuron reached the threshold (`V_th`) at 17 ms, generating a spike. After generating the spike, the neuron is set to the refractory state, visible by the flat spot in the interval from 17.1ms to 19.1ms, which matches our expectations of a 2 ms refractory period defined in the model. Furthermore, the membrane potential is set to the reset voltage (`V_reset`), rising after the refractory period because of the remaining synaptic current. This is caused by the update strategy implemented in the iaf_psc_exp model not evolving the dynamics while it is refractory. That the spike was actually generated is visible in the recorded values from the receiving neuron by the rise in the synaptic current and membrane potential.

Balanced network Balanced networks are carefully constructed networks that capture a phenomenon in the brain resp. the coretex to be more accurate. This is a highly connected area in the brain, which was observed to react to small fluctuations in the input. [vVS96]. It is believed that the reason for this lies in the inhibitory and excitatory

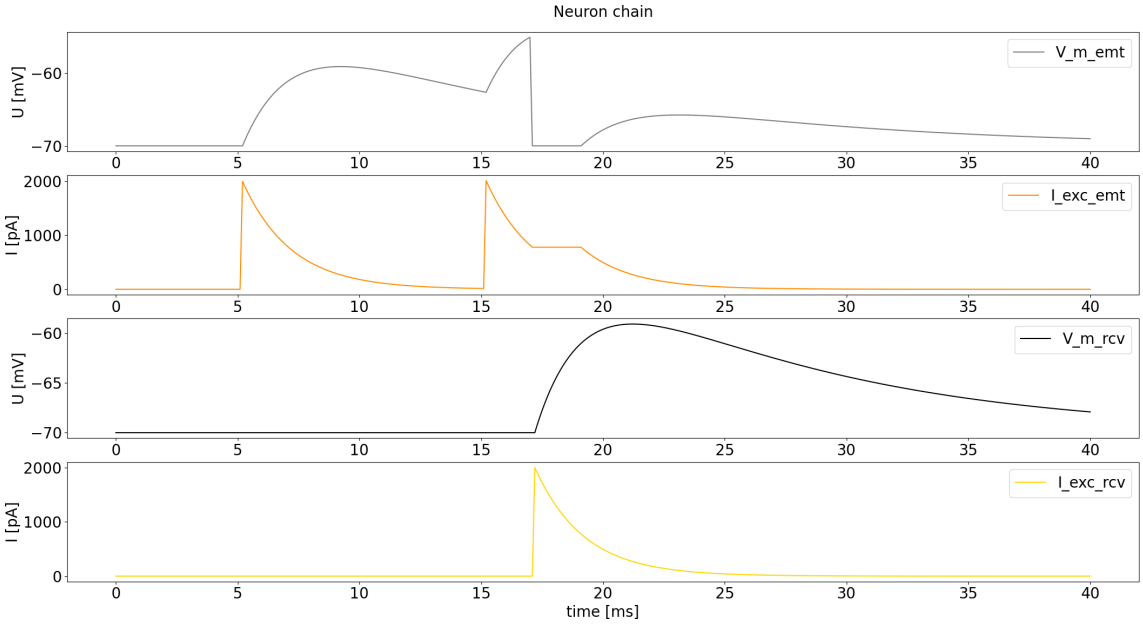

Figure 3.5: A simple network of two neurons connected by a static synapse with a connection strength of 2000. The first and second graphs show the emitting neuron's membrane voltage and synaptic current. In graphs three and four the same variables are recorded for the receiving neuron. The emitting neuron is externally excited with a connection strength of 2000, achieving excitation above the threshold, leading to the generation of an action potential, which is received by the second neuron.

connections canceling each other out .[AB97] [SN94] To model this behavior, the network is structured into an excitatory and inhibitory population, with experiments showing that the balance is around 80% excitatory to 20% inhibitory. Projections in the network are formed inside the populations and between them, the inhibitory population has a higher connection strength to the excitatory population to counter the difference in number. Additionally, both populations are connected to an external noise generator. (Fig: 3.6). In [Bru00] an analysis of these networks is provided, with proposed ways to calculate the network parameters.
We used an existing implementation of a balanced network implemented for NEST[wwwb] that is based on the values proposed by Brunel and converted it for SpiNNaker with the generated model. The script with the chosen values can be seen in A.1, the result in figure 3.7. Compared to the activity domains identified in [Bru00], this network is in the domain of synchronous regular (SR) domain, with tendencies to the asynchronous regular (AR) domain at the end of the simulation. This can be identified based on the visible clusters of neurons along the neuron index 4000 and the periodic activity pattern strongly visible to 400 ms and slowly fading out from there on. This suggests that the balance is not yet correct, with the excitatory population overpowering the inhibitory population. Because this test already showed that the neuron model works in larger networks, we did not spend more time fine-tuning the parameters to achieve a stable balanced network.

