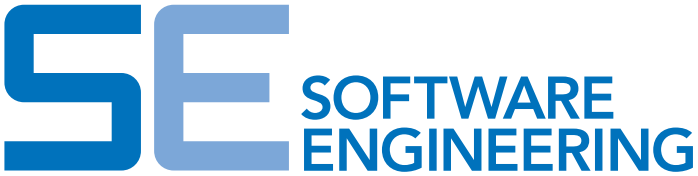
Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering



Masterarbeit

NESTML – Creating a Neuron Modeling Language and Generating Efficient Code for the NEST Simulator with MontiCore

Tammo Ippen
Matrikel-Nr.: 281214

Aufgabenstellung:	Prof. Dr. B. Rumpe
Zweitgutachter:	Prof. Dr. M. Diesmann
Betreuer:	Dipl.-Inform. Dipl.-Wirt. Inform. Markus Look
	Dr. Jochen M. Eppler

Aachen, den 17. Dezember 2013

Erklärung

Ich versichere, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Aachen, den 17. Dezember 2013

Kurzfassung

Neurowissenschaftler verwenden Computersimulationen als eine Möglichkeit, um das Gehirn und Hirnaktivität zu erforschen. Sie entwickeln und veröffentlichen zahlreiche Neuronen- und Synapsen-Modelle mit unterschiedlichem Detailgrad die in Simulationen von einzelen Neuronen oder großen biologischen neuronalen Netzen verwendet werden. Neben Neuronen- und Synapsen-Modellen entstanden mehreren Simulatoren für neuronale Netzwerke mit unterschiedlichem Umfang und meist inkompatiblen Modellbeschreibungssprachen. Dies erschwert die Entwicklung und Veröffentlichung neuer Neuronen- und Synapsen-Modelle, sowie die Überprüfung und Verifikation von Ergebnissen zwischen Simulatoren, da jedes Modell für jeden Simulator implementiert und angepasst werden muss.

Diese Arbeit beschreibt das Design von NESTML und seine Entwicklung mit dem MontiCore Framework. NESTML ist eine erweiterbare Modellierungssprache spezifisch für die neurowissenschaftliche Domäne. Es ermöglicht die Modellierung von Punkt-Neuron-Modellen in einer sauberen und prägnanten Syntax. Ein zugehöriges Verarbeitungstool führt statische Analysen an NESTML Modellen durch, um programmatische Korrektheit zu überprüfen und Neurowissenschaftler bei der Entwicklung neuer Neuron-Modelle zu unterstützen. Desweiteren generiert das Tool effizienten Simulationscode für den NEST Simulator, sowie die komplette Infrastruktur für NEST Module, damit der generierte Code komfortabel compiliert und in NEST geladen werden kann. Dies reduziert den Arbeitsaufwand um Neuron-Modelle zu erstellen, zu pflegen und zu verteilen.

Abstract

Neuroscientists use computer simulations as one way to research the brain and brain activity. They developed and published numerous neuron and synapse models with different levels of detail to be used in simulations of single neurons or large biological neuronal networks. Besides the neuron and synapse models, the neuroscience community has developed several simulators with different scope and, mostly, incompatible model description languages. This makes it hard to develop and publish new neuron and synapse models and even harder to compare and verify findings across simulators, since the model must be implemented and adjusted for every simulator.

This thesis describes the design of NESTML and its development with the MontiCore framework. NESTML is an extendable modeling language for the neuroscience domain. It allows modeling spiking point neuron models in a clean and concise syntax. An associated processing tool performs static analysis on NESTML models to check for programmatic correctness and, thus, supports neuroscientists in creating new neuron models. Furthermore, it generates efficient code for the NEST simulator and the NEST module infrastructure, which allows to easily compile and load the generated code into NEST. This reduces the work to create and to maintain neuron models for NEST and, by adding more simulator targets in the future, across simulators.

Task Description

The task of this master thesis is to design and develop an extendable domain specific language to model components of neuronal networks, i.e. neurons and synapses, and to generate efficient code from these models for neuronal simulators, like the NEural Simulation Tool (NEST). The purpose of this language is to enable neuroscientists to easily define and exchange newly developed models in a standardized way and to test the models in different simulation tools.

During the design phase existing neuron modeling languages, like NineML, are reviewed and current state-of-the-art in neuroscience research is examined. Additionally, a close cooperation with the NEST team in Jülich takes place, so that in extensive interviews requirements for the language are analyzed, domain specific knowledge can be incorporated into the language and its usability can be tested.

To implement the modeling language, the DSL development framework MontiCore is used. It allows specifying an extendable domain specific language and generates the lexer and parser for the language. Context conditions, defined in the design phase, have to be implemented. They assure, that only valid models are accepted. The to be implemented code generators have to generate efficient code, since simulations of neuronal networks are a computationally complex task, and the code needs to easily integrate into the existing simulation tools. Finally, MontiCore can generate editor support for syntax highlighting, which will assists future users of the language.

The focus of the thesis is, on the one hand, to design and develop the modeling language for neuron models, to implement neuron specific context conditions and to generate efficient code for a NEST plugin., and on the other hand, to design the language and the implementation to be easily extendable. It is planed to model synapse models with that language and to generate code for other simulators in the future.

Contents

List of Figures

Chapter 1

Introduction

Understanding the brain and neuronal activity is a difficult task. The human brain, for example, consists of up to 10^{12} neurons; each receives input from approximately 10^4 other neurons and generates output to about as many. A single cubic millimeter of cortex contains at least 10^5 cells with about 10^9 connections. Measurements of the biochemical and electrophysiological processes in neurons and brain areas in vitro and in vivo builds the foundation for computer simulations of neural tissue and neural circuits. These simulations allow the detailed investigation of neuronal activity and the testing of hypotheses about dynamics and function of the modeled system.

To carry out such simulations the field of *computational neuroscience* developed various dedicated and specialized programs to execute models of parts of the brain with different levels of detail – from the behavior of a single neuron to the interaction of biological neuronal networks of increasing size [Bre+07]. To achieve results that correspond to the activity in real brains detailed models of neurons and synapses are needed. Over the years the neuroscience community developed numerous models of both, each with a different level of detail or with the focus on different biophysical details [GK02; MDG08].

1.1 Motivation

The variety of neuron and synapse models leads to assets and drawbacks. On the one hand, neuroscientists can choose models that fit best to the experiment they want to conduct through computer simulation. On the other hand, each neuron or synapse model that is relevant for a simulator has to actually be implemented and optimized for the simulator and has to be maintained throughout the lifetime of the simulator.

Additionally, neuroscientists regularly develop new models that incorporate new findings or that better suit their experiment. Publishing those models includes their mathematical properties and possibly a concrete implementation for the simulator that the neuroscientist used. To reproduce the findings in another simulator or to conduct own experiments with the new model requires the model to be implemented for the other simulator. The neuroscientist has to become familiar with the programming language and the framework of the other simulator and has to adjust the model, so that it behaves the same as the published one. This alone can become a cumbersome task,

since modern simulators can be highly complex programs that make use of various parallel computing techniques to process the enormous neuronal networks, which are needed in order to have realistic simulations of parts of the brain.

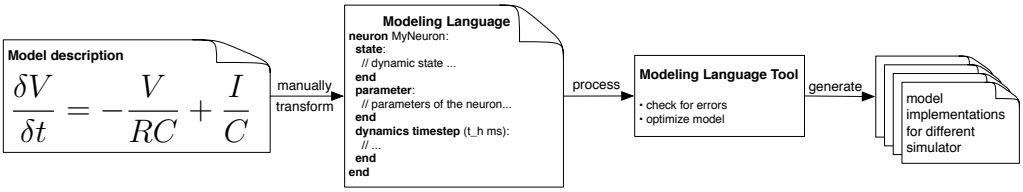


Figure 1.1: A desirable workflow for developing new neuron or synapse models. From the mathematical description create a specification in a dedicated modeling language. Let the corresponding tool support the model developer by analyzing and optimizing the model and pointing to errors. Finally, let the tool generate simulator specific implementations of the model.

From the point of view of a neuroscientist or of a developer for a neuronal network simulator, it would be preferable that a neuron or synapse model is specified once and can be reused in different simulators. A dedicated modeling language or domain specific language can be used to completely specify a neuron or synapse model in a standardized way [Cro+12]. A corresponding tool can then process the model, support the model developer by pointing out errors and generate specialized implementations of that model for different simulators. Figure 1.1 illustrates this workflow.

Such a modeling language and a corresponding tool can support neuroscientists and developer of simulator in several ways. If the model description is compact and expressive, it is well suited to be published along with the mathematical description. Neuroscientist then only have to use the tool to generate the implementation for their preferred simulator. When all models of a simulator are specified in this language and some code in the simulator changes, only the code generation of the tool for that specific simulator has to be adjusted and new implementations of the models can be generated.

1.2 Goals

The goals of this thesis are:

- Design a modeling language that is compact, concise, expressive and easy to learn. This language should allow specifying models with enough detail, so that simulator specific implementations can be generated out of models.

With respect to the scope of this thesis, the language should at least be able to model point neurons (Section 2.2). Extendibility for more sophisticated neuron models and for synapse models should be considered.

- Develop a corresponding tool for the modeling language that can process model descriptions, analyze them for computational correctness and generate model implementations for neuronal network simulators.

The processing tool will be developed with the MontiCore framework [Grö+08; Kra09] and besides others, includes the implementation of a symbol table and various context conditions. These context conditions check models for computational correctness and help model developers to create their neuron models by supplying helpful error messages. The code generation will be limited to the *NEST simulator* [GD07], for now, but future versions of the tool should support more simulators. Hence, the working name of this language is *NESTML* – NEST Modeling Language.

1.3 Thesis Outline

Chapter 1, *Introduction*, introduces the topic of this thesis and motivates the usage of a domain specific language to describe neuron models for the computational neuroscience.

Chapter 2, *Fundamentals*, gives the basic knowledge and introduces tools that are used in order to create NESTML. This includes an explanation of the structure of the brain and the function of neurons. Furthermore, formal languages, such as programming languages and domain specific languages, and the processing of such languages are explained. In particular, the framework MontiCore is introduced.

Chapter 3, *Requirements Analysis*, identifies and summarizes the functional and non-functional requirements for this language.

Chapter 4, *Design*, shows the design of NESTML and its related languages. This includes a description of the syntax of each language as well as its semantics and context conditions. Different design choices are identified and explained.

Chapter 5, *Implementation*, depicts the implementation of NESTML and its related languages with MontiCore. It describes the project layout and general implementation strategies for the symbol table, type calculations and the code generation.

Chapter 6, *Application Scenario*, contains a short introduction into the usage of the language and the tool itself. On the example of the famous *integrate-and-fire* neuron a full implementation of a neuron model in NESTML is shown, alongside with the generated code for the NEST simulator.

Chapter 7, *Related Work*, examines languages and processing tools that are related to NESTML.

Chapter 8, *Conclusion*, concludes this thesis and gives an outlook on future work.

Appendix A, *Abbreviations*, explains the used abbreviations in diagrams.

Appendix B, *Language Grammars*, contains the MontiCore grammar definitions for the developed languages.

Appendix C, *Example NESTML neuron*, contains an example implementation of the *integrate-and-fire* neuron model in NESTML.

Appendix D, *Generated NEST Code (neuron)*, contains the C++ header and implementation file that are generated from the neuron model in Appendix C.

Appendix E, *Generated NEST Code (unit)*, contains an example UnitDSL model with the corresponding generated C++ code.

Chapter 2

Fundamentals

This chapter introduces the fundamental background that is used in this thesis. An explanation of the basic structure and functionality of neurons and synapses is contained in Section 2.1. Section 2.2 describes different simulators for biological neuronal networks. These two sections are roughly based on the descriptions in Eppler [Epp10]. Section 2.3 describes the building blocks of programming languages and differentiates between domain specific and general purpose languages. Section 2.4 contains the general structure and workflow of tools that process those programming languages – the so-called *compilers*. Finally, Section 2.5 introduces the development framework MontiCore for domain specific programming languages.

2.1 The Brain on a microscopic scale

Although the brains of different animals differ considerably on a macroscopic scale, the brain of vertebrates consist of the same building blocks, on a microscopic scale. In particular, these are nerve cells (*neurons*) that communicate via electric pulses (*action potentials* or *spikes*) over connections called *synapses*. Figure 2.1 shows the structure of a typical neuron. It consists of a cell body or *soma*, dendrites and the axon.

The *membrane* of a neuron maintains an electric potential – the *membrane potential* – via different biochemical processes. One process consists of *ion pumps* that transport specific types of ions and can be open or closed depending on the membrane potential. In the resting state (i.e. in the absence of external input) the membrane potential fluctuates around a *resting potential* (about -70 mV). Incoming action potentials from other cells lead to an excursion of the membrane potential of the cell. If the membrane potential reaches a certain threshold, the cell fires an action potential itself that travels along the axon. If the spike reaches a synapse it causes the release of chemical neurotransmitter into the synaptic cleft. After a spike, the neuron is inactive (*refractory*) for some milliseconds. The receptors on the membrane of the postsynaptic neuron register the spike and, depending on the type of this neuron and synapse, the receptors trigger a rise (*excitation*) or decline (*inhibition*) of the membrane potential.

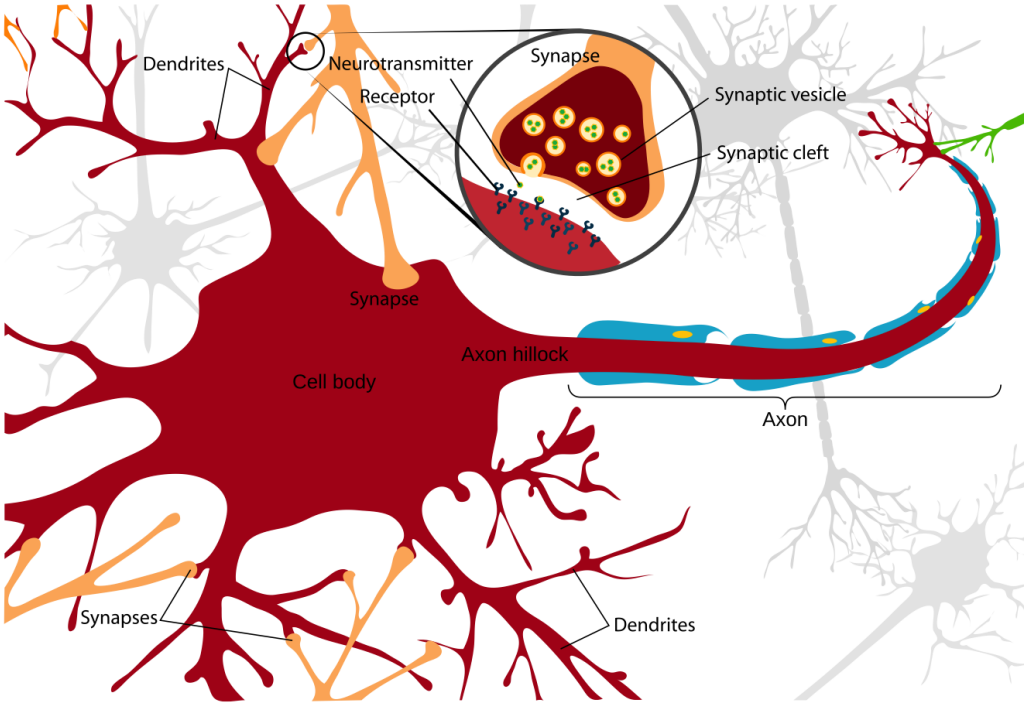


Figure 2.1: The structure of a vertebrate neuron. The red neuron receives input from the orange neuron and forwards action potentials to the green neuron. The round clipping shows a synaptic connection in detail. [Epp10]

2.2 Simulation of Neuronal Networks

Two different types of neuronal networks have to be distinguished: artificial and biological neuronal networks. *Artificial neuronal networks* assume that a brain function has many processing elements (the neurons) with weighted connections between them (the synapses). They are successfully applied in computer science, e.g. for pattern recognition, solving of optimization problems and machine learning. To approximate a target function with an artificial neuronal networks, a small network with a task oriented structure is created. An input vector is propagated through the network and processed by the neurons. The output can be compared to the target function and the weights of the connections can be modified in order to optimize the approximation. This adaption is often referred to as learning.

Artificial neuronal networks, however, can not be used to understand the brain, since they are not based on the properties of real neurons. Neuron models in *biological neuronal networks* are not solving a certain task, but replicate the behavior of real neurons. Different neuron models were developed with mathematical descriptions for certain aspects of real neurons, e.g. spiking behavior or membrane potential. These descriptions are based on observations and measurements in real neural systems.

Different fields of computational neuroscience require different level of detail for their neuron model. *Reaction-diffusion* models describe the interaction of molecules inside cells or at the cell membrane. *Compartment* models reconstruct real neurons by dividing them into a large number of electrical compartments. The compartments describe the propagation of action potentials and the dynamics of parameters like the mem-

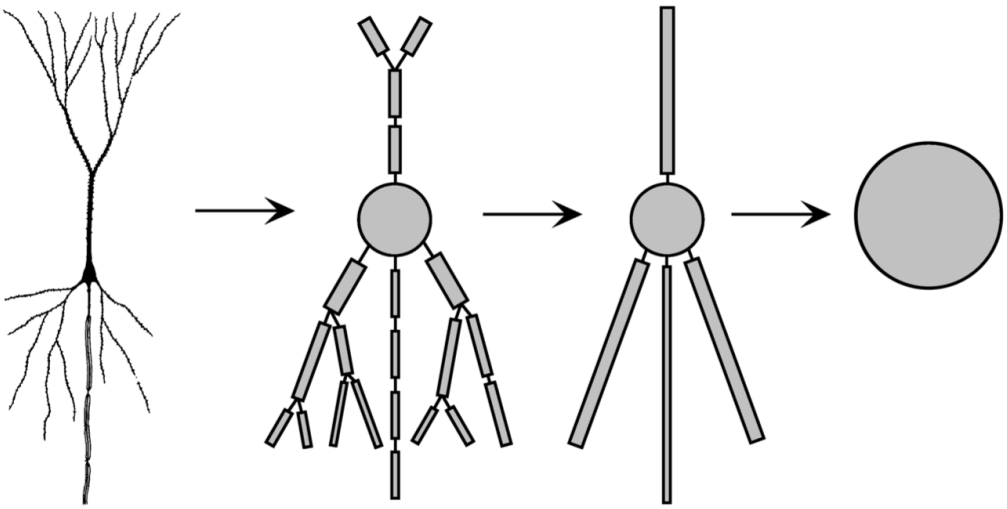


Figure 2.2: The abstraction levels of neuron models. The real neuron is modelled as a compartment model with a variable number of compartments, each representing a separate region with its own membrane potential. The simplest representation of the real neuron has a single compartment – the point neuron. [Epp10]

brane potential in the dendrites and axon of a cell via equations from cable theory. *Point neuron models* only have a single compartment and are modeled using mathematical equations for the membrane potential and for spike generation. *Population* models describe neuronal networks on the functional level. They model whole neuron populations as single entities and describe their behavior upon its overall input and output. *Field* models are basically population models that take the spatial composition of populations into account. The derivation of the *integrate-and-fire* point neuron model is described in Chapter 6 in detail.

Synapses play an important role in the information processing themselves. They differ by the neurotransmitter they use, their strength (*weight*) and by the time they need to transmit the signal (*delay*, usually in the order of 1 ms). Depending on their usage, the strength of a synapse can change, they can die or new synapses can grow. If synaptic plasticity occurs on the scale of milliseconds, seconds or minutes it is called short-term depression/ potentiation (STD/STP). If it occurs in the range of hours and days it is called long-term depression/ potentiation (LTD/LTP) and can last for months or even years. Hence, long-term changes are the basis of learning and memory.

A biological neuronal network of point neurons is described as a graph with nodes with different neuron models, and edges with different synaptic properties. The communication between nodes is based in spikes. Depending on the size of those networks they can be analyzed either analytically or by computer simulations. In the subsequent sections some simulators for biological neuronal networks with their key characteristics will be described.

NEST [DG02; Epp10; GD07] The *NEural Simulation Tool* is a framework that is optimized for simulating large, structured neuronal networks. The built-in neuron models are mostly spiking point neurons, but there is no restriction on the type of neuron model in general. The synapse models can either be static or can change

their weights according to synaptic plasticity rules. Networks in NEST are represented as weighted, directed graphs of nodes and connections between the nodes. The communication between nodes is based on events, e.g. spike or electric current events.