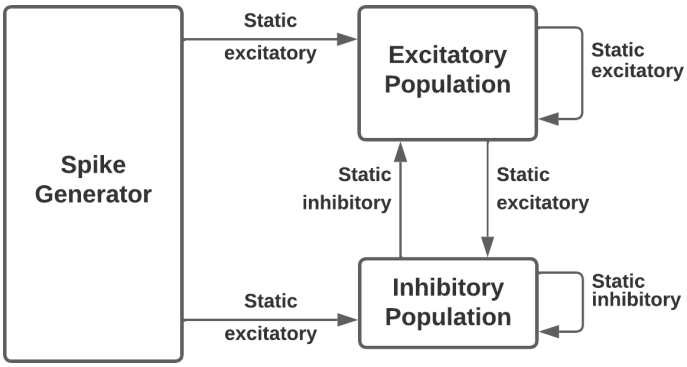

Figure 3.6: Balanced network structure (adapted from [AB97][Fig. 1])

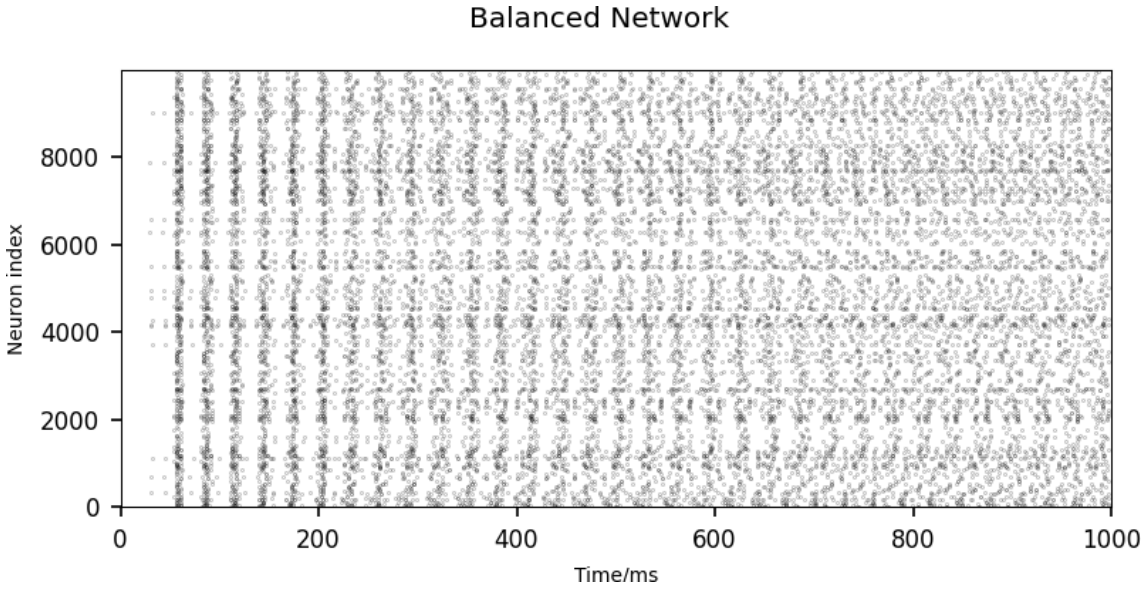

Figure 3.7: Balanced network test on SpiNNaker with generated neuron model. The graph shows all spike events that occurred throughout the simulation. Each neuron is indicated by its identification number, shown on the y-axis.