NEST is developed in an object-oriented style with a number of independent modules in C++. A built-in simulation language interpreter (SLI) is the primary interface for users to create neuronal networks and start the simulation. The simulation kernel implements all functionalities to define and simulate this neuronal network. It contains a base class for neuronal models with derived implementations of important models. New models can be developed by extending this base class and implementing the required abstract functions, e.g. the `update`-function. The development of a neuronal model is covered in Section 5.3.4 in depth. The network driver of the kernel manages the temporal update of all simulation elements. Basically, NEST simulates a neuronal network in a time driven strategy with the following algorithm:

```
1 t <- 0                                  // start simulation at time zero
2 WHILE t < T_Stop                        // perform simulation until T_Stop is reached
3   PARALLEL on all processes
4     deliver all events to designated node
5     advance current state S_{t-1} to S_t by calling update forall nodes
6   END
7   exchange new generated events between all processes
8   increment time t <- t+Delta           // the smallest connection delay in the network
9 END
```

Brian [GB08] Brian is a time driven simulator for spiking neuronal networks entirely written in the Python programming language. This makes it easy to learn and suitable for rapid prototyping of single compartment neuron models. It also supports multi-compartment models, but developing those models is more involved. The dynamics of neuron and synapse models are directly defined by their differential equations written in ordinary mathematical notation. Additionally, Brian has basic support for synaptic plasticity. To achieve the necessary performance even for larger networks Brian uses vectorized operations based on NumPy [Oli06] and optionally certain core routines are written in optimised C code. The left set of equations can directly be used in Brian to model leaky integrate and fire neurons, which in general are described by the right set of equations:

```
1 '''
2 dV/dt  = (ge+gi-(V+49*mV))/(20*ms) : volt
3 dge/dt = -ge/(5*ms)                 : volt
4 dgi/dt = -gi/(10*ms)                : volt
5 '''
```

$$\tau_m \frac{dV}{dt} = -(V - E_L) + g_e + g_i$$
$$\tau_e \frac{dg_e}{dt} = -g_e$$
$$\tau_i \frac{dg_i}{dt} = -g_i$$

NEURON [HC97] The *NEURON* simulator is capable of building and using "biological" neuron models with complex morphological and biophysical properties and many compartments, as well as more "artificial" point neuron models, like the integrate and fire neuron model. New neuron and synapse models and networks of them can be defined with a set of GUI tools or with the custom programming language *HOC*. Both methods can be combined, to take advantage of the strengths of both. The library of biophysical mechanisms, which can be used as building blocks for

neuron or synapse models, can be extended with descriptions in the *NMODL* language, whose syntax for expressing nonlinear algebraic equations, differential equations, and kinetic reaction schemes closely resembles familiar mathematical notation.

Others [Epp10] Besides the above described simulators and many "home grown" simulators of individual laboratories, the field of computational neuroscience has other important and popular programs and s few of them will be mentioned here. The *GEneral NEural SImulation System* (GENESIS) [BB98] is a simulation platform for neuronal systems of sub-cellular components, complex single neurons and large networks. The *Multiscale Object-Oriented Simulation Environment* (MOOSE) [RB08] extends GENESIS and can be used for large, detailed simulations in computational neuroscience and systems biology. Finally, *MCell* [Ker+08] is a modeling tool for realistic simulation of cellular signaling in the three-dimensional subcellular microenvironment in and around living cells.

In this thesis we concentrate on modeling and processing point neuron models and generating simulator code for the NEST simulator from these model descriptions.

2.3 Domain Specific and General Purpose Languages

Domain specific languages (DSL) are programming languages that are tied to a specific application domain, have a limited language scope and in most cases are not computationally complete [Kra09] or executable. The purpose of DSLs is to support domain experts to express concepts and models of their specific domain more clearly and declarative than with *general purpose languages* and do not focus on their solution. DSLs have high entry costs, since the language has to be created from ground up and developers have to learn the new language. They pay of, if they are heavily used afterwards, because they generally decrease the work of developers a lot.

General purpose languages (GPL) have a very general language scope and are computational complete, i.e. that everything, that can be computed, can be expressed in such a language. So all domain concepts can be modeled with a GPL, too, but the expressiveness of the concepts is limited to the language scope. For example, with object oriented languages like Java these domain concepts can be modeled with classes that contain appropriate methods and variables. Another benefit is, that compilers for these languages are availably and potential developers already know that language, but the work do express domain concepts in a GPL is generally higher than in a dedicated DSL.

It is possible to differentiate domain specific languages into *internal* and *external* DSLs [Gho10]. *Internal* DSLs are implemented inside another, general purpose language and thereby are a subset of the host language. This reduces the work to create such a language, since no separate compiler needs to be developed, but the host language limits its expressiveness. Examples for this kind language are *ScalaTest* which is a testing framework for the scala programming language [Ven13], *PyNEST* which is the Python interface to the NEST simulator [Epp+09], and *nemo* which is a neuron modeling language implemented in CHICKEN Scheme [Rai13], see Chapter 7.

External DSLs are completely new designed languages. This involves developing a separate compiler, but has the benefit, that all properties of the language can be defined freely, so that clarity and expressiveness of the resulting language can be superior to embedded DSLs. Prominent examples for this kind of language are the structured query language (SQL) for relational databases [Mül08], the hypertext markup language *HTML* [Ber+13] and the COmmon Business-Oriented Language *COBOL*, that can be seen as a DSL for business applications [Völ11].

All programming languages have several components that define their look, usage and meaning. These are fundamental components, such as concrete syntax, abstract syntax and their semantics, and more practical components, such as their type system, symbol tables and context conditions. In the following, a short introduction to these components is given, since NESTML needs all these components.

Concrete Syntax The concrete syntax describes, how the programming language presents itself to potential users. There are textual programming languages, graphical or mixture of them. Textual programming languages are often defined by grammars, that are used to create lexers and parsers [Völ11], see Section 2.4. Some editors can highlight their syntax, but basically the developer is working with plain text. Graphical programming languages need some kind of editor, with which graphical elements, like rectangles and circles, are manipulated. Examples for graphical programming languages are *Scratch* [Mal+04] and *blockly* [FS13]. They are often used for educational purposes.

Abstract Syntax The abstract syntax describes the internal representation of a language, which can be independent to the concrete syntax. Therefore, a given program written in a concrete syntax has to be translated into its abstract syntax, which is often modeled as a tree structure. Language-processing tools use this representation to analyze and manipulate a given program. The abstract syntax of a programming language can be developed separately from the concrete syntax, allowing for several concrete syntaxes translating into one abstract syntax, or the concrete and abstract syntax can be developed simultaneously, this makes sure, that there always is a concrete syntax for testing the language, and keeps both syntaxes consistent, see Section 2.5.

Semantics The semantics of a language describes the meaning of the language itself and of each element of the language. Just like the syntax of a programming language, its semantics can be exactly defined: the *denotational* semantics uses mathematical constructs that describe the meaning of each element. The *operational* semantics uses abstract machines and state transitions to explain semantics. Finally, *translational* semantics translates elements into a language with well-known semantics [Völ11]. With these semantics the output of a program with a given input can be evaluated.

Type System Most programming languages have several different types that describe the variables they are manipulating. These types can be primitive, like boolean values, integer or floating point numbers, and can become more complex with structs in C or classes in Java. Programming languages have type systems with different degrees of strictness, ranging form untyped languages, where variables have no type at all, like *Smalltalk* [SR91], over dynamically typed languages like

Python [Sum08], to strictly typed languages, like in *Haskell* [OGS08]. With the type system of a language, the type correctness of a program can be tested at compile time which reduces the amount of runtime errors the more strictly the language is typed.

Symbol Table The symbol table of a programming language relates a variable or type name with its contextual information, e.g. the type of the variable or the embedded variables and functions of a type. Symbol tables can be nested into so-called *namespaces*, so that the 'visibility' of a name can be restricted. The main purpose of a symbol table is to offer an interface to merge models, i.e. a model publishes all its usable symbols in its symbol table and other models can refer to those symbols, without the need to recompile the source code. Beside that, the symbol table supports checking of context conditions and code generation.

Context Conditions A program can conform to its concrete syntax, i.e. it is syntactically correct, but there still can be errors that can be caught in the static analyzes during compile time. These errors are checked with context conditions that analyze the abstract syntax of a program. The use of context conditions over syntax definitions can have different reasons: either the error can not be described with the syntax definition, since it involves context knowledge that can not be expressed with context-free grammars, e.g. a variable has to be declared before use, or describing the error in the syntax definition is not desired to keep the definition simple. It is beneficial, if as many errors as possible are caught during static analyzes, so that developers can fix these errors before they deploy their program.

2.4 Compiler

This section introduces the basic concepts of compilers for programming languages, focusing on textual languages, since in this thesis a textual DSL will be created, and discusses the steps of modern language processing. From the outside a compiler translates a high level program, i.e. the *source program*, into a lower level program, i.e. the *target program*, while outputting warnings and errors, if something is wrong with the source program. The target program can have any abstraction level: it could be machine code for a specific processor, code for another programming language or even optimized code for the source programming language.

Figure 2.3 is an overview of the components and workflow of a typical compiler. In real-world compilers these components do not need to be so clearly separated, but they are still present in some way. When the compiler gets a source program, it first passes the source code to the *lexer*. The lexer analyzes the sequence of characters and splits it, according to its lexical rules, into *tokens*. Thus, a token is the internal representation of a sequence of characters in the compiler. If a sequence of characters can not be matched with a lexical rule, the lexer reports an error to the *error handler*.

The purpose of the error handler is to gather all error reports of the static analyzes and produce meaningful error messages for the programmer. The more errors can be found during the different steps and the better error messages are provided, the more

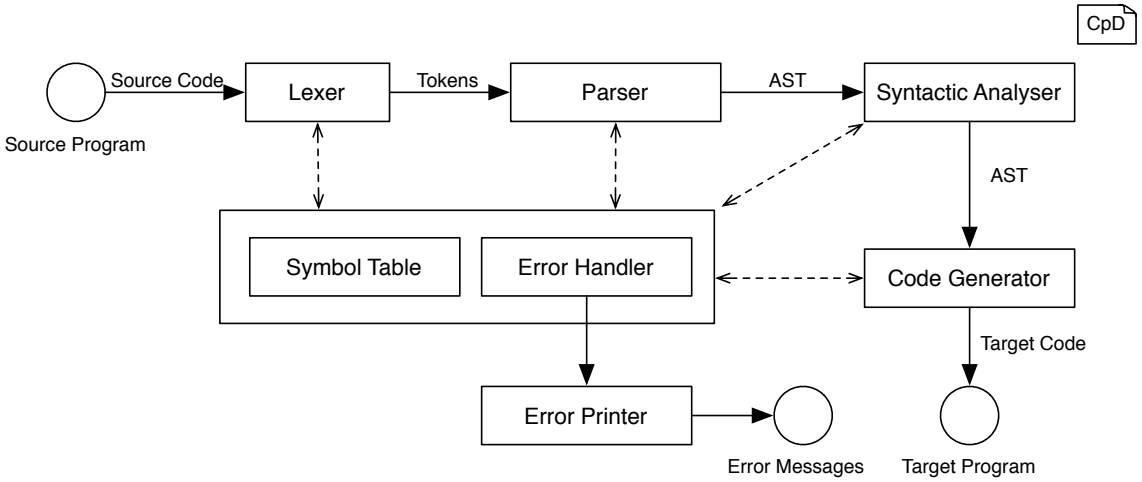


Figure 2.3: The basic architecture of a compiler. Circles denote inputs or outputs of the compiler, the boxes its components. Arrows denote the dataflow inside the compiler and their labels state, what data is transferred. The dashed arrows show other communication between components. [Mül08]

productive can a programmer correct his work. This means, that, as a general rule, the compiler should not stop upon one error, but should continue to try to find more errors.

The *parser* takes the stream of tokens from the lexer and tries to build a tree on top of these tokens. This tree is called *abstract syntax tree* (AST) and is built according to rules defined by the abstract syntax. If tokens are in the wrong order or missing, the parser again reports errors to the error handler. The work of the lexer and parser will be demonstrated with a small example language, called ab-language. The language allows source programs, that match the regular expression `(a|b)+`, e.g. “a”, “b”, “ba” and “aaabbbababba” are programs of that language. Then ’a’ and ’b’ are candidates for the tokens `A` and `B`, respectively. The abstract syntax for this language can be defined with a EBNF grammar, with `S` as start-production [Aho+06]:

$$S \to A \mid B \mid SS$$
$$A \to \mathsf{a}$$
$$B \to \mathsf{b}$$

When the lexer reads in the program “aba”, he outputs the token-stream `ABA` to the parser. According to this grammar, the parser builds the AST, which can be seen in Figure 2.4. At the bottom is the character sequence, on top of that – in squares – are the tokens as the leaves of the AST and on top of the tokens are the `S`-productions that span the rest of the tree.

When the parser has finished building the AST and no errors occurred, it passes the AST further to the *syntactical analyzer*. The analyzer is responsible for checking context conditions of the respective language. Therefore, he builds up all necessary infrastructures, like the symbol table, and reports all errors to the error handler. An example context condition for the ab-language could be, that at most five ’a’s are allowed. Extending the given grammar to catch that condition is possible, but increases

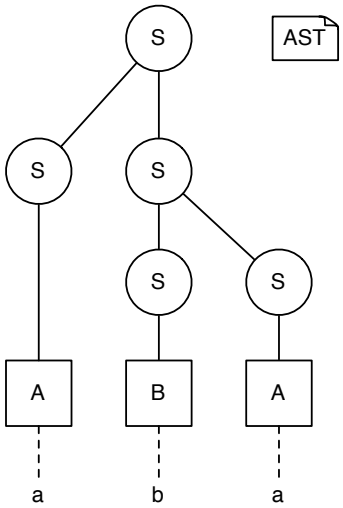


Figure 2.4: The abstract syntax tree of the program “aba” for the ab-language. The circles denote the S-productions, the squares denote the tokens as the leaves of the AST. At the bottom is the actual character sequence.

its complexity a lot. A context condition can check this on the AST very easily. Finally, if the AST complies with the language definition and fulfills all context conditions, the AST is passed to the *code generator*, that generates the target program out of the AST and symbol table. This generator step is often called the compiler *backend* and different code generators can be used interchangeably, which is important, if the compiler should generate code for different targets, e.g. machine code for different processors or neuron models for different simulators.

2.5 MontiCore

The development of compilers for a new programming language is a hard task and has lots of pitfalls. The MontiCore project [Fic+13] is a tool that supports the development of languages with Java in many ways. On the one hand, MontiCore is a generator for lexers and parsers for context-free programming languages and on the other hand, MontiCore is a framework that simplifies and speeds up the other steps of language processing described previously in Section 2.4. Hence, MontiCore fits very well into agile, model-driven software development, as DSLs created with MontiCore assist developers to define models more declarative and its template based code generation allows fast, iterative development and improves source code maintainability.

MontiCore uses one grammar format to formulate new language definitions. With this format the concrete and the abstract syntax of the new language is defined simultaneously. Out of a language defined in this grammar MontiCore generates the lexer and parser for this language, as well as class definitions for the AST and other auxiliary classes. An introduction to the grammar format and its usage is given in Section 2.5.1.

The MontiCore framework contains support for the creation and processing of symbol tables and for handling name resolution. It has support for checking of context conditions and has a sophisticated error-reporting engine. These concepts and how they

are organized in the MontiCore DSLTool are explained in Section 2.5.2. MontiCore is capable to analyze the hierarchical structure of the AST in two ways. The standard way is to use a variation of the visitor pattern [Gam+95]. The other way is to use attribute grammars [Ste12] that allow calculating attributes of nodes in the AST top down or bottom up and are described in Section 2.5.4. For code generation MontiCore works with the FreeMarker template engine [DRS13]. Its syntax and usage is illustrated in Section 2.5.5.

To allow for modularized development of languages, MontiCore enables developers to compose languages in different manners. A language can inherit syntax from another language or it can allow embedding parts of another languages syntax into itself. Since languages can be composed in such a way, MontiCore makes composition of context conditions and symbol tables possible, too. Section 2.5.3 introduces the concepts of language composition in MontiCore in more detail.

2.5.1 MontiCore's DSL Grammar

Languages developed with MontiCore use a grammar system based on regular expressions to define its concrete and abstract syntax; similar to ANTLR [Par07; PH13], upon which MontiCore is built. In this Section the basic syntax of MontiCore's grammar system will be introduced. An extensive presentation of MontiCore's grammar can be found in its documentation [Fic+13]. All MontiCore grammars start with a package definition, which is followed by import statements, to include names of other packages. After that the actual grammar definition starts with the keyword **grammar**, followed by the grammar name and a block that contains the options and productions of the grammar. Single-line comments start with `//` and continue until the end of the line. Multi-line comments span from an opening `/*` to a closing `*/`. Listing 2.1 shows an example grammar for a language that describe (physical) *units* and their range – this is a simplified version of the units grammar defined in Section 4.1. MontiCore differentiates between lexical productions that describe how tokens are created from of character sequences, and parser productions, that describe how sequences of token are identified as sentences of the language.

Lexical productions start with the keyword **token**, have a token-name and a regular expression assigned to them. The regular expressions of lexical productions only consist of other tokens or terminal symbols. Terminal symbols are either a character surrounded by single quotes, like `'a'`, or a sequence of characters surrounded by double quotes. Alternatives for symbols can be expressed by vertical bars or with ranges of characters, like `'a'..'z'`. The expected number of occurrences of a symbol can be expressed with either a question mark for zero or one occurrence, a plus sign for one or more occurrences or with a star sign for zero or more occurrences. Symbols can be grouped with parenthesis. If a token is preceded with a type, like *Number* in line 10, then the source of the token is not stored as a string, but the lexer tries to cast the string into that type.

Parser productions, or *nonterminal* productions, can start with the keywords **interface** or **abstract** or with no keyword. They also have a name and a regular expression assigned to them. The regular expressions of parser productions consist of tokens, terminals or nonterminals. Concatenation is expressed by putting symbols one after another, see

line 16 of the grammar in Listing 2.1 – there is no special character for concatenation. Expressing alternatives and number of occurrences is handled in the same way as for lexical productions. Additionally, alternatives for terminals can be expressed with square brackets. Symbols can be preceded with an identifying name, like from:Number. Terminals in parser productions are automatically identified as special keywords of the language and as such are not permitted as identifiers.

```
1  package unit;
2  // A sample grammar for a language describing physical units.
3  grammar Unit {
4    options {
5      nostring noident
6    }
7    // token for identifier
8    token Name = ( 'a'..'z' | 'A'..'Z' | '_' )+;
9    // token for integer numbers
10   token Number = ('0'..'9')+ : int;
11
12   Units = (Unit ";")* ; // allows zero or more units to be defined
13
14   // unit definition: a name, a domain and its range with inclusive
15   // or exclusive parenthesis
16   Unit = "unit" unitName:Name Domain LeftParenthesis Range RightParanthesis;
17
18   Domain = ([real:"Real"]|[integer:"Integer"]);
19
20   LeftParenthesis = ([inclusive:"["]|[exclusive:"("]);
21
22   RightParanthesis = ([inclusive:"]"]|[exclusive:")"]);
23
24   Range = (from:Number | [fromInf:"-inf"]) "..." (to:Number | [toInf:"inf"]);
25
26   // add methods or variables to the generated AST classes
27   ast Unit = method public String toString() {
28           return "unit " + getUnitName();
29       };
30 }
```

Listing 2.1: Example Grammar for Units in MontiCore.

When MontiCore processes such a grammar definition, it generates a lexer and a parser for the language and creates class definitions for the abstract syntax tree. This generation process gets customized in different ways: the **options** block of the grammar may include options like noident or nostring to remove predefined tokens IDENT and STRING, respectively, or options like parser lookahead=1 and lexer lookahead=4 influence the parser and lexer. Parser productions are translated to a class for the AST and its regular expression defines the inner structure of the generated class. Terminal symbols are ignored, since they do not contain additional information for the AST. Tokens and nonterminals become member variables of the generated class, where tokens translate to a String and nonterminals to the appropriate AST class. The name of an element in the production influences the name of the appropriate member variable of the AST class. Alternatives in square brackets become integer member variables, with constants defined in a separate ASTConstants<GrammarName> class, if there is more than one option in the square brackets. Otherwise, this member variable is a boolean which states, whether it is there or not, see line 18 in Listing 2.1. The AST class hierarchy for the simple units grammar from Listing 2.1 is shown in Figure 2.5 – for simplicity some gen-

erated classes and methods are omitted.

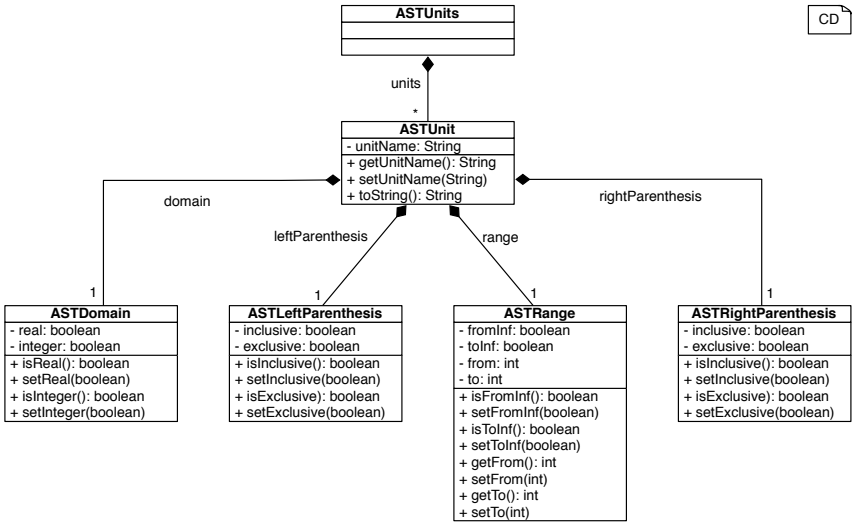


Figure 2.5: Generated class hierarchy from the simple unit grammar in Listing 2.1. A production becomes a class of the same name with "AST" prepended. Elements inside the production become member variables of the corresponding class.