3.5.1 Synapse validation

Even though the synapse implementation was not ready for validation, this chapter presents three tests to verify the model. These are similar to the test proposed for the neuron to give a better intuition for the values and to reduce the implementation overhead. Relevant attributes of the synapse to check are the application of the weight to the spike and the plastic behavior. To confirm the correctness, a comparison to NEST is recommended. For the first test, we propose to set up the neuron chain example but with the new synapse implementation. Recording the respective synaptic current in the receiving neuron gives insight into the strength of the connection. Assuming the current was recorded as 0 in the previous steps, which is the default case, the recorded peek of the synaptic current is equal to the connection strength.
The second test reuses the neuron chain network, but instead of generating only one spike in the first neuron, we continue until the receiving neuron also generates a spike. When triggering one last spike in the first neuron, this updates the synapse dynamics so that the

changes to the connection strength can be recorded in the synaptic current of the receiving neuron. This test confirms that the plasticity mechanisms are working. For this test case, a comparison to NEST is a helpful addition to confirm the values.
Lastly, balanced networks can also be set up with plastic synapses, which makes for an interesting test to confirm the function of the synapse in larger networks.

Chapter 4

Discussion

This chapter will discuss the results of the extension process and provide some ideas to tackle encountered problems and possible improvements identified throughout the writing of this thesis.

4.1 Extension Process

We already concluded that SpiNNaker is a compatible target for NESTML (see chapter 3.1). This proved throughout the development of the extension, as no significant problems were encountered. The results presented in chapter 3.5 confirm this, as the implementation behaves equivalently to the existing implementation for NEST. The minor deviations measured were explainable due to the design differences in the hardware. Further tests have to confirm that the same is true for the synapse implementation, but from the analysis, this seems likely. Before this, the synapse template needs to be fully implemented. At the time of writing, the necessary API in the C source file template was identified and implemented to the point of successful compilation. For now, the problem regarding the traces is fixed in so far that it works with the iaf_psc_exp model included in NESTML, but a solution for this must be found in the future. Two possibilities are presented below in the paragraph "Handling traces" The PyNN interface still needs to be implemented, but the API for this is documented through the inherited classes.

Handling traces Relying on the naming convention to discern the trace is not a stable solution applicable for production, as the user manually sets the name of the trace variable. Instead, we propose a solution similar to a concept already implemented for NEST, which allows adding a tag to the variable declaration. This is already defined for the delay and weight parameters, which are standard variables necessary in the models. The tag is added with a whitespace after the variable declaration in the form of `@nest::weight` and `@nest::delay`. Following this concept, tags in the form of `@spinnaker::pre_trace` and `@spinnake::post_trace` could be added. A disadvantage of this approach is the additional complexity for the user, but as this already was added for NEST, the increase would be slight. An alternative solution could be to specify the trace names as parameters to the build process. This also is already applied for NEST to identify the post trace that has to be moved to the neuron. Currently, this references the port instead of the trace, while the trace is determined from the defined dynamics during code generation. This

probably is the better solution, as it is less reliant on the user and can be achieved with already implemented methods.

4.2 Improvements

Improvements to the templates In chapter 2.3.3, we discussed how a transformer is used to move the trace of the post-synaptic spikes to the neuron model when generating for the NEST target. NEST uses this strategy to reduce memory duplication. This may also apply to SpiNNaker as it uses a single binary for the neuron and synapse. Unlike the trace of the pre-synaptic neuron, the post-trace is stored in the DTCM, so it would be possible to reference it from the synapse. This could be a future improvement when all templates are fully implemented.

Improvements to SpiNNaker During the extension process, we found two ways that may improve the SpiNNaker software stack. A problem we encountered during the development was the debug process, which sometimes lacked the hints to be effective. For example, we encountered an error from a spelling mistake in a filename. The error message did not hint at this. Instead, it reported that it expected a different value at some address. Debugging in an embedded system is a difficult problem, especially on custom hardware like this, but this may be interesting to tackle in future revisions.
Another improvement to the usability of the software stack could be the addition of an on-the-fly update to the neuron and synapse models without altering the network structure. As the same mapping would be used for the network, no new mapping phase would be necessary, reducing the setup time. During the development, each time we wanted to test our often small changes to the model, we had to wait for the board to be reinitialized with the same network and a slightly different model. This could also be a good improvement for simulation quality, ensuring that tests with different models are conducted with the same network layout, leading to less interference in the results from different network layouts.