The MontiCore grammar has several other elements that influence the parsing behavior or the generation process. The element **ast** <ProductionName> = ... allows to extend the generated AST class with additional member variables and methods. If a parser production is preceded with a slash, like /Unit = ..., then MontiCore only generates an abstract prototype class for that production which the developer has to extend. Syntactic predicates can be used to eliminate ambiguity of alternatives in productions and block operations can solve the "dangling-else-problem" [Aho+06] by setting the greediness of the parser. The "dangling-else-problem" can occur, when nested *if*-statements are followed by an *else*-statement, like in **if** (<TEST>) **if** (<TEST>) /*.. */**else** /*.. */. Then it is ambiguous to the parser, to which *if*-statement the *else*-statement belongs to.

2.5.2 MontiCore Framework

The MontiCore framework is built from modules, that assist developers to create the desired language-processing tool, with lexing and parsing of input programs, static analyzes to check correctness of the program and corresponding code generation. The main entry point to the framework is the `mc.DSLTool` class. Developers need to subclass this class for their tool. The DSLTool supports the implementation of a command-line interface to the compiler and has a basic file processing behavior implemented. Figure 2.6 presents the architecture for a compiler developed with MontiCore. All components need to be registered with the DSLTool. *Workflows* make up the building blocks of all tasks a compiler has to perform. The *root factories* create the *DSLRoot* objects that represent an input file or a model and are handles to their AST and auxiliary data. The *language-processing* contains the lexer, parser and AST classes that are responsible for processing input files and creating the AST, see Section 2.4. The *error delegators* contain several *error handlers*, that process upcoming *problem reports* and flag the respective root object, if an error occurred. The workflows, which are executed after

an error was found, can then decide, whether to continue or to finish the processing. When the DSL grammar is passed to MontiCore, it generates the language-processing infrastructure and the root factories. Some basic workflows, e.g. the parsing workflow, are also generated, which is indicated by the dashed arrow.

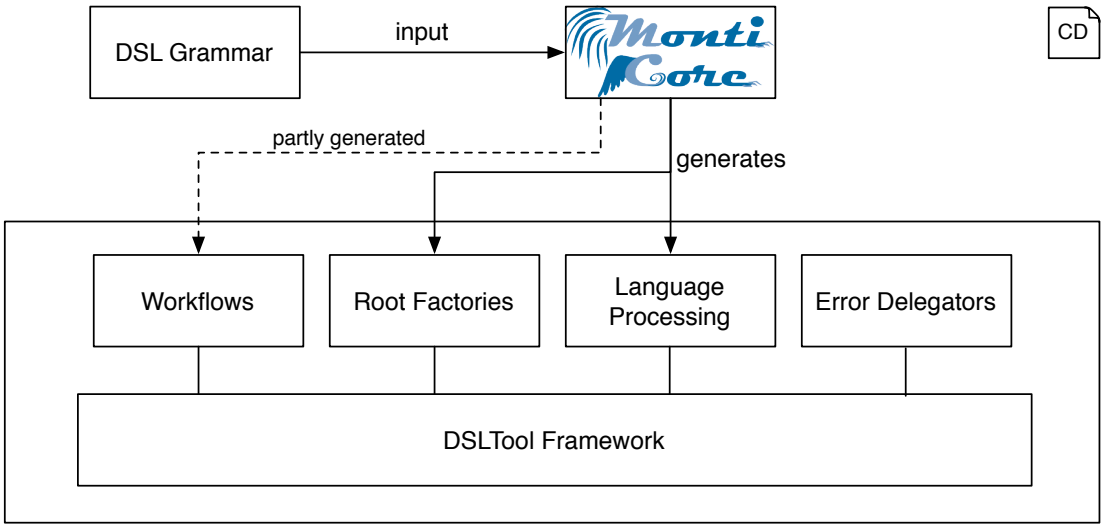


Figure 2.6: The basic architecture of a compiler developed with MontiCore. Redrawn from Krahn [Kra09]. MontiCore reads in a grammar definition and generates the core language processing infrastructure.

The *workflow infrastructure* is an important part of language-processing tools built with MontiCore. The parsing and the checking of context conditions is modeled as a workflow and the creation of symbol tables is modeled as a composition of workflows. Whenever the developers want to analyze or work with the AST, the MontiCore framework favors the use of workflows. Figure 2.7 outlines this infrastructure. The DSLTool as a subclass of the `mc.ADSLToolExecuter`, executes all `mc.ExecutionUnit` in the order they are registered to it. The `mc.DSLWorkflow` is a subclass of the ExecutionUnit, that represents one step in the processing of one file. Developers need to subclass the DSLWorkflow to implement their own tasks. Both, the ExecutionUnit and the DSLWorkflow, work on `mc.DSLRoot` objects which are created by a `mc.DSLRootFactory`. Every language generated with MontiCore has its own subclass of DSLRoot and DSLRootFactory and a DSLRoot object represents a file or model of that language. The DSLRoot object contains a handle to the AST for that model, has information about all errors in that model and stores its symbol table.

Developers can analyze and manipulate the AST in different ways within a workflow. The *visitor pattern* [Gam+95] is the preferred way of MontiCore: a supervising visitor with a set of client visitors traverses the AST in depth-first order [SW11]. When this visitor visits a node, it calls the `visit`-method of its client visitors with this node as argument. Then it traverses the children of that node and, when all children are visited, he calls the `endVisit`-method of its client visitors with this node as argument. The developer only needs to implement the `visit`- and `endVisit`-methods of these client visitors for nodes he is interested in. With the `ownVisit`-method he can influence the order of visiting child-nodes. Another way is to use MonitCores *functional API* and perform filter-, fold- and map-functions on the AST [Kra09]. Finally, MontiCore supports the

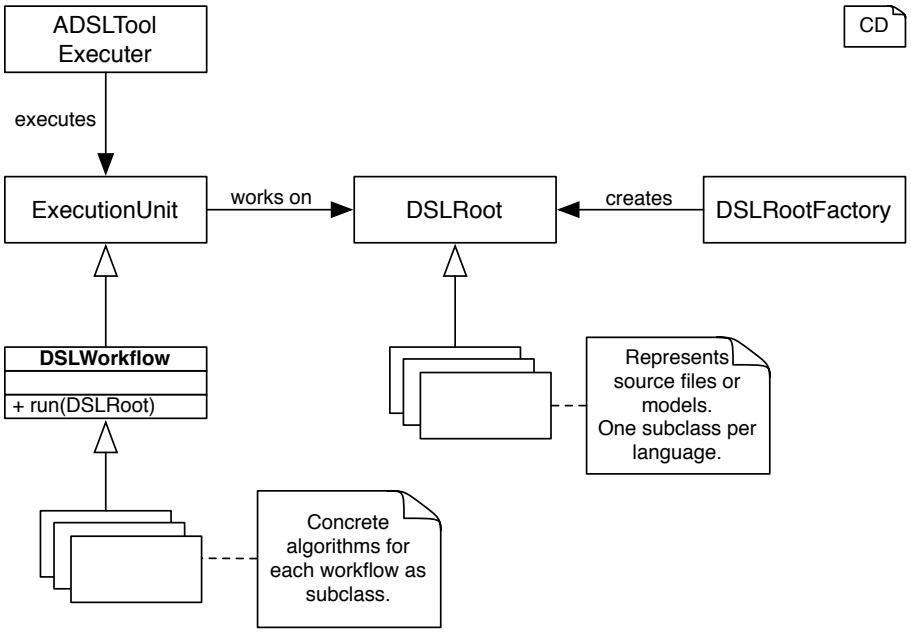


Figure 2.7: The workflow infrastructure of MontiCore. Adopted from Krahn [Kra09]. The `ADSLToolExecuter` executes all `DSLWorkflows`, which work on the `DSLRoot` of the current model.

concept of *attribute grammars*, which is explained in Section 2.5.4 in more detail.

MontiCore generates the parsing workflow and the root factories, but the MontiCore grammar does not state, which production should be used as the start-production or how the workflows and factories should be named. For this purpose, MontiCore uses a separate language DSL that state these properties. Listing 2.2 shows the language file for the simple units grammar from Listing 2.1. It defines the name for the root class, the name for the parsing workflow associated to that root class and it defines the root factory name associated to that root class. In line 10 the production `Units` from the grammar `Unit` is set as start-production. This start-production is also needed behind the name of the root class in angle brackets. Also, inside this language DSL the embedding of other languages is declared, see Section 2.5.3.

```
1  package unit;
2
3  dsltool UnitTool {
4      //Root Class
5      root UnitRoot<Package>;
6      // Parsing Workflow
7      parsingworkflow UnitParsingWorkflow for UnitRoot<Package> ;
8      // RootFactory
9      rootfactory UnitRootFactory for UnitRoot<Units> {
10         Unit.Units units <<start>> ;
11     }
12 }
```

Listing 2.2: Language DSL for simple units.

The MontiCore framework contains support for creating and working with *symbol tables*. The developer needs to define the namespace establishing AST nodes and thereby

creates a hierarchical structure for the namespaces, this allows for scoping of symbols. Each namespace has a link to its parent namespace and can have multiple child namespaces. A namespace has different types of symbol tables, that represent the symbols origin or whether it should be public to other namespaces. Symbols are stored as language specific symbol table entries that store, beside their name, all relevant information about that symbol, e.g. its type and modifiers. Figure 2.8 shows the relationship between namespaces, symbol tables and its entries. A dedicated workflow for building up and filling namespaces and symbol tables is predefined in MontiCore. A visitor is used to traverse the AST. Whenever the visitor inspects a node that is relevant for the symbol table, it is responsible for transforming that node into a symbol table entry and for storing that entry in its corresponding symbol table.

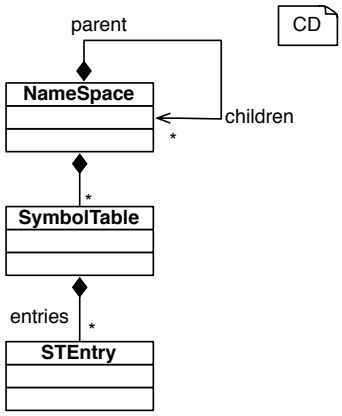


Figure 2.8: The relationship between NameSpace, SymbolTable, Entries. Adopted from Völkel [Völ11]. `NameSpaces` are hierarchically structured and contain several `SymbolTables`, which in turn contain the symbols (`STEntry`) of the current namespace.

After the symbol table workflow has finished without errors, the context condition check workflow can be executed. Therefore, the developer registers, on the one hand, a set of `interfaces2.coco.ContextConditions`, which contains methods to report errors and to resolve symbols, and, on the other hand, a `CheckWorkflowClient` visitor, which gets that set of context conditions and executes these context conditions on all relevant nodes of the AST.

Language composition and code generation are also part of the MontiCore framework, and are introduced in Section 2.5.3 and Section 2.5.5, respectively.

2.5.3 Language Composition

Building language-processing tools from ground up is a complex task. Therefore, MontiCore has different methods to reuse languages or part of languages, that are build with MontiCore, in new languages. This also reduces the complexity of building a language, since this task can be split into smaller chunks. This section describes MontiCore's three methods of compose languages: *inheritance*, *embedding* and *aggregation* [Völ11].

Language inheritance means that one language inherits all productions from another language, e.g. a language's `Literals` can contain several useful literals and by `extending` that language all other languages can reuse those literals. Listing 2.3 shows, how

the new language `UnitsWithDescription` extends the language `unit.Unit` in line 3 and thereby inherits all its literals and productions. It is also possible to extend these productions, like in line 4 of Listing 2.3. *Interface* and *abstract* productions allow building an interface for a language, to make clear at what points extensions are intended.

```
1 package unit2;
2
3 grammar UnitsWithDescription extends unit.Unit {
4   UnitWithDescription extends Unit = "description" "unit" unitName:Name desc:String;
5 }
```

Listing 2.3: Example for language inheritance.

MontiCore accepts *external* productions that are filled by *embedding* productions of other languages. To declare such blank productions, the developer adds a statement in the grammar definition, that starts with `external` and is followed by the production name, like **external** Calculation;. This way, every time a calculation is needed in the grammar, this production can be used. In the language DSL file of a language this production is then linked to a real production of another language, see Listing 2.4, line 8.

```
1  package mc;
2
3  dsltool SampleTool {
4      // ... some parts are omitted
5      rootfactory SampleRootFactory for SampleRoot<Package> {
6          Sample.Package package <<start>> ;
7          // link the production Expression to the external production Calculation
8          mc.javadsl.JavaDSL.Expression expression in mc.Calculation;
9      }
10 }
```

Listing 2.4: Link a foreign production to an external production.

It starts with the full path to the real production (`mc.javadsl.JavaDSL.Expression`), then an identifier (`expression`), followed by the keyword **in** and the path to the blank production (`mc.Calculation`). Another language, that should use C++ expressions with anything else staying the same, could reuse the grammar definition and only need to supply a modified language DSL file.

In some application scenarios two or more languages are used to support each other, e.g. one language declares a class hierarchy and the other languages uses the declared classes. This kind of information exchange is possible with *language aggregation* that processes different languages within one `DSLTool`. The `DSLTool` has to register their root factories and can specify different workflows for those root objects, e.g. parsing workflows. This allows to create one tool, that aggregates all DSLs, that a client needs, and has the possibility, to build symbol tables and context conditions across the models of different DSLs, e.g. a sequence diagram DSL could check with a class diagram DSL whether a certain object can call a certain method.

When languages are composed in such ways, their symbol tables and context conditions potentially need to be merged. To make symbols from one language available to another language adapters are used [Gam+95; Völ11]. Adapters can be introduced either during the set-up of the symbol table, by registering a `QualifiedEntryHandler`, that creates an adapter for every entry, or during resolving of symbols, by registering a dedicated `IResolverClient`, that forwards the resolving to clients of the other language and

builds the adapter around the result. Merging the context conditions of two languages requires registering both sets of `ContextCondition` and both `CheckWorkflowClient` visitors. Additional context conditions to check the interplay of both languages are also possible.

2.5.4 Attribute Grammar

Extending context-free grammars with attributes and computation rules is an alternative method to analyze abstract syntax trees, especially its hierarchical structures. These context-free grammars are called *attribute grammar*. Only nonterminals of an attribute grammar are extended with attributes and have computation rules to compute these attributes. There are two kinds of attributes: *Synthesized* attributes are calculated bottom-up, i.e. from the leaves of the AST to the root. *Inherited* attributes are calculated top-down, i.e. from the root to the leaves. The following grammar is a modified version of the ab-language from Section 2.4.

Productions	Synthesized Rules	Inherited Rules
$S' \to S$	$\mid len(S') = len(S)$	$\mid dep(S) = 1$
$S \to S_1 S_2$	$\mid len(S) = len(S_1) + len(S_2)$	$\mid dep(S_1) = dep(S_2) = dep(S) + 1$
$S \to A$	$\mid len(S) = len(A)$	$\mid dep(A) = d(S) + 1$
$S \to B$	$\mid len(S) = len(B)$	$\mid dep(B) = d(S) + 1$
$A \to$ a	$\mid len(A) = 1$	
$B \to$ b	$\mid len(B) = 1$	

Every nonterminal has the synthesized attribute len of type integer and the inherited attribute dep of type integer. The len-attribute should store the length of the ab-word in that node and the dep-attribute should store the depth of the appropriate node. To the right of the productions are the computation rules. The synthesized rules always assign a value to the attribute of the nonterminal on the left-hand-side of the production and the inherited rules assign a value to the attribute of every nonterminal on the right-hand-side of the production. In the computation rules all synthesized and inherited attributes of the current production can be used, but it is important, that no circular dependencies between rules exist. These circular dependencies can not only occur locally to the production, but also across multiple nodes of the AST. If there are circular dependencies between rules, the attributes can not be computed [Ste12]. Algorithms to check that no circular dependencies can occur within a given attribute grammar exist, but need exponential time, if no further restrictions are applied [Ste12]. Figure 2.9 shows a computation of the attributes on the AST of Figure 2.4.

```
1 ...
2 concept attributes {
3   syn len: /java.lang.Integer; // declare the synthesized attribute len of type Integer
4   inh dep: /java.lang.Integer; // declare the inherited attribute dep of type Integer
5 }
6 ...
```

Listing 2.5: Attribute definition inside grammar.

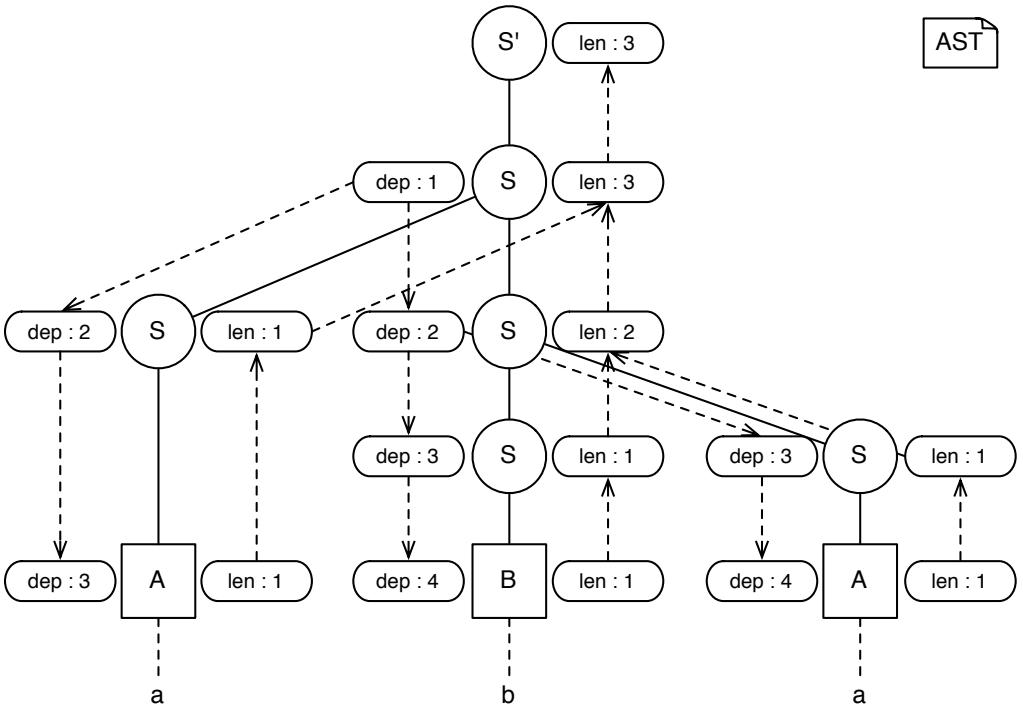


Figure 2.9: Computing inherited and synthesized attributes on the word “aba” with the AST from Figure 2.4. The inherited attribute dep is on the left side of the node and the synthesized attribute len is on the right side of the node. The dashed arrows show the computation flow for the attribute.

With MontiCore the attributes are defined inside the grammar definition of the language, but their rules are defined with a special calculator class. Listing 2.5 shows, how the synthesized len- and inherited dep-attributes are defined inside the grammar definition of the ab-language – a slash in front of the `java.lang.Integer` denotes, that a Java-type is meant and not a production from the grammar. Listing 2.6 shows, how the attributes are linked to their calculator class. A synthesized attribute needs a calculator class, that extends `SynthesizedAttributeCalculation` and as `calc`-methods for every AST node, that is important for that attribute. An inherited attribute needs a calculation class, that extends `InheritedAttributeCalculation` and as `calc`-methods for every AST node, that is important for that attribute.

```
1  ...
2  concept Attributes {
3    len {       // let the class ablang.LenCalculator calculate the attribute len
4      ablang.ABLang.len = /ablang.LenCalculator
5    }
6    dep {       // let the class ablang.DepCalculator calculate the attribute dep
7      ablang.ABLang.dep = /ablang.DepCalculator
8    }
9  }
10 ...
```

Listing 2.6: Computation rules inside language DSL.

MontiCore generates for every language a subclass of `AttributeStorageConnector`, which handles the evaluation of rules and the caching of already calculated attribute

values, i.e. the attributes are not stored on the nodes of an AST, but within this `AttributeStorageConnector`. Circular dependencies are checked during runtime and entering a `calc`-method twice during the evaluation of one attribute leads to an exception. Assume the ab-language is defined with MontiCore, has the name `ABLang` and the value of the attribute len is needed from the node n of type `ASTS`, then a new object of the generated connector `ABLangConcreteStorageConnector` has to be created and the method `getLen(node)` is called on that object. [Ste12]

2.5.5 Code Generation with FreeMarker

When the compiler has successfully finished parsing and analyzing the input program, its AST can be forwarded to code generation. The `mc.GenerationTool`, a subclass of `mc.DSLTool`, implements a general infrastructure for code generation with *FreeMarker templates* [DRS13]. To use that infrastructure, the developer has to register template- and AST node-pairs and the `CodegenVisitor` from the `CodegenWorkflow` will execute a template whenever it visits its corresponding node.

FreeMarker templates are text files that contain arbitrary text and *FreeMarker* syntax side by side. Whenever a template is executed, this syntax is interpreted and replaced with resulting text. FreeMarker has two kind of elements in its syntax: *interpolations* and *directives*. *Interpolations* execute all kind of expressions and write out the result of that expression. The syntax for interpolations is `${<expression>}`, e.g. `${3+6}`. *Directives* are responsible for assigning variables, for branching and for looping. All directives basically have the following structure: they have a starting tag `<#directivename parameters>`, with different parameters according to the directivename, and most directives need a closing tag `</#directivename>`. See Listing 2.7 for a FreeMarker example, with Listing 2.8 as the result of executing that template.

```
1  <#assign x=0 names=["johnson", "smith", "alice", "bob"]>
2  <#if x == 0>
3  x is zero
4  <#else>
5  x is not zero
6  </#if>
7
8  Let's loop through some names:
9  <#list names as n>
10 - Hello, ${n}!
11 </#list>
```

Listing 2.7: Examples for the FreeMarker syntax.

```
1  x is zero
2
3  Let's loop through some names:
4  - Hello, johnson!
5  - Hello, smith!
6  - Hello, alice!
7  - Hello, bob!
```

Listing 2.8: Result from executing Listing 2.7.

MontiCore extends FreeMarker's template processing engine to allow more complex calculations on the AST outside of the templates. Therefore, when using FreeMarker

templates with MontiCore, the templates contain two predefined variables: the current AST node is stored in the variable `ast` and allows the access to all information and children of that node. A `TemplateOperator` is stored in the variable `op`. The TemplateOperator has auxiliary methods, like `callTemplate` or `includeTemplate`, that embed the resulting text or generate a separate file. Other methods, like `callCalculator` or `applyCalculator`, execute subclasses of `TemplateCalculators`, that implement calculations in Java, which are too extensive for the FreeMarker syntax. To support these two tasks, it has an own variable infrastructure, to pass down variables from calculators to the calling template or from a template to the templates it calls. With `op.setValue("varName", <value>)` a variable `varName` with some value can be declared. Those variables can be accessed either the same way as variables declared with the `assign`-directive or with the `getValue`-method of the TemplateOperator.

Chapter 3

Requirements Analysis

In this chapter the requirements for the domain specific language NESTML and the tool, that processes NESTML models, are listed, including a short description of each requirement. The order does not indicate an ordering of the requirements. Section 3.1 lists the non-functional or quality requirements, which define constraints for NESTMLs design and implementation. Section 3.2 lists the functional requirements defining what NESTML is supposed to be capable of.

3.1 Non-Functional Requirements

This section lists the non-functional requirements of NESTML and its processing tool. With NESTML it should be possible to express the most important concepts of neuron models and it should be easy to read and comprehend neurons written in NESTML. Ease of use and completeness with respect to the concepts are a prerequisite for the adoption of NESTML by neuroscientists.

NF01 NESTML should be expressive and easy to comprehend and learn.

The workload and complexity to model neuron models with NESTML should be notably lower, than to implement the models for the simulator directly. In addition, NESTML's processing tool should support the development, e.g. with constructive and comprehensive error messages.

NF02 NESTML should support developers in creating neuron models.

When NESTML's feature set will be extended in the future, a clean and comprehensible API will support developers to create these features for NESTML. Also, the code base of simulators is likely to change over time, hence the code generator of NESTML needs to cope with those changes.

NF03 NESTML should be easy to maintain.

In the scope of this thesis, only the most important concepts of neuron models will be implemented. More concepts and features are planned for the future. Therefore,

extensibility is an important design goal.

NF04 NESTML should designed to be easy extendable.

NF04.1 NESTML should be extendable to model more detailed neuron models, e.g. multi-compartment neurons.

Several methods to advance the neuron dynamics in time are known and used in computational neuroscience. The dynamics could be expressed by advancing the neuron state by one simulation time step or by the minimum delay over all synapses [Han+10]. The dynamics can even be modeled as an event based system, where incoming spikes or electric currents are processed individually. In the scope of this thesis only the first of these dynamics will be considered, but extendibility for the others would be desirable.

NF04.2 NESTML should be extendable to model different types of neuron dynamics.

Many neuron models are described in terms of ODEs and solving ODEs can be very difficult. The ability to model neurons with ODEs and defer their solving to an ODE solver of the target simulator makes formulating neuron models easier.

NF04.3 NESTML should be extendable to model neurons in terms of ordinary differential equations (ODE).

A neuron model should be specified only once in NESTML. From these models the code for a neuronal network simulator should be generated. NESTML's processing tool should support code generation for different simulators eventually, so that the same neuron model can be simulated with those simulators. This would allow to reproduce and verify scientific findings across simulators.

NF05 A NESTML model should be used to generate simulator specific code, that represents this neuron model in the simulator.

NESTML should only generate code for programmatic correct models and issues that could lead to compilation errors in the generated code should already be caught by context conditions of NESTML.

NF06 Generated code should be programmatic correct.

Different simulators have different approaches to efficient code. Some may only need fast code, some may have memory limitations and others may have constraints for the code size. In either case, NESTML should allow to follow the simulator dependent guidelines to generate efficient code for specific simulators.

NF07 Generated code should be efficient.

3.2 Functional Requirements

This section contains the functional requirements of NESTML and its processing tool. They state, what NESTML shall allow to model (F01 – F14) and what code shall be generated for the NEST simulator (F15 – F17).

NESTML is a domain specific language (DSL) to model neurons. We decided to start with modeling point neurons, since they represent the structurally simplest neuron models. Extendibility for more complex models is stated in NF04.1.

F01 NESTML shall model point neurons.

During simulations of neuronal networks neurons get inputs and generate output. Based on the input the internal state of a neuron changes over time, e.g. the membrane potential increases or decreases upon incoming spikes from other neurons. The state of a neuron needs to be modeled with NESTML.

F02 NESTML shall model the state of a neuron.

F02.1 The state of neurons shall be modeled as variables.

F03 NESTML shall model the state changes (dynamics) of a neuron.

A neuron has several attributes that do not change over time, but can vary from neuron to neuron. Important for simulations could be individual size, resting potential or spiking threshold of a neuron. We decided, that individual neuron should be parameterizable, to model this behavior.

F04 NESTML shall allow neurons to be parameterizable.

F04.1 The parameters of a neuron shall be modeled as variables.

F04.2 The parameters of a neuron shall have initial (default) values.

F04.3 The parameters of a neuron shall not be changeable during simulation.

Besides state and parameters, neuron models often need auxiliary and intermediate values to advance the state. These could be values, that do not represent biophysical properties of a neuron, but are needed in order to calculate the neuron dynamics.

F05 NESTML shall model internal variables for neurons.

Many attributes of a neuron have physical units, e.g. the membrane potential is measured in millivolt and the refractory time is measured in milliseconds. NESTML should reflect this, by modeling physical units and allowing variables to be defined with these units as data type.

F06 NESTML shall model physical units.

F07 Variables in NESTML shall be declared with an appropriate type.

Since variables in NESTML already have a fixed type, a strict type system helps to find errors during static analysis and thereby assists developers in creating neuron models with NESTML.

F08 NESTML shall have a strict type system.

In a neuronal network neurons can receive different types of input. This could be spikes from other neurons or electric currents from external devices. These inputs influence the state of a neuron and are thus important, to express the neuron dynamics.

F09 NESTML shall model neurons with different types of inputs.

Neurons exchange information by emitting spikes via their axon to the dendrites of other neurons. Hence, the spike output needs to be modeled by NESTML.

F10 NESTML shall model neurons to allow output.

To model the neuron dynamics, function bodies (F12) or initial values of variables in a detailed and structured way, NESTML should have a simple procedural language. This language should allow expressing program flow and to model different dynamics and functions. This includes (possibly infinite) looping, branching, mathematical expressions and local variables. Furthermore, the language should have a clean and consistent syntax that supports neuroscientists to be productive and only have a minimum set of statement types consistent with NESTML.

F11 NESTML shall have a simple procedural language to model dynamics and functions.

F11.1 The simple procedural language shall allow to model all possible program flows.

F11.2 The simple procedural language shall have a minimum set of statement types.

F11.3 The simple procedural language shall have a readable and concise syntax.

F11.4 The simple procedural language shall be consistent with NESTML.

When certain, common functionality is grouped and can be reused by different neuron models, the complexity to model a neuron can be reduced and neuron models potentially have less errors. Therefore, NESTML should allow formulating functions and modeling components, that contain domain-specific and often used functionality.

F12 NESTML shall allow to modularize functionality into functions.

F13 NESTML shall allow to modularize often used functionality into components.

Components and units can only be reused, if it is possible to refer to them in an unambiguous way. Therefore, neuron models, units and components should be organized in a hierarchical way.

F14 NESTML shall organize neuron models, units and components in a hierarchical way.

The NESTML processing tool should generate code for different simulators from a neuron model specification (NF05). The NEST simulator is chosen as the first supported simulator, hence the following requirements concern the code generator for the NEST simulator. On the one hand, a NESTML neuron model need be transformed into a NEST neuron model. On the other hand, all auxiliary components and units that are used by the NESTML neuron model need to be transformed into elements, that the NEST neuron model can use.

F15 The NESTML processing tool shall generate neuron model code for the NEST simulator.

F16 The NESTML processing tool shall generate all auxiliary components and units that are used by a NESTML neuron model.

It shall be easy to integrate the generated neuron models into to NEST simulator, so the NESTML processing tool has to generate all necessary code for NEST's module infrastructure. This way the generated code can directly be compiled for NEST and included as a dynamic library.

F17 The NESTML processing tool shall generate additional code for NEST's module infrastructure.

Chapter 4

Design

This chapter describes the design of NESTML and its sub-languages. It describes the purpose of each language, how they are composed and the meaning of the individual elements of the languages. Section 4.1 covers the *UnitDSL*, which is used to model physical units. In Section 4.2 a *simple procedural language* (SPL) is described. *NESTML* and the composition of all languages are outlined in Section 4.3. Finally, Section 4.4 contains a detailed description of context conditions for the languages.

All languages inherit from a modified version of `mc.literals.Literals`. Listing B.1 shows the important parts of the `nestml.literals.Literals` grammar. This language includes several literals for numbers, strings, booleans and identifiers. This allows a notation of numbers and strings similar to Java, C/C++ or Python. Identifiers for variables, functions and types can contain any alphabetic letter, numbers, underscores or dollar signs ($). The identifier are case-sensitive. Many languages use semicolons as statement delimiter, but to reduce typing and provide a clean and concise syntax (F11.3), NESTML should allow to end a statement with a line break. Since MontiCore ignores all whitespaces by default, including line breaks, this behavior needs to be changed. In addition, single-line comments represent a line break and thereby a statement delimiter, but MontiCore skips those comments by default, too. By inheriting from `nestml.literals.Literals` the tokens `EOL` and `SL_COMMENT` can be used as a delimiter for statements.

The concrete syntax of NESTML, the UnitDSL and the SPL should be consistent across the languages (NF01,F11.4), hence some general design choices are needed: Since NESTML allows line breaks as statement delimiters, line breaks are statement delimiters for the UnitDSL and SPL, too. Requirement NF01 states, that the language should be easy to learn. Since the Python language is widely used in the neuroscientific community, we decided to design the languages to use a syntax similar to Python's. As Python's indentation syntax is inherently context-sensitive and MonitCore can only process context-free languages, we choose to start continuous blocks of statements with a colon, like in Python, and mark the end of blocks with the keyword **end**. Section 4.2 explains the context-sensitivity of Python in more detail.

4.1 The UnitDSL

Requirement F06 states, that NESTML should model physical units, and that the attributes of a neuron should be strictly typed (requirement F07). The size of a neuron is better described with the unit μm than with a generic floating point type. The spiking behavior involves the membrane potential in mV and the electric currents in pA that are gathered from the dendrites and the soma. Different time constants in ms are needed for the refractory behavior and the neuron dynamics.

One way to introduce units to NESTML would be, to implement a limited set of units in the core language. This would prohibit the use of other units since it would require a change to the language itself. A different approach is, to model physical units with a dedicated domain specific language and use those units to describe neurons. Such unit definitions can be grouped into a library and reused by future neuron models. Therefore, we decided to create the *UnitDSL* as part of NESTML. This section describes its syntax and gives a general definition of its semantics. The integration of the UnitDSL is described in Section 4.3.

Basically, units consist of a *name* that identifies them, and a *domain* in which elements of the unit reside. The domains in the UnitDSL are either the real numbers $\mathbb{R}$, denoted with the keyword **Real**, or the integer numbers $\mathbb{Z}$, denoted with the keyword **Integer**. The UnitDSL extends this definition, by adding a *range* for a unit. This can be useful: not only for physical units, but also allows to model primitive data types, like the unsigned integers, or any mathematical set of numbers, like the natural numbers $\mathbb{N}$. Variables, which use such a unit type, can then be checked for correct values. In principle, expressions with variables of physical units can be checked for correctness with respect to the resulting unit, but this is beyond the scope of this thesis.

The *concrete syntax* of the UnitDSL starts by declaring a package, using the command `package <name>`, followed by a colon. Then the unit definitions are listed. A semicolon or a line break can delimit them. If semicolons delimit multiple unit definitions, the definitions can be written in one line one after another. Finally, the keyword `end` completes the package definition. Single-line comments start with a `//` and continue to the end of the line and multi-line comments start with a `/*` and continue to the matching `*/`. See Listing 4.1 for a full UnitDSL example and Listing B.3 for the full UnitDSL grammar.

```
1  // This is an example for the UnitDSL.
2  package example.units:  // the units are declared inside package example.units
3      // One Volt V consits of 1000 mV
4      unit mV Real (-inf ... inf);  // declare the unit mV of domain Real with the
5                                    // possible values between -infinty and infinity
6      unit ms Integer (-inf ... inf)
7      unit natural Integer [0 ... inf) // declare the natural numbers of domain Integer
8                                       // with possible values zero to infinity (excluding)
9      /*
10     declare a special unit of domain Real with values in
11     { x ∈ ℝ | -0.01 < x ≤ 1.5 * 10^13 }
12     */
13     unit special Real (-0.01 ... 1.5e13]
14 end
```

Listing 4.1: Example units for the UnitDSL.

Listing 4.2 shows the MontiCore production, with which a single unit can be described. A unit definition starts with the keyword **unit** and is followed by its base name – its simple name. Next, the domain of the unit is specified as either **Integer** or **Real**. The range of the unit is described in mathematical notation: parenthesis denote, that the left or right value is not included in the range. Square brackets denote, that the left or right value is included in the range. The left and right values denote the lower and upper bound of the range. The values can either be a numeric literal or infinity, denoted by −**inf** or **inf** for plus or minus infinity. The three dots indicate the numbers between the borders in the given domain. Hence, a unit can directly be translated into a mathematical set, e.g. the unit `special` from Listing 4.1 corresponds to the set $\{ x \in \mathbb{R} \mid -0.01 < x \leq 1.5 * 10^{13} \}$. The fully-qualified name of a unit is the simple name prepended with its package name, e.g. `example.units.mV`.

```
1  Unit = "unit" unitName:Name
2         ( [real:"Real"] | [integer:"Integer"] )            // domain
3         ( [leftInclusive:"["] | [leftExclusive:"("] )      // left delimiter
4             (from:SignedNumericLiteral | [fromInf:"-inf"]) // lower bound
5             "..."
6             (to:SignedNumericLiteral | [toInf:"inf"])      // upper bound
7         ( [rightInclusive:"]"] | [rightExclusive:")"] );   // right delimiter
```

Listing 4.2: The MontiCore production to describe a single unit with the UnitDSL.

Listing 4.2 specifies the *abstract syntax* of a single unit. As described in Section 2.5.1, productions in the grammar result in generation of corresponding abstract syntax tree (AST) classes and to keep the API flat, we decided to not divide the production like in Listing 2.1. Hence, the class `ASTUnit` corresponds to the `Unit` production and has the following member variables: The string `unitName` contains the base name of a unit. The boolean variables `real` and `integer` state, whether the unit has an integer or a real domain. The boolean variables `leftInclusive`, `leftExclusive`, `rightInclusive`, `rightExclusive` state, whether the range is delimited by inclusive or exclusive braces. The boolean variables `fromInf` and `toInf` state, if the lower or upper bound is infinity. Otherwise the variables `from` and `to` contain the lower or upper bound of the range.

```
1  UnitLine = Unit (options {greedy=true;}:";" Unit)* (";")?;
```

Listing 4.3: The MontiCore production to describe a single line of unit definitions.

As stated above, multiple unit definitions can be written in one line, if semicolons delimit them. Listing 4.3 shows the responsible production `UnitLine`. The corresponding AST class `ASTUnitLine` solely contains the member variable `unit`, which basically is a list of `ASTUnit` objects.

end	inf	Integer	package	Real	unit

Table 4.1: The reserved keywords of the UnitDSL.

4.2 A Simple Procedural Language

NESTML needs a way to define the dynamics of a neuron. Requirement F11 states, that a procedural language would be an appropriate approach, enabling a fine-grain control

over the neuron dynamics. MontiCore contains a complete DSL for Java, but requirement NF01 points out, that the language should be easy to learn for neurosientists. Since the Python language is widely used in this community, it is likely, that neuroscientist will adopt a language similar to Python faster than a language similar to Java. Hence, we decided to create a *simple procedural language* (SPL) with a syntax similar to Python's.

Lexers and parsers created with MontiCore can only process context-free languages (see Section 2.5), and the Python syntax is inherently context-sensitive: on the one hand, a continuous block of code is identified by the same level of indentation. This requires the lexer to know which is the current level of indentation and how deep previous indentations are. Listing 4.4 shows an example, that highlights this behavior: the `print i` in line 4 belongs to the body of the inner `for`-loop and the final `print` belongs to the body of the outer loop, since its indentation lines up with the start of the block of the outer loop in line 2. On the other hand, Python ignores indentation inside round, square and curly braces [RD11]. To make the syntax context-free and keep similarity with Python, a continuous block of code starts with a colon, but ends with the keyword `end`.

```
1 for i in x:
2     for j in y:   # start of outer loop code block
3         print j     # belongs to inner loop
4         print i     # belongs to inner loop
5     print         # belongs to outer loop
```

Listing 4.4: Different level of indentation for Python.

The concrete syntax of SPL essentially is a sequence of statements, which either can be simple statements, which span only one line, or statements that influence the control flow, possibly spanning multiple lines. Multiple simple statements can be written in one line, if a semicolon separates them. A line break or a single-line comment can be used as a delimiter of simple statements, too. Simple statements include *variable declaration*, *variable assignment*, *function calls* and *return statements*. The control flow can be influenced with `if`-branching and `for`- and `while`-loops. The syntax and semantics of these statements will be described in the rest of this section and Listing B.4 contains the complete grammar definition for SPL.

In several statements of the SPL mathematical and logical expressions can be used. Before discussing those statements, these expressions will be explained. Most of Pythons operators are also available in SPL. The operator $//$ is omitted due to simplicity. All operators that work with lists, tuples or dictionaries are omitted, since no such data types are present in SPL.

The binary operators $+$, $-$, $*$ and $/$ represent ordinary addition, subtraction, multiplication and division on numeric types The division operator performs integer division only if both operands are integer types. The expression $a**b$ rises a to the power of b: a^b. Strings are concatenated with other strings, numbers or boolean values with $+$. The unary and prefix operators $-$ and $+$ can be used on any number value, variable of numeric type or bracket term. The $-$ sign negates the associated expression, but the $+$ sign has no impact on the expression. Successive unary $+$ and $-$ signes are not allowed.

The following operators only work on integer expressions and are added for complete-

ness: the binary operators $|$, $\&$ and ^ represent the bitwise OR, XOR and AND, respectively. The bitwise left and right shift is done with the operators $<<$ and $>>$. The modulo operator % computes the remainder of an integer division. The unary, prefix operator ~ inverts every bit of the associated integer expression.

Numeric expressions can be compared with the typical binary operators $<$, $<=$, $>$, $>=$, $==$, $!=$ and $<>$ and result in a boolean value. In contrast to Python and for simplicity multiple comparisons can not be lined one after another. One of the boolean operators **and** or **or** is needed between two boolean expressions. The unary, prefix operator **not** can be used to negate a boolean expression. The operator precedence is given in Table 4.2. It is important to note, that expressions can not span multiple lines, since the line break is a delimiter for statements. Every expression can be surrounded by parenthesis to specify a non-default evaluation order.