Chapter 5

Conclusion

To conclude this thesis, we will go through a short review of the discussed topics, results and future work. After the introduction to the topic and the formulation of the research question in chapter 1, we laid out the fundamentals for the topic of this thesis, starting with the necessary domain specific information in chapter 2.1 and introducing SNNs as an abstraction of the brain used in computational neuroscience. This is followed up with the discussion of DSL languages (chapter 2.2), a concept used to reduce the complexity overhead of programming encountered due to the increasing use of software in different research domains. With this knowledge, we provide a detailed look into NESTML, a DSL used in neuroscience(2.3). During this discussion, we laid out the components of NESTML which will be necessary for the extension process. The last part of the introduction focuses on neuromorphic hardware (chapter 2.4) and SpiNNaker (chapter 2.5), the target for the extension process. We provide information about the nature of neuromorphic hardware and discuss their use. With SpiNNaker as an example, we go through the hardware and software solutions used in SpiNNaker, to gain the insights relevant for the extension process. This follows in chapter 3. We go through the steps taken to achieve the extension of NESTML to SpiNNaker. The implementation is tested against NEST as a known good target. Finally, we discussed the results in chapter 4, concluding that the extension of NESTML to SpiNNaker is possible and that we achieved this for the neuron model, with synapse models also viable, but in need of further development. Furthermore, we discussed some problems encountered throughout the development, adding some thoughts for possible solutions

In future work, the synapse models needs to be finalized to fully support SpiNNaker. For this, a closer analysis of the problems encountered with referencing of the traces is necessary, looking into the suggested solutions. With regard to the neuron, the possibility of using the optimizations seen in NEST also for SpiNNaker needs further investigation. Apart from this, further targets for NESTML need to be identified and implemented. Based on the findings during the extension process, we suggest to use the neuron abstraction and simulation strategy used to hint at new targets.

Bibliography

[AA22] Astropy Collaboration and Astropy Project Contributors. The Astropy Project: Sustaining and Growing a Community-oriented Open-source Project and the Latest Major Release (v5.0) of the Core Package. , 935(2):167, August 2022.

[AB97] D. J. Amit and N. Brunel. Model of global spontaneous activity and local structured activity during delay periods in the cerebral cortex. *Cerebral Cortex*, 7(3):237–252, 04 1997.

[ACG+09] F. A. C. Azevedo, L. R. B. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. L. Ferretti, R. E. P. Leite, W. J. Filho, R. Lent, and S. Herculano-Houzel. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5):532–541, 2009.

[AGH05] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[ASP22] G. Azkona and R. Sanchez-Pernaute. Mice in translational neuroscience: What R we doing? *Progress in Neurobiology*, 217:102330, 2022.

[Bac78] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, aug 1978.

[Bru00] N. Brunel. Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *Journal of computational neuroscience*, 8:183–208, 2000.

[CCGC+14] R. C. C. Cannon, R. Gleeson, S. Crook, G. Ganapathy, B. Marin, E. Piasini, and R. Angus Silver. LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Frontiers in Neuroinformatics*, 8, 2014.

[CM08] A. Citri and R. C. Malenka. Synaptic Plasticity: Multiple Forms, Functions, and Mechanisms. *Neuropsychopharmacology*, 2008.

[CR07] E. Crivellato and D. Ribatti. Soul, mind, brain: Greek philosophy and the birth of neuroscience. *Brain Research Bulletin*, 71(4):327–336, 2007.

[DBE+09] A. Davison, D. Brüderle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2, 2009.

[DSL+18] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, Sri H. C., G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro*, 38(1):82–99, 2018.

[EFK54] J. C. Eccles, P. Fatt, and K. Koketsu. Cholinergic and inhibitory synapses in a pathway from motor-axon collaterals to motoneurones. *The Journal of physiology*, 126(3):524, 1954.

[ERW07] W. Elmenreich, M. Rosenblatt, and A. Wolf. Fixed Point Library Based on ISO/IEC Standard DTR 18037 for Atmel AVR Microcontrollers. In *2007 Fifth Workshop on Intelligent Solutions in Embedded Systems*, pages 101–113, 2007.

[FB20] S. Furber and P. Bogdan. *SpiNNaker: A Spiking Neural Network Architecture*. Boston-Delft: now publishers, 2020.

[Fow10] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[GD07] M. Gewaltig and M. Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007.

[GFCA21] K. Grewal, J. Forest, B. P. Cohen, and S. Ahmad. Going Beyond the Point Neuron: Active Dendrites and Sparse Representations for Continual Learning. *bioRxiv*, 2021.

[Gra59] E. G. Gray. Axo-somatic and axo-dendritic synapses of the cerebral cortex: an electron microscope study. *Journal of anatomy*, 93(Pt 4):420, 1959.

[Heb05] D. O. Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[Jor22] T. J. Jorgensen. Is the human brain a biological computer? 2022.

[Jö13] S. Jörges. Construction and Evolution of Code Generators : A Model-Driven and Service-Oriented Approach, 2013.

[Ker08] Kerr, J. N. D. and Denk, W. Imaging in vivo: watching the brain in action. *Nature Reviews Neuroscience*, 2008.

[KR06] B. W. Kernighan and D. M. Ritchie. *The C programming language*. 2006.

[LBB+23] C. A. P. Linssen, P. N. Babu, M. A. Benelhedi, J. M. Eppler, B. Rumpe, and A. Morrison. NESTML 5.3.0, June 2023.

[LBME21] C. A. P. Linssen, P. N. Babu, A. Morrison, and J. M. Eppler. ODE-toolbox: Automatic selection and generation of integration schemes for systems of ordinary differential equations. Dec 2021.

[Mah92] M. Mahowald. VLSI analogs of neuronal visual processing: a synthesis of form and function. 1992.

[MBD+15] E. Muller, J. A. Bednar, M. Diesmann, M. Gewaltig, M. Hines, and A. P. Davison. Python in neuroscience. *Frontiers in Neuroinformatics*, 9, 2015.

[Mea90] C. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636, 1990.

[MMG+05] A. Morrison, C. Mehring, T. Geisel, A. Aertsen, and M. Diesmann. Advancing the Boundaries of High-Connectivity Network Simulation with Distributed Computing. *Neural Computation*, 17(8):1776–1801, 08 2005.

[Ope23] OpenAI. GPT-4 Technical Report, 2023.

[Par13] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PBC+22] C. Pehle, S. Billaudelle, B. Cramer, J. Kaiser, K. Schreiber, Y. Stradmann, J. Weis, A. Leibfried, E. Müller, and J. Schemmel. The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity. *CoRR*, abs/2201.11063, 2022.

[PRB+16] D. Plotnikov, B. Rumpe, I. Blundell, T. Ippen, J. M. Eppler, and A. Morrison. NESTML: a modeling language for spiking neurons. March 2016.

[PRP+18] K. Perun, B. Rumpe, D. Plotnikov, G. Trensch, J. M. Eppler, I. Blundell, and A. Morrison. Reengineering NestML With Python And Monticore. Technical report, 2018.

[RBB+18] O. Rhodes, P. A. Bogdan, C. Brenninkmeijer, Simon D., D. Fellows, A. Gait, D. R. Lester, M. Mikaitis, L. A. Plana, A. G. D. Rowley, A. B. Stokes, and S. B. Furber. sPyNNaker: A Software Package for Running PyNN Simulations on SpiNNaker. *Frontiers in Neuroscience*, 12, 2018.

[RBD+19] A. G. D. Rowley, C. Brenninkmeijer, S. Davidson, D. Fellows, A. Gait, D. R. Lester, L. A. Plana, O. Rhodes, A. B. Stokes, and S. B. Furber. SpiNNTools: The Execution Engine for the SpiNNaker Platform. *Frontiers in Neuroscience*, 13, 2019.

[RHK21] B. Rumpe, K. Hölldobler, and O. Kautz. *MontiCore: Language Workbench and Library - Handbook*. 2021.

[RLH+17] B. Rueckauer, I. Lungu, Y. Hu, M. Pfeiffer, and S. Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in neuroscience*, 11:682, 2017.

[Ron08] A. Ronacher. *Jinja2 Documentation*. 2008.

[RSRB66] W. Rall, G. M. Shepherd, T. S. Reese, and W. M. Brightman. Dendrodendritic synaptic pathway for inhibition in the olfactory bulb. *Experimental Neurology*, 14(1):44–56, 1966.