Operators	Description
or	Boolean OR
and	Boolean AND
not	Boolean NOT (unary, prefix)
<, <=, >, >=, ==, !=, <>	Comparisons
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, −	Addition and subtraction
∗, /, %	Multiplication, division, remainder
+, −, ~	Positive, negative, bitwise NOT (unary, prefix)
**	Exponentiation

Table 4.2: The operator precedence in SPL from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence and are evaluated from left to right, except for exponentiation, which groups from right to left.

```
1 AND_Expr = SHIFT_Expr ("&" SHIFT_Expr)*; // shifts are more binding than bitwise AND,
2                                          // hence AND_Expr has a list of SHIFT_Expr
3 SHIFT_Expr = ARITH_Expr (SHIFT_ExprEnd)*;
4 SHIFT_ExprEnd = sign:["<<"|">>"] ARITH_Expr; // differentiate the sign of every
5                                              // subsequent operand
```

Listing 4.5: Example operator productions of SPL.

The abstract syntax for mathematical and logical expressions is very similar for all operators. The operator precedence is reflected in the abstract syntax, which on the one side makes the AST rather deep and on the other side simplifies the evaluation of certain context conditions, e.g. type evaluations of expressions. Listing 4.5 shows the productions for the shift operators and for the boolean operator AND (&) as examples. If there is only one operator in a row of Table 4.2, like for example the bitwise AND, the corresponding production does not need to differentiate operators. Hence, the corresponding AST class only has a list containing the next stronger binding precedence production, e.g. the AST class `ASTAND_Expr` for the production `AND_Expr` (line 1 in Listing 4.5) has a list containing `ASTShift_Expr` objects, which correspond to the `SHIFT_Expr`

production. If there are two or more operators in a row of Table 4.2, like the shift operators, the production of that precedence has to differentiate between the operators. The AST class for the production SHIFT_Expr (line 3) (ASTSHIFT_Expr) has a member variable containing the AST class of the next stronger binding precedence production ARITH_Expr (ASTARITH_Expr) – representing the first element of a shift expression –, and a list of ASTSHIFT_ExprEnd objects, which correspond to the SHIFT_ExprEnd production (line 4) and differentiates the corresponding left and right shift operator.

The concrete syntax of SPL contains two alternatives to *declare variables*: with and without an initial assignment of an expression. According to requirement F07, the declaration of variables involves a type. In contrast to many popular programming languages, we decided that the type is written after the variable name. This resembles the way, how units are written in physics: first the value or variable and then the unit. SPL itself does not allow to create new types, so some are predefined: the type `real` represents floating point numbers and `integer` represents integer numbers. The type `boolean` represent the boolean values `true` and `false`. Finally, the type `string` represents string values. It is possible, to declare more than one variable of the same type in one variable declaration by separating the variable names with comma. If a declaration with multiple variables has an initial assignment, all variables are assigned the result of the expression. If the declaration does not have an assignment, the variable has a default value depending on its type: zero for numeric types, the empty string for `string` and `true` for boolean. This eliminates the need for a generic `null` value for uninitialized variables and keeps unassigned variables in a deterministic state.

```
1  // a declaration can declare one or more variables, that are represented with
2  // the list vars, the variables have a type and an optional expression assigned
3  Declaration = vars:Name ("," vars:Name)* type:DottedName ( "=" Expr )?;
4
5  // an assignment assignes a expression to a variable
6  // which is represented by the dotted name
7  Assignment = DottedName "=" Expr;
```

Listing 4.6: SPL production for variable declaration and assignment.

After the variable is declared it can be (re)assigned or used in expressions. Listing 4.6 shows the production for variable declarations and variable assignments in SPL and Listing 4.7 shows examples for variable declarations and assignments. The type of the expression of an assignment must match the type of the assignee variable.

```
1  foo real = -4.34e12           // variable foo of type real with initial value -4.34e12
2  x, y, z integer               // three integer variables with initial value 0
3  bar string = "Hello, world!"  // variable bar of type string
4  b boolean = true              // variable b of type boolean
5
6  // assign x, y and z
7  x = 1 << 5; y = x ^ 0xFF      // x equals 0x20 or 32, y equals 0xDF or 223
8  z = y + 1                     // z equals 0xE0 or 224
9
10 // function call demonstrating string concatenation and mathematical expressions
11 print("result: " + (2 + 3 - 4 * 5 / 6. ** -random()) * 0x23)
```

Listing 4.7: Example of variable declarations and variable assignments.

In SPL functions can be called in expressions and as a simple statements. SPL itself does not allow creating new functions, but some functions are predefined in a standard

library. In the last line of Listing 4.7 an example of function calls as a simple statement is given: first the name of the function is stated, then, in parenthesis, the arguments are given to the function. If a function needs more than one argument, they are separated by comma. The types of the arguments have to match the types of the function parameters. SPL contains the **return** statement, since requirement F12 states, that NESTML should have functions. The **return** statement returns the result of an expression from a function.

Branching uses an **if**-statement similar to Python's: it starts with an **if**-statement, followed by a boolean test and, after the colon, the corresponding block of statements. The code block for the **if**-branch is followed by zero or more **elif**-clauses with own tests and blocks. An optional **else**-clause finishes the **if**-statement. See Listing 4.8 for an exemplary **if**-statement in SPL and Listing 4.9 for the corresponding **if**-statement in C.

```
1 if <Test1>:
2     <Block1>
3 elif <Test2>:
4     <Block2>
5 else:
6     <Block3>
7 end
```

Listing 4.8: Exemplary **if**-statement in SPL.

```
1 if (<Test1>) {
2   <Block1>
3 } else if (<Test2>) {
4   <Block2>
5 } else {
6   <Block3>
7 }
```

Listing 4.9: Corresponding **if**-statement in C.

Looping is possible using either a **while** loop, that loops until its test becomes false, or a **for**-loop. Listing 4.10 gives an exemplary **while**-loop and Listing 4.11 gives the Corresponding **while**-statement in C.

```
1 while <Test>:
2     <Block>
3 end
```

Listing 4.10: Exemplary **while**-loop in SPL.

```
1 while (<Test>) {
2   <Block>
3 }
```

Listing 4.11: Corresponding **while**-statement in C.

```
1 for <Variable_Name> in <Start_Value> ... <End_Value> [step <Step_Value>]:
2     <Block>
3 end
```

Listing 4.12: Exemplary **for**-loop in SPL.

```
1 for( <Variable_Name> =  <Start_Value>; // the variable needs to be defined beforhand
2      <Variable_Name> <  <End_Value>;   // if the given step_value is negative, then
3                                        // <Variable_Name> > <End_Value> is checked
4      <Variable_Name> += <Step_Value>)  // default step_value is 1
5 {
6   <Block>
7 }
```

Listing 4.13: Corresponding **for**-loop in C.

The concrete syntax of the **for**-loop diverges from the Python syntax, since SPL does not contain support for collections so far. Listing 4.12 contains an exemplary **for**-loop: after the keyword **for** follows the variable name, over which this loop iterates. This variable needs to be defined previously. The keyword **in** starts the definition of a range, similar to the UnitDSL, see Section 4.1, except that the *Start-Value* and *End-Value* can

contain arbitrary numeric expressions. The *Step-Value* after the keyword **step** contains a numeric value, which is added to the iteration-variable after every loop. If the step value is omitted a default value of one is assumed. The **for**-loop starts by assigning the start value to the iteration variable. In every loop it first checks, whether the iteration variable is smaller than the end value, then it executes its block and, finally, it adds the step value to the iteration variable. If the step value is negative, it checks, whether the iteration variable is greater than the end value. Listing 4.13 shows the corresponding **for**-loop in C for the **for**-loop in SPL in Listing 4.12.

Every new block in an **if**-statement, **while**-statement or **for**-statement resembles a new scope for variable definitions. This means, that variables defined inside a scope are only visible inside that scope and scopes defined inside this scope. If variables with the same name are defined in surrounding scopes, these variables are hidden by the variables in the inner scope. Listing 4.14 illustrates this behavior.

```
1 x real = 0.5          // first declaration of x
2 if x > 0:             // start a new scope
3     x integer = 3       // Ok! The integer-x hides the real-x from the outer scope!
4     x string = "Hello" // Error! integer-x and string-x are both in the same scope!
5 end
```

Listing 4.14: Scoping in SPL.

and	elif	else	end	for	if	in	not	or	return	step	while

Table 4.3: The reserved keywords of the SPL.

4.3 NESTML

NESTML is a language specifically designed for the neuroscience domain. It describes neuron models in a compact and expressive way (requirement NF01) and allows model descriptions that are detailed enough, to generate code for simulators from them (requirement NF05). In this section the design, composition and semantics of NESTML and its elements are described in detail. Listing B.6 contains the full grammar for NESTML.

NESTML is a composition of the languages described above. Just like the UnitDSL and SPL it extends the `nestml.literals.Literals`, so that it has access to the same common literals and follows the same general rules for comments and line breaks.

To define new units directly in a NESTML file, NESTML embeds the `UnitLine` production from the UnitDSL. From SPL it embeds the `Block` production, that is used for the blocks in function declarations and the dynamics declaration, and the `Declaration` production, that is used in the state, parameter and internal variable blocks to declare variables of the neuron or component. Figure 4.1 illustrates this composition.

In the following the concrete and abstract syntax of NESTML are described and the semantics of individual elements is outlined. A NESTML file starts with the definition of a package name, e.g. **package** <name>:, and concludes with the keyword `end`. The package name is a dot separated series of names like `models.iaf.leaky` and can be freely chosen, but must not start with `nestml`, which is a package reserved for predefined types. All types defined inside a package definition belong to that package and their

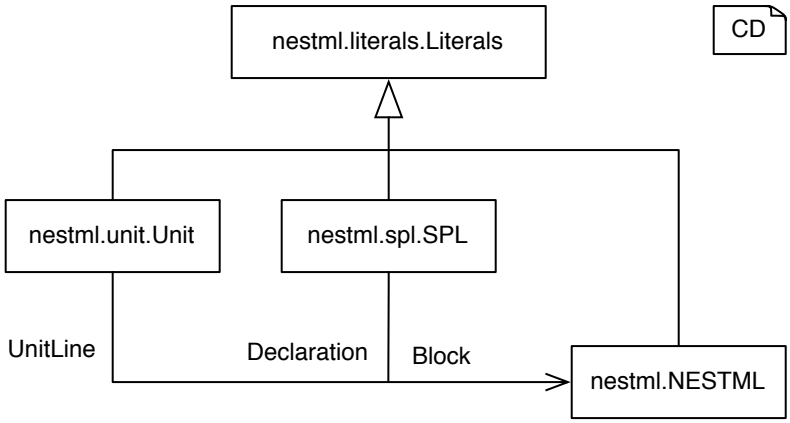


Figure 4.1: The composition of NESTML. The boxes represent the language definitions. *Unit*, *SPL* and *NESTML* all extend the *Literals* language. NESTML embeds the *UnitLine* production from Unit and the *Declaration* and *Block* productions from SPL.

fully-qualified name is composed of their base name prefixed with the package name, e.g. let the neuron `iaf_neuron` be defined inside the package `models.iaf`, then its fully-qualified name would be `models.iaf.iaf_neuron`. A package definition can contain any number of **import** statements, **unit**, **neuron** and **component** definitions. New units can be defined as described in Section 4.1, see line 5 in Listing 4.18. Listing 4.15 shows the general structure of a NESTML file.

```
1 package <pname>:                  // define the package
2     <import-statements>           // import any number of types
3     <unit-definitions>            // declare any number of new units
4     <component-definitions>       // declare any number of new components
5     <neuron-definitions>          // declare any number of new neurons
6 end
```

Listing 4.15: General structure of a NESTML file.

The import statement allows to import unit names and component names from other packages. This way their base name can be used in the definition of neurons and components within this package. Either all names from a package can be imported by putting a star at the end of the package name or exactly one base name from a type can be imported by using its fully-qualified name. See Listing 4.16 for import examples.

```
1 import units.unitless.*    // all names from package unit.unitless can be used
2                            // now, like 'real' and 'integer'
3 import nestml.Logger       // now the name 'Logger' can be used, to access the
4                            // component nestml.Logger
```

Listing 4.16: Import examples for NESTML.

A *component* contains a related set of functions and variables that can be reused in the definition of neuron models (requirement F13). A component definition starts with the keyword **component** and the name of the component. The body of the component definition is opened with a colon and closed with the keyword **end**. Similar to the component definition, the definition of a *neuron* starts with the keyword **neuron** and the name of the neuron. Its body is opened with a colon and closed with the keyword **end**. Listing 4.17 shows the basic productions that are responsible for neuron and compo-

nent definitions. The body elements are *variable blocks*, *input declarations*, *output declarations*, *dynamics declarations*, *function declarations* and *use declarations*.

```
1  Neuron = "neuron" Name Body;        // neurons and components both have an identifying
2  Component = "component" Name Body; // name and a body with basically the same elements
3
4  interface BodyElement; // all elements in the body extend the BodyElement
5  // The body starts with a colon, can have multiple empty lines, line breaks and
6  // body elements and is finished with the keyword end
7  / Body = ":" ( SL_COMMENT! | EOL! | BodyElement)* "end";
```

Listing 4.17: The productions for neurons and components in NESTML.

There are three different kind of *variable blocks*: *state*, *parameter* and *internal*. Every neuron and component is allowed to have at most one *state*-variables block, one *parameter*-variables block and one *internal*-variables block. A variable block starts with one of the keywords **state**, **parameter** or **internal** and starts a block of `AliasDeclarations` with colon and ends it with the keyword **end**.

```
1  package samplePackage:       // the package definition: package name is samplePackage
2
3    import units.time.*        // import the units from package units.time (e.g. ms)
4
5    unit mV Real (-inf ... inf) // declare the unit mV next to the neuron definition
6
7    neuron SampleNeuron1:      // start neuron definition, full name of neuron is
8                               // samplePackage.SampleNeuron1
9      state:                   // start block with state variables
10       V_m mV                 // declare the membrane potential V_m as state variable
11                              // of type mV (millivolt)
12       alias foo mV = 3 * V_m  // alias variable, that always returns three times
13                                // the membrane potential
14     end                      // end of state variables block
15
16     parameter:               // start block with parameter variables
17       V_th mV = -55.0        // declare membrane threshold V_th as parameter
18     end                      // end of parameter variables block
19
20     internal:                // start block with internal variables
21       h ms = Time.resolution() // declare variable h as the simulation time resolution
22     end                      // end of internal variables block
23
24     output: spike            // declare, that this neuron emits spikes
25
26     input:                   // start input block
27       inhSpikes <- inhibitory spike // the buffer inhSpikes receives inhibitory spikes
28       excSpikes <- excitatory spike // the buffer excSpikes receives excitatory spikes
29     end                      // end of input block
30   end                        // end of neuron definition
31 end                          // end of package
```

Listing 4.18: Example neuron in NESTML with state, parameter and internal variables, in- and outputs.

The `AliasDeclarations` in these blocks are similar to variable declarations in SPL (Section 4.2). Variables can have the following types: `nestml.boolean`, `nestml.string` or any unit declared in UnitDSL. In addition to declarations in SPL it is possible, to precede the declaration of a variable with the keyword **alias**. These variables become an alternative

handle to its assigned expression and every use of the variable invokes the expression. Alias variables can be useful to express, that a variable always depends on an other variable, which is for example useful, if the internal value differs from the biophysical property it represents because of efficiency reasons. Declarations of regular variables and alias variables can be seen in Listing 4.18.

The variables declared in the *state* block represent the time dependent state of a neuron or a component, i.e. variables which change over time during simulation (requirement F02). Hence, they are the most interesting values of a neuron model to observe for a neuroscientist. A prominent example would be the membrane potential of a neuron which should be modeled as a state variable. Every attribute of a neuron or component, that does not change over time, but can vary across neurons, should be modeled as a variable in the *parameter* block (requirement F04). Examples for neuron parameters would be its refractory time, its spiking threshold or its size. If the neuron or component needs precalculated values or variables that fit into neither state nor parameter, they should be modeled as an *internal* variable (requirement F05). Parameter and internal variables are not supposed to change during simulation (requirement F04.3) and they are supposed to have an initial value. If no initial value is given the default value for the variable type is assigned (see Section 4.2). Listing 4.18 is the definition of a sample neuron in NESTML. This sample neuron contains state, parameter and internal variables and shows the concepts of package definition and import statements.

A neuron definition is allowed to have a single block of *input* declarations (requirement F09). It starts with the keyword **input** that is followed by a colon. Then several inputs can be declared. The block is finished with the keyword **end**. An input declaration starts with the name for a buffer that should receive the specified input. The input type can either be spikes or electric currents denoted by the keywords **spike** and **current**. Spike input can further be specified to be inhibitory, excitatory or both indicated by the keywords **inhibitory** and **excitatory** in front of the keyword **spike**. If both or no spike type is mentioned the buffer is used for inhibitory and excitatory spikes. The left arrow <– indicates that the input type specified on its right side should go into the buffer on its left side. An example input block is given in Listing 4.18 lines 26 – 29. The production for individual input declarations is shown in Listing 4.19.

```
1 InputLine = Name "<-"       // the buffer name
2             InputType*      // zero or more spike types
3             ([spike:"spike"]|[current:"current"]); // either spike or current input
4
5 // spikes can either be inhibitory or excitatory
6 InputType = ([inh:"inhibitory"]|[exc:"excitatory"]);
```

Listing 4.19: The productions to declare a single input.

The *output* definition of a neuron (requirement F10) starts with the keyword **output** that is followed by a colon. Then the type of the output is stated: NESTML supports spike output, denoted by the keyword **spike**, and electric current output, denoted by the keyword **current**. Listing 4.18 line 24 shows the definition of spike output.

With the possibility to define *functions*, requirement F12 is implemented. The concrete syntax of a function starts with the keyword **function** and then the function name is stated. After that a list of zero or more function parameters can be named. Just like declaring a variable, a parameter is declared by first stating its name and then its

type. Multiple parameters are separated by comma. Behind the parameter list an optional return type can be defined. Finally, the function body is started by a colon and concluded with the keyword **end**. The code inside the function body is a block of SPL code. Listing 4.20 shows the productions responsible for defining functions.

```
1 Function implements BodyElement = "function" Name  // define function name
2                 "(" Parameters? ")"        // optional list of parameters
3                 (returnType:DottedName)?   // optional return type
4                 ":"  Block "end";          // function body is an SPL code block
5 Parameters = Parameter ("," Parameter)*;   // one or more parameters
6 Parameter = Name type:DottedName;          // a parameter has a name and atype
```

Listing 4.20: The productions to define a function in NESTML.

Functions resemble a reusable chunk of code. They can be called in SPL code as separate statements or inside an expression, like in Listing 4.7. The block of SPL code forms a new scope for local variable definitions. The parameters count as variables inside that scope, so no local variables with the same name as a parameter is allowed inside the top-most scope of a function. The type of the result from a function is determined with its return type. If the return type is omitted, the function does not return a value, i.e. its return type is the predefined type `nestml.void`. To return the result from a function, the *return*-statement of SPL is used.

```
1  // previously import the type units.unitless.integer
2  function fib (n integer) integer: // function takes an integer and returns an integer
3      if n == 0:                    // calculate fibunacci number n in function body
4          return 0                  // possibly several return statements
5      elif n == 1:
6          return 1
7      else:
8          return fib(n - 1) + fib(n - 2) // == fib(n)
9      end
10 end
```

Listing 4.21: The Fibonacci function in NESTML.

The *dynamics* definition of a neuron is similar to a function (requirement F03). The definition starts with the keyword **dynamics**. Then the type of the dynamic update is specified, which can be either be **timestep** or **minDelay**. Next a list of parameters surrounded by parentheses can be defined. Finally, a block of SPL code, delimited by a colon and the keyword **end**, describes the neuron dynamics. Listing 4.22 shows an example for the *timestep* dynamic update.

```
1 dynamics timestep (t units.time.ms):  // start dynamics definition
2     <SPL–code>                        // any block of SPL code
3 end                                   // end of dynamics definition
```

Listing 4.22: Exemplary timestep dynamics in NESTML.

If the **timestep** dynamics is specified, the parameter list only contains a single parameter of type units.time.ms. The execution of the timestep dynamics should advance the state of the neuron by one simulation timestep to the current simulation time, which is provided by that parameter. The **minDelay** dynamics is currently just an example for future dynamics descriptions (requirement NF04.2) and is not completely defined yet. The execution of the minDelay dynamics should advance the state of the neuron by the minimum delay of all synapses to the current simulation time.