[SN94] M. N. Shadlen and W. T. Newsome. Noise, neural codes and cortical organization. *Current Opinion in Neurobiology*, 4(4):569–579, 1994.

[Str00] B. Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

[TUM00] M. Tsodyks, A. Uziel, and H. Markram. Synchrony generation in recurrent networks with frequency-dependent synapses. *Journal of Neuroscience*, 20(1):RC50, 2000.

[vARS+18] S. J. van Albada, A. G. Rowley, J. Senk, M. Hopkins, M. Schmidt, A. B. Stokes, R. Lester, D, M. Diesmann, and S. B. Furber. Performance Comparison of the Digital Neuromorphic Hardware SpiNNaker and the Neural Network Simulation Software NEST for a Full-Scale Cortical Microcircuit Model. *Frontiers in Neuroscience*, 12, 2018.

[vMSB10] J. J. van Middendorp, G. M. Sanchez, and A. L. Burridge. The Edwin Smith papyrus: a clinical reappraisal of the oldest known document on spinal injuries. *Eur Spine J*, 2010.

[vN86] J. von Neumann. *The Computer and the Brain*. Yale University Press, 1986.

[VRD09] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[vVS96] C. van Vreeswijk and H. Sompolinsky. Chaos in Neuronal Networks with Balanced Excitatory and Inhibitory Activity. *Science*, 274(5293):1724–1726, 1996.

[WC82] J. R. Wolff and B. M. Chronwall. Axosomatic synapses in the visual cortex of adult rat. A comparison between GABA-accumulating and other neurons. *Journal of neurocytology*, 11:409–425, 1982.

[WKP11] C. Watson, M. Kirkcaldie, and G. Paxinos. *The Brain*. Academic Press, 2011.

[wwwa] Fixed point arithmetic *https://vanhunteradams.com/FixedPoint/FixedPoint.html*. Accessed: 05.09.2023.

[wwwb] NEST balanced network *https://nest-simulator.readthedocs.io/en/stable/auto_examples/brunel_exp_multisynapse_nest.html*. Accessed: 05.09.2023.

[wwwc] NEST Website *https://www.nest-simulator.org/*. Accessed: 05.08.2023.

[wwwd] NVIDIA GeForce RTX 4090 *https://www.nvidia.com/de-de/geforce/graphics-cards/40-series/rtx-4090/*. Accessed: 04.08.2023.

[wwwe] PyNN *https://neuralensemble.org/PyNN/*. Accessed: 05.09.2023.

[wwwf] Segger Blog *https://blog.segger.com/floating-point-face-off-part-2-comparing-performance/*. Accessed: 05.09.2023.

[wwwg] SpiNNaker Website *https://apt.cs.manchester.ac.uk/projects/SpiNNaker/*. Accessed: 05.08.2023.

[Zuc96] R. S. Zucker. Exocytosis: a molecular and physiological perspective. *Neuron*, 17(6):1049–1055, 1996.

Appendix A

```
neuron iaf_psc_exp:
    state:
        r integer = 0     # Counts number of tick during the refractory period
        V_m mV = E_L      # Membrane potential
    equations:
        kernel I_kernel_inh = exp(-t / tau_syn_inh)
        kernel I_kernel_exc = exp(-t / tau_syn_exc)
        inline I_syn pA = convolve(I_kernel_exc, exc_spikes)
                          - convolve(I_kernel_inh, inh_spikes)
        V_m' = -(V_m - E_L) / tau_m + (I_syn + I_e + I_stim) / C_m
    parameters:
        C_m pF = 250 pF            # Capacitance of the membrane
        tau_m ms = 10 ms           # Membrane time constant
        tau_syn_inh ms = 2 ms      # Time constant of inh synaptic current
        tau_syn_exc ms = 2 ms      # Time constant of exc synaptic current
        t_ref ms = 2 ms            # Duration of refractory period
        E_L mV = -70 mV            # Resting potential
        V_reset mV = -70 mV        # Reset value of the membrane potential
        V_th mV = -55 mV           # Spike threshold potential
        I_e pA = 0 pA              # constant external input current
    internals:
        RefractoryCounts integer = steps(t_ref) # refractory time in steps
    input:
        exc_spikes pA <- excitatory spike
        inh_spikes pA <- inhibitory spike
        I_stim pA <- continuous
    output:
        spike
    update:
        if r == 0: # neuron not refractory, so evolve V
          integrate_odes()
        else:
          r = r - 1 # neuron is absolute refractory
        if V_m >= V_th: # threshold crossing
          r = RefractoryCounts
          V_m = V_reset
          emit_spike()
```

Figure A.1: Parameters for balanced network [TUM00]

Appendix B

```
import pyNN.spiNNaker as p
from pyNN.utility.plotting import Figure, Panel
import matplotlib.pyplot as plt
from python_models8.neuron.builds.iaf_psc_exp_nestml import
↪  iaf_psc_exp_nestml

dt = 0.1  # the resolution in ms
simtime = 1000.0  # Simulation time in ms
delay = 1.5  # synaptic delay in ms
g = 4.0  # ratio inhibitory weight/excitatory weight
eta = 2  # external rate relative to threshold rate
epsilon = 0.05  # connection probability
order = 2500
NE = 4 * order  # number of excitatory neurons
NI = 1 * order  # number of inhibitory neurons
N_neurons = NE + NI  # number of neurons in total
CE = int(epsilon * NE)  # number of excitatory synapses per neuron
CI = int(epsilon * NI)  # number of inhibitory synapses per neuron
C_tot = int(CI + CE)  # total number of synapses per neuron
tauMem = 20.0  # time constant of membrane potential in ms
theta = 20.0  # membrane threshold potential in mV
J = 0.1  # postsynaptic amplitude in mV
neuron_params = {
    "C_m": 0.7,
    "tau_m": tauMem,
    "t_ref": 2.0,
    "E_L": 0.0,
    "V_reset": 0.0,
    "V_th": theta,
    "tau_syn_exc": 0.4,
    "tau_syn_inh": 0.4,
}
J_ex = J  # amplitude of excitatory postsynaptic current
J_in = -g * J_ex  # amplitude of inhibitory postsynaptic current
nu_th = theta / (J * CE * tauMem)
nu_ex = eta * nu_th
p_rate = 1000.0 * nu_ex * CE
```

```
p.setup(dt)

nodes_ex = p.Population(NE, iaf_psc_exp_nestml(**neuron_params))
nodes_in = p.Population(NI, iaf_psc_exp_nestml(**neuron_params))

noise = p.Population(
        20, p.SpikeSourcePoisson(rate=p_rate),
        label="expoisson", seed=3)
noise.record("spikes")
nodes_ex.record("spikes")
nodes_in.record("spikes")

exc_conn = p.FixedTotalNumberConnector(CE)
inh_conn = p.FixedTotalNumberConnector(CI)

p.Projection(nodes_ex, nodes_ex, exc_conn, receptor_type="exc_spikes",
            synapse_type=p.StaticSynapse(weight=J_ex, delay=delay))
p.Projection(nodes_ex, nodes_in, exc_conn, receptor_type="exc_spikes",
            synapse_type=p.StaticSynapse(weight=J_ex, delay=delay))
p.Projection(nodes_in, nodes_ex, inh_conn, receptor_type="inh_spikes",
            synapse_type=p.StaticSynapse(weight=J_in, delay=delay))
p.Projection(nodes_in, nodes_in, inh_conn, receptor_type="inh_spikes",
            synapse_type=p.StaticSynapse(weight=J_in, delay=delay))

p.Projection(noise, nodes_ex, exc_conn, receptor_type="exc_spikes",
            synapse_type=p.StaticSynapse(weight=J_ex, delay=delay))
p.Projection(noise, nodes_in, exc_conn, receptor_type="exc_spikes",
            synapse_type=p.StaticSynapse(weight=J_ex, delay=delay))

p.run(simtime=simtime)

exc_spikes = nodes_ex.get_data("spikes")

Figure(
    # raster plot of the presynaptic neuron spike times
    Panel(exc_spikes.segments[0].spiketrains, xlabel="Time/ms",
     ↪  xticks=True,
          yticks=True, markersize=0.2, xlim=(0, simtime)),
    title="Balanced Network",)

plt.show()
```

Figure B.1: Network setup for balanced network