The purpose of components is to modularize functionality (requirement F13), so they can contain associated functions and variables. Neurons and other components can use them in two ways. Every component has a single, global instance. This instance can be referenced with its (fully-qualified) type name. Its functions and variables can be used with the dot-syntax, e.g. `Math.sin(0)` or `Logger.info("Hello.")`. This is useful for components that do not have a state and supply auxiliary functions, like mathematical or I/O functions.

```
1 use sample.RefractoryComponent as refr
```

Listing 4.23: Use-statement in NESTML.

If components contain a mutable state that should be different for every neuron, a neuron or component can use an own instance of a component by stating an appropriate *use*-statement inside its definition. The statement starts with the keyword **use** followed by the name of the component. After that, the keyword **as** is followed by the referencing name. Listing 4.23 shows a full *use*-statement, in which an instance of the component `sample.RefractoryComponent` is referenced by the name `ref`. Functions and variables of that component can be used with the dot-syntax.

Finally the keyword **structure** is reserved for future extensions to declare the structure of a multi-compartment neuron (requirement NF04.1).

alias	and	as	component	current	dynamics	elif
else	end	excitatory	for	function	if	import
in	inf	inhibitory	input	Integer	internal	minDelay
neuron	not	or	output	package	parameter	Real
return	spike	state	step	structure	timestep	unit
use	while					

Table 4.4: The reserved keywords of NESTML.

4.4 Context Conditions

The following sections list and describe the context conditions for the UnitDSL, the SPL and NESTML. Since NESTML is a composition of those languages, the context conditions of the UnitDSL and the SPL are valid for NESTML as well. Possible severity levels are *warning* and *error*. If a *warning* context condition is violated, only a message is emitted. If an *error* context condition is violated, the further processing after the context condition checking is aborted.

4.4.1 UnitDSL Context Conditions

The context conditions for the UnitDSL make sure, that units are only defined once (U01) and that the range of a unit is valid and non-empty (U02 – U06). The severity level of all these context conditions is *error*.

U01 No two units with the same fully-qualified names are allowed, i.e. within one package every unit has to have a unique name.
Severity: error

U02 Infinity can never be included in a range, because it is neiter part of the real nor of the integer domain. This applies for both the lower and the upper bound. Hence, ranges like [−inf ... inf], [−inf ... <value>] or [<value> ... inf] are not allowed.
Severity: error

U03 Units with integer domain can only have integer values in the range definition. A real value can never be part of the range of an integer unit. This forbids unit-definitions like: unit foobar Integer (1.3 ... 5e3].
Severity: error

U04 The lower bound of a range must be smaller or equal to its upper bound, otherwise the range would be empty. This forbids unit-definitions like:
unit empty Real (10.5e3 ... −2.4].
Severity: error

U05 If the lower and the upper bound of the range are equal, both braces need to be including, otherwise the range would be empty. This forbids a unit-definition like:
unit empty Integer (1 ... 1].
Severity: error

U06 If the upper bound of an integer unit is greater by only one, then at least one of the braces has to be including, otherwise the range would be empty. This forbids a unit-definition like: unit empty Integer (0 ... 1).
Severity: error

4.4.2 SPL Context Conditions

The context conditions of the SPL make sure, that variables are declared correctly before they are used (S01 – S03), that functions are present (S04), that the types of expressions match the variable types in assignments and parameter types in function calls (S05, S06). They check that operators in expressions only get correctly typed operands and that all other statements are type-correct (S07 - S14). All context conditions have severity level error, if they are violated.

The blocks in **if**-/ **elif**-/ **else**-statements, **while**-statements and **for**-statements each introduce a new scope. The variables of an outer scope are hidden by variables in an inner scope with the same name.

S01 Variables with the same name can be declared only once in every scope.
Severity: error

S02 It is not allowed to use or assign variables, that are not declared, since SPL is statically typed and the type of an undeclared variable is unknown.
Severity: error

S03 The declaration of a variable must be prior to its first use or reassignment. This also means, that a variable can not be used in its own initial assignment.
Severity: error

Identifying properties of a function in SPL are both its fully-qualified name and the number of its parameters. Other general programming languages (GPLs) have stricter rules, e.g. C does not allow two functions with the same name, and others allow more ways to overload functions by including parameter types as an identifying property. The way SPL identifies functions is sufficient for most standard situations, helps to understand the control flow, since argument and parameter types need not be compared, and reduces the effort for resolving symbols.

S04 Functions that are called in an SPL program need to be defined.
Severity: error

The type of a mathematical expression can either be `integer` or `real` and the type of a logical expression is `boolean`. String concatenation has the type `string`. The types of a variable and an expression match, if they are the same or if the expression type can be converted into the variable type without loss of information. This conversion is only possible from an `integer` expression to a `real` variable, since the value set of real numbers is (theoretically) a superset of the integer numbers.

S05 In an assignment the type of the expression must match the type of the variable that gets assigned.
Severity: error

S06 When a function is called, the types of the arguments must match the types of the function parameters.
Severity: error

While the SPL grammar allows to put any number of $+$, $-$ and $\sim$ signs in front of a factor (like $\sim -$`x`), multiple occurrences of one type of symbol or simultaneous presence of plus and minus in front of a factor does not increase expressiveness of the language, but decreases readability.

S07 Factors are allowed to have at most one occurrence of any $+$, $-$ and $\sim$ sign. Simultaneous presence of plus and minus is forbidden.
Severity: error

S08 The unary operator `~`, the binary operators `|`, `^`, `&`, `<<`, `>>` and `%` only allow operands of type integer.
Severity: error

S09 The unary operator **not**, the binary operators **or** and **and** only allow operands of type boolean.
Severity: error

If an expression contains a string expression or some string concatenation, the type of the whole expression is `string`. A string can be concatenated to string expressions, to numeric expressions or to boolean expressions with the $+$ sign. Other string operations are currently not provided.

S10 Concatenation of a string with something different than a string expression, numeric expression or boolean expression is not allowed.
Severity: error

S11 Operators other than the $+$ sign can not be used with a string expression as operand, e.g. `"Hello"* 5` is not permitted.
Severity: error

S12 The type of the test in an **if**-, **elif**- and **while**-statement must be of type `boolean`.
Severity: error

S13 The variable in a **for**-loop statement must be of type `real` or `integer`.
Severity: error

S14 The type of the start and end expression and the step value in a **for**-loop must match the variable type.
Severity: error

4.4.3 NESTML Context Conditions

This section contains the NESTML specific context conditions. Since NESTML is composed of the UnitDSL and SPL, the context conditions of those also apply to NESTML.

N01 The package `nestml` is reserved for predefined units and components and is imported automatically in every NESTML model. Users can not name their packages `nestml`, so that types defined inside their package do not get imported unintentionally in other models.
Severity: error

N02 The qualified name in an **import** statement must be a valid package or type name.
Severity: error

N03 Units, neurons and components can not have the same fully-qualified name, since they all are types in NESTML.
Severity: error

N04 There can be at most one state-variable-block, one parameter-variable-block, one internal-variable-block and one structure definition in every component and neuron.
Severity: error

N05 The parameter and internal variables are not supposed to change during simulation time, so reassigning them outside their appropriate setter-function should emit a warning.
Severity: warning

N06 State, parameter or internal variables of a component or neuron can not have the same name, because they live in the same scope. Besides the scopes that are spanned from SPL statements, new scopes are spanned by neuron, component, dynamics and function definitions.
Severity: error

N07 Variables can not have the same name as a unit, neuron or component, whose base name is visible in the same scope or a surrounding scope.
Severity: error

N08 The declaration of a state, parameter or internal variable must be prior to its first use in an assignment of other state, parameter or internal variables. This includes, that a variable can not be used in its initial assignment in the declaration. Additionally, state, parameter or internal variables can only use variables from their own variable block or from the parameter block.
Severity: error

N09 In the declaration of an **alias** variable, it is forbidden to declare more than one variable. Those variables represent an alternative handle for the associated expression and multiple aliases for the same expression do not increase expressiveness or readability.
Severity: error

N10 All **alias** variables need a setter-function in the same neuron or component of the form: **function** set_<variable−name>(x <variable−type>). This setter is internally used, to handle assignments to these **alias** variable. Therefore, the setter-function should manipulate the variables inside the **alias** expression, so that on the next usage of that **alias** variable the assigned value is returned. Those functions can not be anticipated, since the assigned expression can contain various variables that might need to be assigned in the setter-function.
Severity: error

N11 Variables, **alias** variables, function parameters and function return types can only have one of the following types: string, boolean or any unit from the UnitDSL. Neuron, component or reference types are not allowed. The later can only be used in a **use** statement.
Severity: error

N12 Components cannot be used as stand-alone parts of a neuronal network and thus have no **output**.
Severity: error

N13 Neurons can have at most one type of **output**. This can be either spike events or electric current events.
Severity: error

N14 Neurons should have some **output**. If no output is defined, the neuron can not contribute to its neuronal network.
Severity: warning

N15 Neurons should have at most one **input** block with one or more declarations of *inputs*. If no input is defined, the neuron can not receive events from the neuronal network.
Severity: warning

N16 Components are not used as individual parts of a neuronal network and thus have no **input**.
Severity: error

N17 A **use** statement has to reference a component model. It is not allowed to reference a unit, a neuron or another reference.
Severity: error

N18 The buffer from an **input** statement can not be reassigned in any function. These buffers can only be used in a read-only fashion, to get the input events of a specific time by invoking the appropriate fun ction.
Severity: error

N19 Functions inside a neuron or component are identified by their name and their number of parameters. Functions with the same name and number of parameters in the same neuron or component are not allowed. This makes function definitions in NESTML consistent with function usage in SPL (S04).
Severity: error

N20 Components are not used as individual parts of a neuronal network and thus have no **dynamics** function.
Severity: error

N21 Every neuron has exactly one **dynamics** function, that describes the propagation of the state during the simulation, i.e. how input is integrated and output is generated.
Severity: error

N22 The **timestep** variant of the dynamics function must have exactly one parameter of type units.time.ms. This can be used inside the dynamics function to determine the current simulation time in milliseconds.
Severity: error

N23 The parameter-variables of functions and **dynamics** belong to the outermost scope of the associated SPL block and can not be hidden by local variables defined in this scope. In deeper scopes of that block hiding is again possible (see context condition S01).
Severity: error

The following context conditions are simulator dependent and are needed for correct code generation. They are specified in terms of the NEST simulator. Generating code for other simulators requires similar context conditions.

N24 If code for the NEST simulator should be generated, the following function names are reserved, since they are internally used by NEST: `init_state_`, `init_buffers_`, `connect_sender`, `check_connection`, `get_status`, `set_status`, `get_instance`, `update`, `calibrate`, `handle`.
Severity: error

N25 If code for the NEST simulator should be generated, getter-functions for every state, parameter or internal variable are generated. Thus, functions of the form get_<variable−type>() are not allowed.
Severity: error

N26 If code for the NEST simulator should be generated, setter-functions for every non-alias state, parameter or internal variable are generated. Thus, functions of the form set_<variable−type>(x <variable−type>) are not allowed.
Severity: error

Chapter 5

Implementation

This chapter explains the implementation of NESTML, SPL and UnitDSL. The first part gives an overview over the implementation, while detailed aspects are illustrated in the subsequent sections: Section 5.1 shows the symbol table structure, its set-up and its usage. Section 5.2 describes, how the type of an expression is calculated and how this type can be used in context conditions. In Section 5.3 the code generation for the NEST simulator is described.

Every language is developed in an own MontiCore project to separate concerns. The project *nestml-core* contains the literals language and cross-project auxiliary classes. Everything that concerns the UnitDSL and the SPL is located in the projects *nestml-unitFE* and *nestml-splFE*, respectively. The Project *nestmlFE* contains the NESTML language and the composition of the UnitDSL and the SPL into NESTML. The ending *FE* stands for *front end*. Furthermore, the project contains the code generation for the NEST simulator. A library for basic units and predefined components is located in the project *nestmlLib*. The dependencies between the projects can be seen in Figure 5.1 with *nestml-core* as the basis and depending projects on top of it.

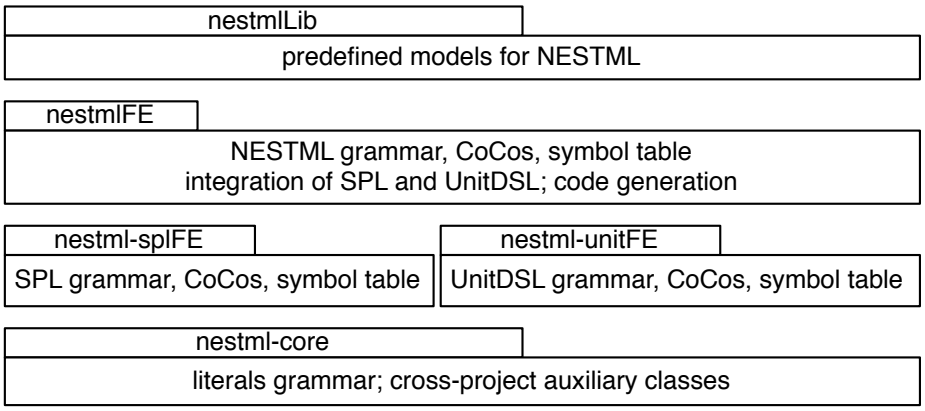


Figure 5.1: The dependencies between projects. The project *nestml-core* is the basis and all further projects depend on it. The projects *nestml-unitFE* and *nestml-splFE* do not overlap, but both are integrated into *nestmlFE*. The models in the project *nestmlLib* are processed by the `NESTMLTool` in *nestmlFE*.

All languages shall have common *syntax elements*, e.g. line breaks can be used as statement delimiter, continuous blocks of code are started with a colon and concluded with the keyword **end**. This means that some predefined concepts of MontiCore can not be used and have to be redesigned.

When line breaks can be used as statement delimiters, single-line comments represent statement delimiters, too, because they continue until the end of the line. The predefined behavior of MontiCore is to skip all white space, including line breaks, and all comments. Hence, to disable this behavior and to enable line breaks and single-line comments as statement delimiter, every language gets the options `nows` (no whitespace) and `noslcomments` (no single-line comments). The literals language contains the new tokens `EOL` and `SL_COMMENT` that represent line breaks and single-line comment and contains a `WS` token that makes sure, that all other white space will be skipped as usual. For consistency, the delimiter for continuous blocks of code are represented by the tokens `BLOCK_OPEN` and `BLOCK_CLOSE`. Listing 5.1 shows the productions for these tokens.

```
1  token WS = (' ' | '\t' | '\f')+
2            {_ttype = Token.SKIP;}; // change the token type to skip all white space
3
4  // allow EOL to be a statement delimiter
5  token EOL = ( options {generateAmbigWarnings=false;}:
6                 "\r\n" // DOS
7               | '\r' // Macintosh
8               | '\n' // Unix
9               ) { newline(); }; // do not skip line breaks
10
11 token SL_COMMENT = "//" (~('\n'|'\r'))* ('\n'!|'\r'!('\n'!)?)?
12     { newline(); } // do not skip line breaks
13     { // omitted part, that links the comment to surrounding nodes
14     };
15
16 token BLOCK_OPEN = ":";      // token that starts a continuous block of code
17 token BLOCK_CLOSE = "end";   // token that concludes a continuous block of code
```

Listing 5.1: Tokens for white space, line breaks and single-line comments.

As a central point of entry, every set of modeling languages based on a grammar has a corresponding implementation of the class `interfaces2.language.ModelingLanguage` that summarizes all relevant parts of that language. This class contains all related workflows of the language and provides the related root class, root factory and filename extension. Additionally, it provides an individual `LanguageComponent`, which has classes related to symbol table processing and context condition checking registered to it. See Figure 5.2 as an example for this structure.

The DSLTool of each language implements a command-line interface for users of that language. Therefore, it has a set of ModelingLanguages that defines, which languages the tool can process – see language aggregation in Section 2.5.3. Additional classes that are needed to glue the languages and symbol tables together are registered directly to the tool. Furthermore, the set of relevant context conditions are specified for the tool. Finally, the parameter processing of the command-line interface is defined inside the tool.

The context condition processing is done very similar in all languages and the following names have a prefix according to the language. Special *checker* interfaces are defined

inside a package `checkers`, e.g. `nestml.ets.checkers` for all NESTML checkers. These interfaces contain a single `check` function for an AST node. The package `cocos` and its sub-packages contain all context conditions of a language. A context condition that performs a check on a specific node, e.g. on `ASTUnit`, extends the class `ContextCondition` and implements the appropriate interface, e.g. the interface `checkers.IUnitChecker`.

A default set of context conditions for a specific language can be obtained from its `DefaultContextConditionCreator` in the `check` package. In addition, the `check` package contains the `CheckVisitor` that, upon creation, gets a set of context conditions and sorts them according to their *checker* interface. Every time the context conditions should be checked, it traverses every node of the AST and performs the `check` functions of all relevant context conditions on that node. A `CheckVisitor` is registered to its corresponding `LanguageComponent` and the context conditions are registered to the tool. Figure 5.2 illustrates the relationships between the DSLTool, the ModelingLanguage, the LanguageComponent and the context conditions using the example of the UnitDSL.

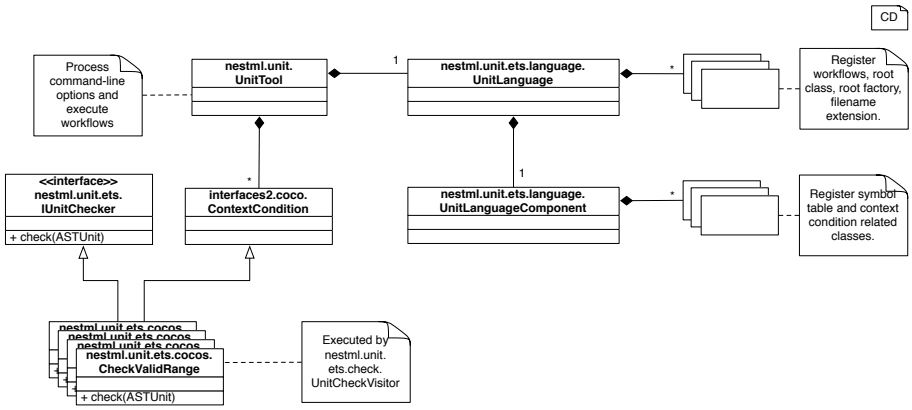


Figure 5.2: Overview of the relationships between the building blocks of a language processing tool with the example of the UnitDSL.

While the previous packages contain grammars and classes to process different languages, the project *nestmlLib* is a library of predefined components and units that are necessary for NESTML. The types `nestml.string` and `nestml.boolean` are provided to represent string and boolean values. The package `units.unitless` contains the basic types for integer and real numbers. Mathematical functions and constants are provided via the component `nestml.Math`. The component `nestml.Time` contains time related functions and constants that are important in the context of neuronal simulations. The component `nestml.Spiking` contains functionality to emit spikes and the component `nestml.Buffer` contains the types for input and functionality to process the received spikes. The packages `units.time` and `units.electric` contain units related to time and electric properties.

The workflow `nestml.workflows.AddImportsWorkflow` adds import statements for the packages `nestml.*` and `units.unitless.*`, if they are not already present, to make sure that all basic types are always available with their base name.

5.1 Symbol Table and Namespaces

The symbol tables of a programming language provide uniform access to all symbols of a program. Symbols are identified by their name within the namespace they are defined in. MontiCore features a comprehensive framework for creating, processing and serializing symbol tables and entries, and we decided to use as much as possible of this framework. See Section 2.5.2 for more detailed information. First, this section gives an overview of the symbol table and namespace structure of NESTML and its associated languages. Then it describes the different types of symbol table entries and how they are exchanged across the languages. Finally, conflicts with the MontiCore symbol table framework, regarding the serialization of symbol tables and the look-up of not yet parsed models, are addressed.

MontiCore has a concept called *compilationunit* that extends a DSL's syntax with a package definition and import statements similar to Java and simplifies symbol table related tasks. We decided to use a different package declaration syntax in NESTML (Section 4.1 and Section 4.3) and introduced the `APackage`-production in the literals language as a common package production for the UnitDSL and NESTML. To conform to the *compilationunit* concept, the package name and the list of import statements must be set in the corresponding root object. A `PackageNameWorkflow` for each language sets the package name and the `AddImportsWorkflow` sets the list of imports.

To implement variable scoping of SPL and NESTML (Section 4.2 and Section 4.3) and to organize neuron models, units and components in a hierarchical way (requirement F14), appropriate AST nodes are declared as namespace establishing nodes in the corresponding `LanguageComponent`. The namespace of such a node contains the symbol tables with the symbols that are defined in subsequent nodes. When a symbol should be looked-up, the resolvers first check the current namespace for a matching name. If none is found, they look in the parent namespaces. This procedure is followed until the topmost namespace is reached. If the name is still unknown there, an error is raised.

Each namespace in MontiCore contains four types of symbol tables [Völ11], of which two are used by NESTML: all types, functions and most variables are public and visible for other models and the entries are in the *exported* symbol tables. Symbols that are imported via the import statement and variables inside function definitions should not be visible to other models and the entries are in the *imported* symbol tables.

Each language has its own symbol table entries. Figure 5.3 shows all language related entries. Every entry extends the `AbstractSTEntry`, which contains some common variables, e.g. the name or functionality (like equality and serialization). The UnitDSL only provides new unit types and, hence, it has only type entries. NESTML and SPL provide type, variable and method entries that contain various types of context information, e.g. at which position a variable is defined or which functions a type contains. Each entry is created from a designated entry visitor that traverses the AST, creates symbol table entries, when a proper node is visited and stores them in the symbol table of the appropriate namespace.

To exchange entries between languages, NESTML has several adapters [Gam+95]. The `UnitEntry2NESTMLEntry_Adapter` makes units defined in the UnitDSL available as types in NESTML. Appropriate qualifiers create the adapters for every `UnitEntry` in a symbol table. The context conditions from SPL should check correctness of SPL blocks in

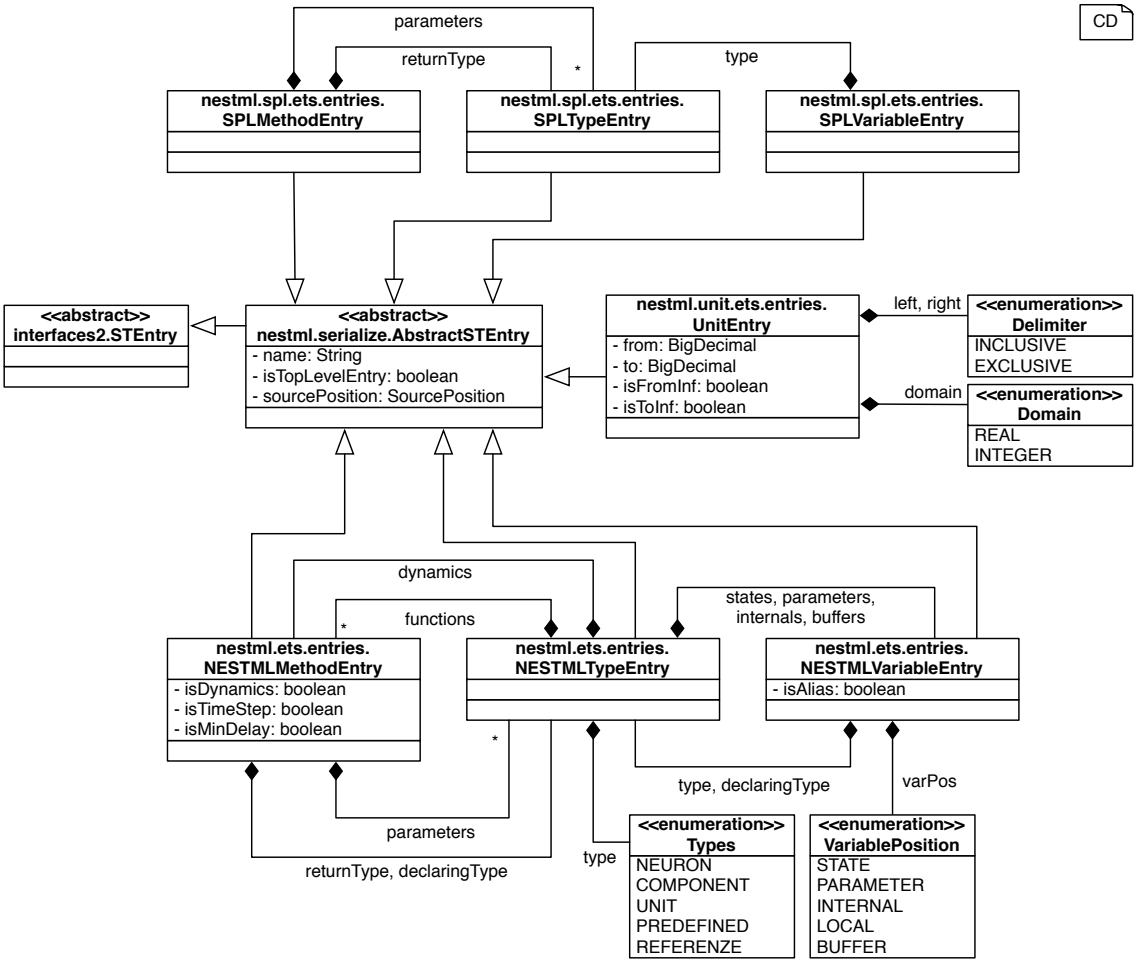


Figure 5.3: The symbol table entries. All entries extend the `AbstractSTEntry` (middle). At the top are the type-, method- and variable-entry classes for SPL and at the bottom are the corresponding entry classes for NESTML. To the right is the `UnitEntry` for the UnitDSL.

NESTML, but they require SPL entries. Hence, the package `nestml.nestml2spl` contains adapters that make types, methods and variables from NESTML available as corresponding SPL entries. Appropriate entry resolvers in the same package create these adapters, whenever an SPL entry is required. The adapter principle is illustrated in Figure 5.4.

It is inefficient to hold the symbol tables for every model in memory or to parse the whole model again, if only some information of a symbol is required. Therefore, the symbols are serialized in symbol table files. MontiCore provides an annotation based `STEntrySerializer` that automatically transforms a correctly annotated entry into a predefined symbol table entry format. Corresponding deserializers reverse the transformation and extract a symbol table entry from the predefined format.

MontiCore's symbol table framework expects a certain structure for handling models defined in a DSL that uses the *compilationunit* concept. These structures are very similar to Java's handling of classes and interfaces. The structures are:

- Every file has a single top-level model with a name. The symbol table entry of

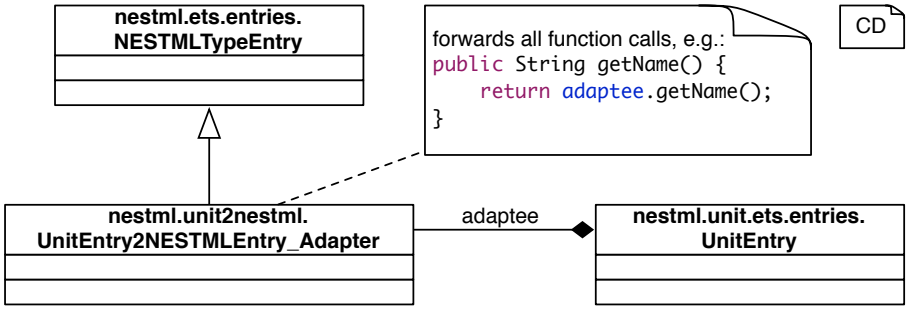


Figure 5.4: The adapter principle with the example of the `UnitEntry2NESTMLEntry_Adapter`. The adapter extends the `NESTMLTypeEntry` and contains a `UnitEntry` member variable (the adaptee). All calls to the adaptor are delegated to the adaptee.

this model is serialized in a single file inside the symbol table directory: `<symtab>/package/name/modelName.st`

- The file has the same name as the top-level model and is stored in a folder structure corresponding to its package name. This helps to find symbols of not yet parsed files by looking up all model-paths.

Since we want to use the framework, but do not have such strict requirements, some of the serializing and look-up workflows were adapted. NESTML and the UnitDSL allow defining multiple top-level models in a single file, so the variable `isTopLevelEntry` of the class `AbstractSTEntry` states, whether an entry should be serialized. A modified `SerializeWorkflow` iterates all exported symbol tables and serializes all entries where `isTopLevelEntry` equals true.

NESTML has no restrictions on file naming and storage positions of files. To find symbols of not yet parsed files, a level of indirection is introduced. Before the *NESTMLTool* starts parsing and processing files, it executes the *NESTMLModelNameTool* that processes all files in the model path. This tool uses the `ModelNameWorkflowWithFilename` to generate so-called model-name files at the correct path – according to the package of the model – inside the symbol table directory for every top-level entry. This file is named after the top-level entry and contains the full path to the file that declares this model. For example, a component with fully-qualified name `sample.components.A` defined inside the file */Users/Name/models/severalComponents.nestml* would produce a model-name file *A-NESTML.mn* in the directory *<symtab>/sample/components* with the path */Users/Name/models/severalComponents.nestml* as content.

Whenever a symbol that is defined outside the current file should be looked-up, the `NESTMLModelLoader` checks, whether a symbol table entry is already defined in the symbol table directory. If not, it looks for the model-name file of that symbol, extracts the contained path and parses the file pointed to by the path. This generates the symbol table entries for future look-ups.

5.2 Type Calculation with an Attribute Grammar

Many context conditions require that the type of a part of an expression or of the whole expression is evaluated and compared to an other type. For example the argument type of a function has to match the corresponding parameter type (context condition S06) and the binary operator | (bitwise OR) requires its operands to be of integer type (context condition S08).

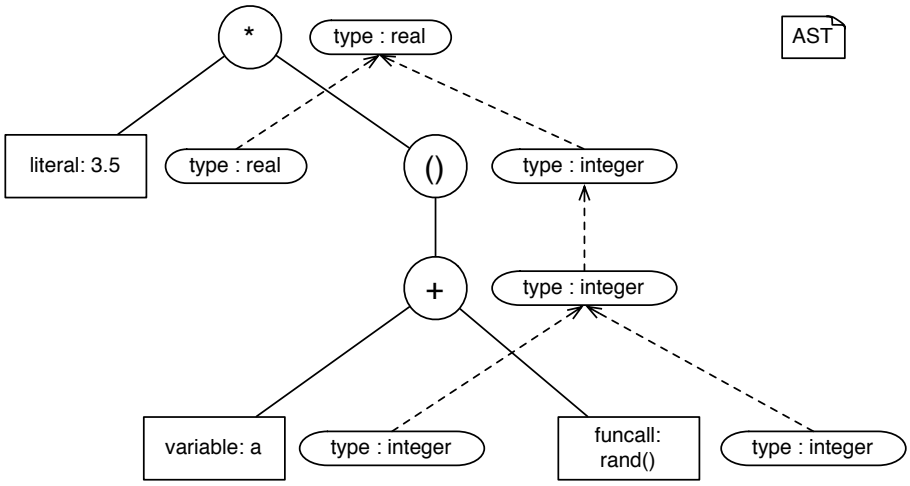


Figure 5.5: A reduced AST for the expression $3.5 * (a + rand())$ with the synthesized attribute `type`. The variable a is of type `integer` and the function $rand()$ has the return type integer.

Attribute grammars are well suited for type calculations, because of their ability to calculate synthesized attributes of the AST in a bottom-up way (see Section 2.5.4). Figure 5.5 shows a reduced AST for the expression $3.5 * (a + rand())$. This attribute grammar solely has the synthesized attribute `type`. The synthesized rules for the different expression productions will be informally discussed in the following. The implementation of these rules for SPL expressions will be discussed afterwards.

a	b	$-a$ / $+a$	$\sim a$	$a + b$	$a * *b$	$a\ op1\ b$	$a\ op2\ b$
`integer`	`integer`	`integer`	`integer`	`integer`	`real`	`integer`	`integer`
`integer`	`real`	`integer`	`integer`	`real`	`real`	`real`	*error*
`integer`	`string`	`integer`	`integer`	`string`	*error*	*error*	*error*
`real`	`string`	`real`	*error*	`string`	*error*	*error*	*error*
`real`	*numeric*	`real`	*error*	`real`	`real`	`real`	*error*
`string`	__	*error*	*error*	`string`	*error*	*error*	*error*

Table 5.1: The resulting type of mathematical operations with a and b and $op1 \in \{-, *, /\}$ and $op2 \in \{\%, \ll, \gg, |, \&, \wedge\}$. Other types of a and b result in errors. The type *numeric* can either be `integer` or `real`.

The type of the leaf of an expression can be calculated very easily: if it is a literal, the type corresponds to the literal type. The leftmost leaf of Figure 5.5 is the *real* literal 3.5 and thus of type `real`. If the leaf is a variable, the type of the leaf corresponds to the variable type. In Figure 5.5 the variable a is of type *integer* and thus the leaf is of

type `integer`. Finally, if the leaf is a function call, the type of the leaf corresponds to the return type of the function.

To calculate the type of the node of an unary and binary operator, the type of its operands has to be considered. Table 5.1 contains the rules for unary and binary mathematical operators with different types for the operands a and b. Figure 5.2 contains similar rules for comparisons and unary and binary boolean operators. All other, intermediate nodes just pass the type of child nodes up to the parent node.

a	b	**not** a	$a\ op1\ b$	$a\ op2\ b$
numeric	*numeric*	*error*	boolean	*error*
boolean	boolean	boolean	*error*	boolean

Table 5.2: The resulting type of boolean and comparison operations with a and b and $op1 \in \{<, <=, >, >=, ==, !=, <>\}$ and $op2 \in$ {**and**, **or**}. Other types of a and b result in errors. The type *numeric* can either be `integer` or `real`.

Since only the SPL has relevant expression productions, the implementation of the attribute grammar is mostly located there. Listing 5.2 shows the definition of the attributes and its corresponding calculators in the SPL grammar and language file. It has a global attribute `SymbolTable` to resolve the type of variables and function calls. Every node has the synthesized attribute `typeEntry` that represents the type of the associated expression.

```
1  // in the grammar
2  concept attributes {
3    // define the synthesized attribute typeEntry of type SPLTypeEntry
4    syn typeEntry: /nestml.spl.ets.entries.SPLTypeEntry;
5
6    // many rules require the type of some symbol, hence the grammar gets
7    // the global attribute SymbolTable, to resolve symbols
8    global SymbolTable: /interfaces2.helper.SymbolTableInterface;
9  }
10 // in the language file
11 concept Attributes {
12   SymbolTable { // global attributs do not get a calculator
13     nestml.spl.SPL.SymbolTable
14   }
15   TypeEntry {  // specify the calculator for the TypeEntry attribute
16     nestml.spl.SPL.typeEntry = /nestml.spl.attribute.TypeEntryCalculator
17   }
18 }
```

Listing 5.2: Definition of the attributes in the SPL grammar and language file.

All the above-mentioned rules are implemented inside the `TypeEntryCalculator`. The `TypeChecker` performs type comparisons. The `SPLToolConcreteStorageConnector` caches the calculated attributes of every node. If the attribute of a node is requested, but the attribute is not yet calculated, it forwards the calculation to the `TypeEntryCalculator`. The `InitAttributeStorageWorkflow` initializes an instance of this storage connector with a `SymbolTableInterface` and stores the connector as an annotation in the corresponding root object. The `SymbolTableInterface` is necessary to resolve the type of a variable or the return type of a function with help of the symbol table. Whenever the type of an expression is needed, the connector for that node can be received from

the `SPLStorageConnectorHelper` and its `getTypeEntry(ASTNode)` function will return the type of the passed node.

The tool for NESTML contains a similar infrastructure to calculate types, but actual type calculations are forwarded to the SPL infrastructure. Type translations from NESTML to SPL are done with the adapters and appropriate resolvers described in Section 5.1. The SPL type returned from the SPL type calculations is converted into an NESTML type by either extracting the adoptee, if the SPL type is actually an adapter, or by creating a corresponding NESTML type for the SPL type, e.g. an SPL `integer` becomes a NESTML `units.unitless.integer`.

These type calculations are very limited for now, but are sufficient to correctly perform computations. The same approach can be used to calculate the correct physical unit of an expression and thus, provide additional and more sophisticated checking for users of NESTML. Required for the extended type calculations is the composition of unit types as separate types (like $mV * ms$ or mV/ms) and reduction and comparison of such composed unit types, e.g. mV/ms and V/s are the same types. If this requirement is fulfilled, appropriate rules for the extended type calculations can be implemented. Multiplication, exponentiation and division results in a composition of units. All other operators require operands of equivalent unit type.

5.3 NEST – Code Generation

This section illustrates the code generation capabilities of the NESTML processing tool. As a first target platform we decided to generate code for the NEST simulator [GD07]. This means that a neuron model described in NESTML can be transformed into a neuron model that the NEST simulator can use to simulate its dynamics. This is described in Section 5.3.4. Furthermore, NESTML allows describing other entities, such as units and components. Adequate transformations for them are described in Section 5.3.2 and Section 5.3.4. For convenient usage of the generated code, the tool offers two generation modes, which are described in Section 5.3.1. An overview over the code generation is given in Figure 5.6.

The `NESTMLTool` is a subclass of the `GenerationTool`, which is provided by the MontiCore framework (see Section 2.5.5). The `GenerationTool` contains a generic `CodegenWorkflow` with a generic `CodegenVisitor`. The tool specifies combinations of AST nodes and FreeMarker templates, and whenever the visitor traverses one of those nodes, it executes the corresponding template (see Section 2.5.5).

The NEST simulator is a high performance neuronal network simulator implemented in the programming language C++ (see Section 2.2). Thus, most of the upcoming templates will contain C++ code with FreeMarker code embedded. All templates for the NESTML code generation reside in the package `nestml.nest` and have the file ending `ftl`. The package prefix and the file ending are omitted for template names in the following description for brevity.

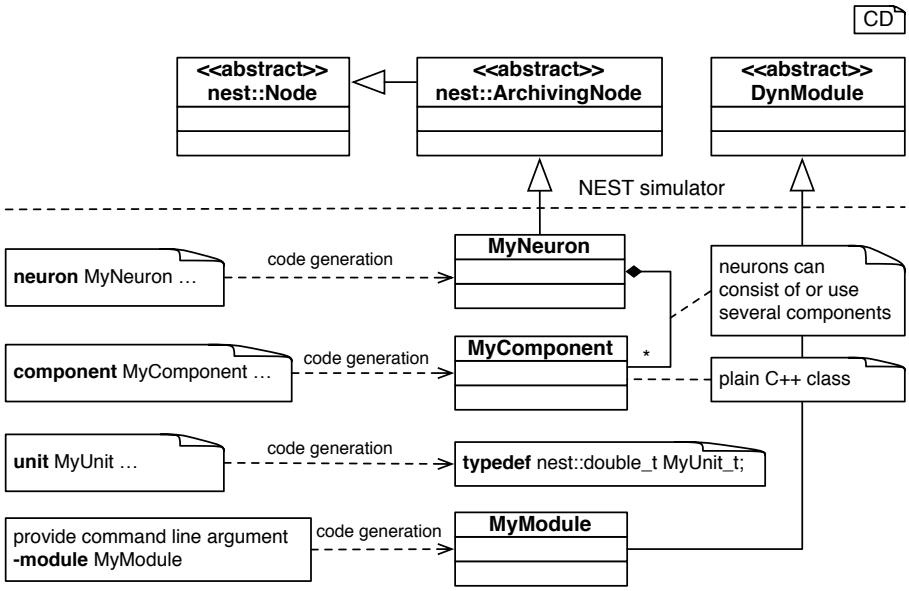


Figure 5.6: Overview over the code generation for NEST. Model descriptions and command line arguments result in the generation of corresponding classes, which are embedded into the NEST simulator architecture.

5.3.1 Module Infrastructure

The NEST simulator has two methods to introduce new neuron models to its simulation kernel. First, the files for new neuron model and all auxiliary files are copied into the *models*-folder inside of NEST's source directory. Then the generated files have to be registered in the file *Makefile.am*, so that they are eventually compiled. Next, the new models have to be registered within the file *modelsmodule.cpp* to be available to the simulation kernel. Finally, the NEST simulator has to be re-compiled.

The second method is to create a plug-in or *module*, which the NEST simulator can dynamically load at runtime. This has the advantage that the code base of the NEST simulator does not have to be modified, all configuration and build files can be generated from a neuron model. The module infrastructure of NEST entails has some *prerequisites*. It requires a successful installation of the NEST simulator, with its source and installation directories accessible. It requires the installation of a current version of the GNU Autotools [Cal10] and the environment variable `NEST_INSTALL_DIR` set to the path, to which the NEST simulator is installed. Compiling the generated module simply requires the following commands to be executed in the terminal:

```
1 $ cd <module folder>          # go to the module folder
2 $ ./bootstrap.sh              # this bootstraps the building infrastructure
3 $ cd ..
4 $ mkdir <build folder>        # create a new folder, in which the compilation
5 $ cd <build folder>           # output should go and change into that folder
6 $ ../<Module folder>/configure --with-nest=${NEST_INSTALL_DIR}/bin/nest-config
7 $ make                        # start the compilation with the usual configure,
8 $ make install                # make, make install commands
```

A NEST module basically consists of five files. The file *bootstrap.sh* contains code to bootstrap the building environment with the help of the GNU Autotools. The file *Make-*

file.am contains amongst others the names of the source files that should be compiled. The file *configure.ac* contains the code for the build configuration. The last two files will be used during bootstrapping to create appropriate makefiles and configuration files. Finally, the module contains the header and the implementation file of the module class. This class is the interface for the simulation kernel and is responsible for registering the new neuron models. Figure 5.7 contains an overview of a complete module of name *MyModule*. The module header and the implementation file is named after the module name, but with lowercase letters only.

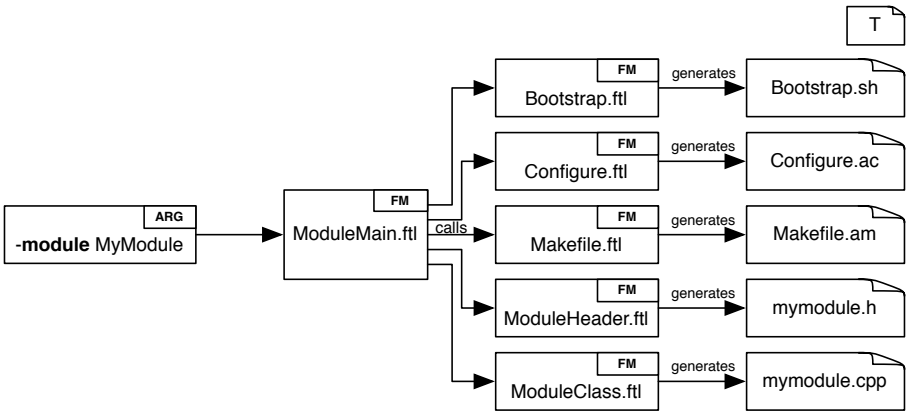


Figure 5.7: Overview of a complete, generated module with name *MyModule*. These files become generated inside the folder *MyModule*. All further generated files end up in the same folder.

The central template for the module infrastructure is `ModuleMain`, which will be called whenever a package node of the AST is traversed. It sets various global variables that are needed by all subsequent templates, including the name of the module, which can be specified as a command line argument (see Section 6.1). Since the generation of the module infrastructure is not mandatory and to reduce repetitive generation of the module for every NESTML file, the template makes sure that the module is generated exactly once.

The *Makefile.am* and the module class require to know, which sources will be there to compile and which neuron models have to be registered, before the module can be generated – potentially before all NESTML models are processed. Therefore, the template `ModuleMain` executes the `ModuleNamesCalculator` that scans the symbol table directory for all model-name files (Section 5.1). Those are created for all models, before the first model is actually processed. The names of those models are stored in the variable `module_names`. In addition to the path of the NESTML file, any model-name file contains the type of a model: either *neuron*, *component* or *unit*. Hence, the variable `neuron_names` stores all neuron names that have to be registered in the module class.

Within the template `ModuleMain` the templates for the above-described files will be called. Each file will be generated in a folder named after the module inside the output directory. The templates `module.Bootstrap`, `module.Configure` and `module.ModuleHeader` are responsible for generating the files Makefile.am, configure.ac and the module header file. The content of those files is very similar for all modules and with FreeMarker only some names are adjusted to the module name.

The template `module.Makefile` is responsible for generating the file Makefile.am. Be-

sides adjusting names to the module name, it lists the source files that should be compiled. Listing 5.3 contains the relevant part, that iterates through all model names and adds the exact position of its header and implementation source to the variable `${lowerModuleName}_la_SOURCES`.

```
1 ${lowerModuleName}_la_SOURCES= ${lowerModuleName}.cpp ${lowerModuleName}.h \
2 <#list module_names as name>
3     ${name?replace(".","/")}.cpp ${name?replace(".","/")}.h \
4 </#list>
5     # last line can not be empty, because of the last \ of the sources
```

Listing 5.3: Add all relevant source files to the `Makefile` template.

The template `module.ModuleClass` is responsible for generating the implementation file of the module class. Like the header file, the implementation file is named after the module in lower case letters. The module class itself has the same name as the module, hence most FreeMarker code adjust names in the implementation file. There are two spots in the template that iterate all neuron names: at the top of the template the header files of the neuron models are included and inside the `init`-function at the end all neuron models are registered to the NEST kernel.

5.3.2 UnitDSL Code Generation

This section describes the translation from a unit defined with the UnitDSL to a representation for the NEST simulator. Units are used as numeric types in NESTML and all mathematical operations are performed on them. In the NEST simulator units can either be modeled as primitive types of C++ or as separate classes. Representing units with classes has several drawbacks. Performing mathematical operations on objects involves a function call for every operation and the object size can vary between compilers. This decreases the overall execution performance and increases the memory footprint. This is contrary to requirement NF07. Hence, units are represented as primitive types of C++. The unit name is used in a `typedef` statement as the alias for a type specified by the domain.

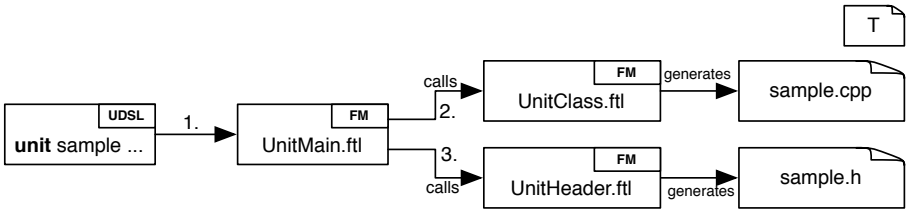


Figure 5.8: Overview over the generation templates for unit definitions.

As a starting point, the template `UnitMain` is called for every unit definition (1. arrow in Figure 5.8). It calls the `unit.UnitClass` and `unit.UnitHeader` templates (2. and 3. arrow) and performs common calculations. The `DomainCalculator` specifies the exact type of the unit representation according to the unit domain: a unit with the `Integer` domain has the type `nest::long_t` and the `Real` domain has the type `nest::double_t`. The `OperationCalculator` calculates C++ representations of the unit range limits and appropriate comparison operators. Those are used in a `check`-function that verifies, if some value is in the unit range.

```
1 <#if op.callCalculator("nestml.codegen.units.UnitPrettyPrintCalculator")>
2 // ${UNIT_MODEL}                          // print full unit definition as comment
3 </#if>
4 typedef ${type} ${ast.getUnitName()}_t; // let the unit name be an alias for ${type}
```

Listing 5.4: Type definition for a unit definition.

The template `unit.UnitHeader` contains the type definition and the prototype of the `check`-function inside a namespace that matches the unit package. Listing 5.4 shows the type definition. Since a unit definition is rather short, the complete unit definition is pretty-printed by the `UnitPrettyPrintCalculator` (line 1) and included as a comment (line 2).

The template `unit.UnitClass` contains the implementation of the `check`-function, which can be seen in Listing 5.5. The namespace is stored in the `nspPrefix` variable. The comparison operators are stored in the variables `from_op` and `to_op` and the left and right limits of the range are stored in the variables `from_value` and `to_value`. The generated files are stored in a folder hierarchy according to the units package inside the module package, if the module is generated. The file names correspond to the unit name.

The generated code for a unit definition – in this case for the `sampleUnit` in Listing E.1 – can be seen in Listing E.2, which contains the generated header file, and Listing E.4, which contains the generated implementation file.

```
1  bool ${nspPrefix}::check_${ast.getUnitName()} // separate check function for all units
2                          (const ${nspPrefix}::${ast.getUnitName()}_t &var) {
3      // assume var is inside the range
4      bool result = true;
5  <#if !ast.isFromInf() > // if left limit is not infinity
6      // test, if var is smaller than left limit
7      if (var ${from_op} ${from_value} ) { result = false; }
8  </#if>
9
10 <#if !ast.isToInf() > // if right limit is not infinity
11     // test, if var is bigger than right limit
12     if (var ${to_op} ${to_value} ) { result = false; }
13 </#if>
14     return result;
15 }
```

Listing 5.5: Implementation of the unit range `check`-function.

5.3.3 SPL translation

This section describes the translation of SPL statements into corresponding C++ statements. NESTML uses blocks of SPL code and SPL declarations in various positions. First, the translation of SPL expressions and tests is described. After that the translation of the simple statements *declaration*, *assignment*, *function calls* and *return statements* is discussed. Finally, the translation of the compound statements *for* loop, *if* branching and *while* loop are described.

Many SPL *statements* contain mathematical *expressions* and boolean *tests*. Each expression and test is translated with the `spl.ExprStatement` and `spl.TestStatement` templates, respectively. Both have a hierarchical order in the abstract syntax, which is

reflected in the templates. For every production in the grammar of an expression, a template in the package `spl.expr` exists. Likewise, the package `spl.test` contains a template for every production in the grammar of a test.

```
1  <#--XOR_Expr = AND_Expr ("^" AND_Expr)*;--> // the XOR_Expr production as comment
2  <#assign sep = "">
3  <#list ast.getAND_Expr() as and> // iterate all sub-expressions
4      ${sep}                       // after the first 'and' insert the sign and call
5                                   // the corresponding template for the sub-expression
6      ${op.includeTemplates("nestml.nest.spl.expr.ANDExpr", and)}
7      <#assign sep = " ^ ">
8  </#list>
```

Listing 5.6: The template for XOR expressions.

The structure of the expression and test templates is very similar. To realize operator precedence in the AST, sub-productions of expressions have either a single operator or differentiate multiple operators with the same precedence between operands (see Section 4.2). If a production has only a single operator between the operands, like the `XOR_Expr` production, the template iterates all operands, puts the correct operator between them and calls the corresponding template for the operands. Listing 5.6 shows the full `spl.expr.XORExpr`-template. If the production differentiates multiple operands with the same precedence between operands, like the `Term` production, its template calls the corresponding template for the first operand. If the production continues, it calls a special template for all further operands, which determines the correct operator between the operands and calls the template for the operand. Listing 5.7 shows the templates `spl.expr.TermExpr` and `spl.expr.TermExprEnd` for the productions `Term` and `TermEnd`.

```
1  // Term.ftl:
2  <#--Term = Factor (TermEnd)*;--> // the Term production as comment
3  // call template for first operand
4  ${op.includeTemplates("nestml.nest.spl.expr.FactorExpr", ast.getFactor())}
5  <#if ast.getTermEnd()??>  // if expression continues with terms,
6      // call End-template for every TermEnd in the ast.getTermEnd()-list
7      ${op.includeTemplates("nestml.nest.spl.expr.TermExprEnd", ast.getTermEnd())}
8  </#if>
9
10 // TermExprEnd.ftl:
11 <#--TermEnd = sign:["*"|"/"|"%"] Factor;--> // the TermEnd production as comment
12 <#if op.callCalculator("nestml.codegen.spl.SignCalculator")>
13     ${SIGN_OP}                       // let SignCalculator determine the sign
14                                      // and call template for sub-expressions
15     ${op.includeTemplates("nestml.nest.spl.expr.FactorExpr", ast.getFactor())}
16 </#if>
```

Listing 5.7: The template for Term-expressions.

The templates for the productions `Atom` and the `NOT_Test` basically route to the template that corresponds to the `Atom`- or `NOT_Test`-type. Literals are rendered through the template `spl.expr.LiteralExpr` with the help of the `LiteralCalculator`. For boolean and numeric literals the `LiteralCalculator` simply returns the source from the SPL code, but string literals are converted to `std::string` objects, so that string concatenation is simplified.

Variable names are rendered with the template `spl.expr.VarExpr`, which uses the calculator `VarNameCalculator` to access the variable correctly: local variables are accessed

through their name. Member variables of a neuron or component are accessed through their `getter`-function. Section 5.3.4 discusses these `getter`-function in more detail. Function calls are rendered with the `spl.expr.FunCallExpr` template that uses the calculator `FunctionCallCalculator` to get the full function-name and calls the template `spl.ExprStatement` for every argument, which is also called for bracket terms and inside comparisons.

Almost all operators in SPL have a corresponding operator in C++, which in most cases even has the same name. The differences are:

- C++ does not have the potentiation operator, so the function `pow` from the `cmath` header is used.
- The operator **and**, **or** and **not** translate to `&&`, `||` and `!`, respectively.
- Concatenation of strings takes advantage of the overloaded $+$ method of the class `std::string`. Concatenation of strings and integer, real or boolean value uses separate $+$ methods defined in the auxiliary header `StringConcatHeader.h`.

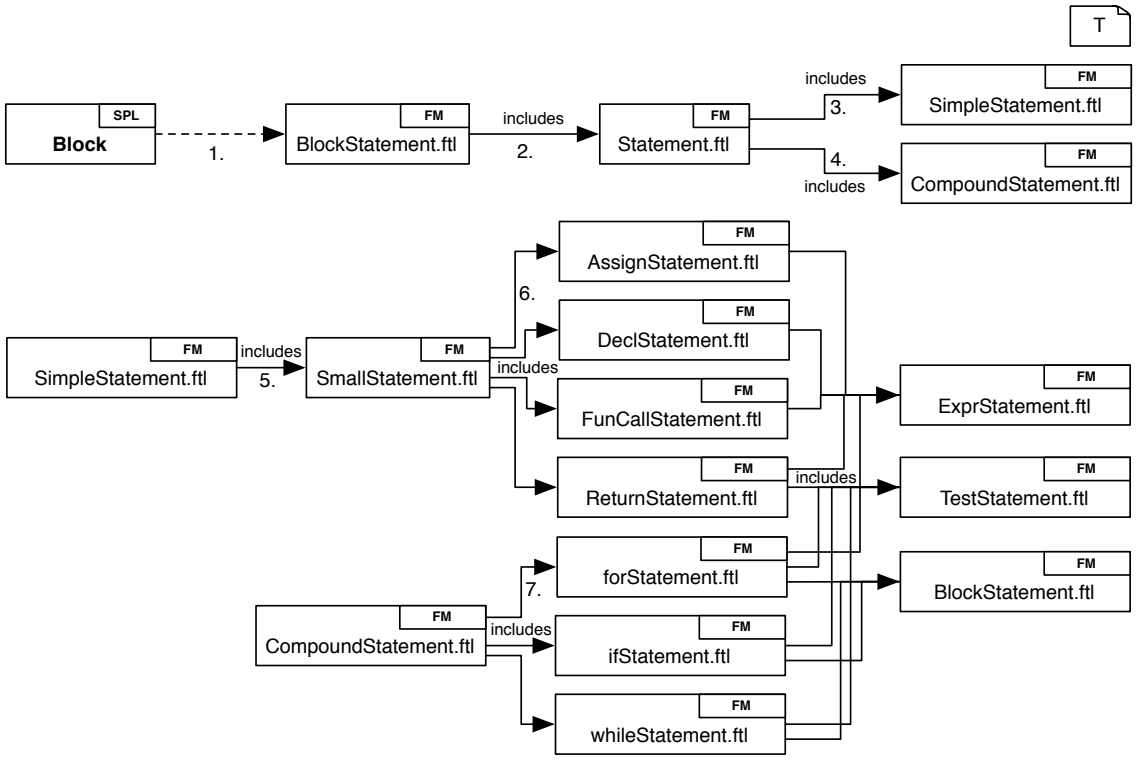


Figure 5.9: Overview over the generation templates for SPL blocks.

Whenever a *block* of SPL code should be translated into a block of C++ code, the template `spl.BlockStatement` is called (1. arrow in Figure 5.9). It basically iterates all contained statements and calls the template `spl.Statment` for each statement (2. arrow). The template `spl.Statment` and subsequent templates route to the template that corresponds to the statement type (3. – 7. arrow).

Declarations of local variables are rendered with the template `spl.DeclStatement`. The `DeclarationCalculator` calculates the corresponding NEST type of the variables and

renders a C++ variable declaration for each. If the SPL declaration assigns an initial expression to the variables, this expression is printed in addition for every C++ variable declaration.

Assignments to variables are rendered with the template `spl.AssignStatement`. The calculator `VarNameCalculator` is again used to access the variable correctly: local variables are assigned directly, while member variables of a neuron or component are assigned through their `setter`-function.

Function calls as separate statements are rendered similar to function calls in expressions. The `spl.FunCallStatement` template uses the `FunctionCallCalculator` to get the full function-name and calls the `spl.ExprStatement` template for every argument. The difference is that this template handles indentation correctly and ends the statement with a semicolon.

```
1 <#--ReturnStmt = "return" (Expr | Test);--> // the return production
2 ${indent}<@compress single_line=true>       // use indent-variable to indent
3 return                // correctly and use @compress directive, to render the following
4 <#if ast.getExpr()??> // in one line (these comments are not part of the template
5     ${op.includeTemplates("ExprStatement", ast.getExpr())} // render expression
6 <#elseif ast.getTest()??>
7     ${op.includeTemplates("TestStatement", ast.getTest())} // or test
8 </#if> ;          // end of return statement
9 </@compress>
```

Listing 5.8: The template for return-statements.

The *return* statement is rendered with the template `spl.ReturnStatement` and forwards the rendering of the expression or test to the appropriate template. All simple statements can be indented correctly, if the variable `indent` contains the whitespace for indentation. See Listing 5.8 as an example for indentation of simple statements.

SPL *for* statements are translated into a C++ *for* statements via the `spl.forStatement` template and the `ForCalculator`. The *from* expression is assigned to the *variable name* in the initialisation part of the C++ *for* loop. In the testing part of the C++ *for* loop the variable is compared to the *to* expression – depending on the *step* value, either $>$ or $<$ is used. Finally, the variable is increased by the *step* value. The loop body is rendered via the template `spl.BlockStatement`.

SPL *if* statements are translated into C++ *if* statements via the `spl.ifStatement` template. All tests are rendered with the template `spl.TestStatement` and all branch bodies are rendered via the template `spl.BlockStatement`.

```
1 // indent while loop with previously defined indent-variable
2 ${indent}while (${op.includeTemplates("TestStatement", ast.getTest())}) {
3 ${op.setValue("indent", indent + "␣␣")}                  // increase indent inside loop
4 ${op.includeTemplates("BlockStatement", ast.getBlock())} // render loop body
5 ${op.setValue("indent", indent?substring(2))}            // decrease indent outside loop
6 ${indent}} /* while end */
```

Listing 5.9: The template for while-statements.

The `spl.whileStatement` template translates SPL *while* statements into C++ *while* statements. The test is rendered with the template `spl.TestStatement` and the loop body is rendered with the template `spl.BlockStatement`. Additionally, all templates of compound statements render their statements with correct indentation. When the template

for a compound statement is called, the variable `indent` should be declared previously and should contain the whitespace for indentation. Listing 5.9 illustrates the indentation in the `spl.whileStatement`.

5.3.4 Neuron Models

In this section the translation of a neuron model in NESTML to a neuron model for the NEST simulator is described. Neuron models for the NEST simulator are basically C++ classes, which extend the class `nest::Node` and represents a single node in a neuronal network graph. It provides the interface for updating the dynamic state, connecting to other nodes, using particular Events, accepting connection requests and handling incoming events. The subclass `nest::Archiving_Node` extends the capabilities of nodes by recording and managing a spike history for use by synaptic plasticity rules that are based on the relative timing between source and target nodes.

First, the generation of NEST neuron classes is described in general. Then the translation of variable blocks into corresponding `structs` with initialization and appropriate *getter*- and *setter*-functions is outlined. After that the function generation is illustrated, followed by a description of the necessary code for output and input and the translation of the *dynamics* into the *update* function. Finally, the generation of components and more general NEST related issues are described.

The Listing C.1 contains the full implementation of the *integrate-and-fire* neuron model in NESTML as described in Section 6.2. The complete, generated header and implementation files for this model can be seen in Listing D.1 and Listing D.14, respectively – for comparison with the generation steps described below.

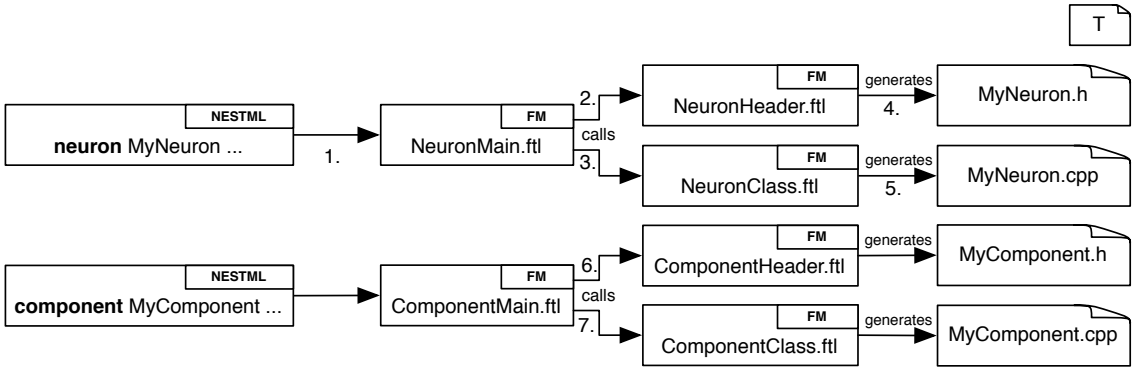


Figure 5.10: Overview over the generation templates for NESTML neurons and components.

The template `NeuronMain` is the main entry point for the code generation of NEST neuron models (1. arrow in Figure 5.10) and is called for every NESTML neuron model. It performs common calculations, e.g. it determines the package name used as the namespace of the neuron model. It calls the templates `neuron.NeuronHeader` (2. arrow) and `neuron.NeuronClass` (3. arrow), which are responsible for generating the class header and class implementation files for that neuron model (arrow 4. and 5.). The generated files are stored in a folder hierarchy according to the neuron package inside the module package, if the module is generated. The file names correspond to the neuron name.

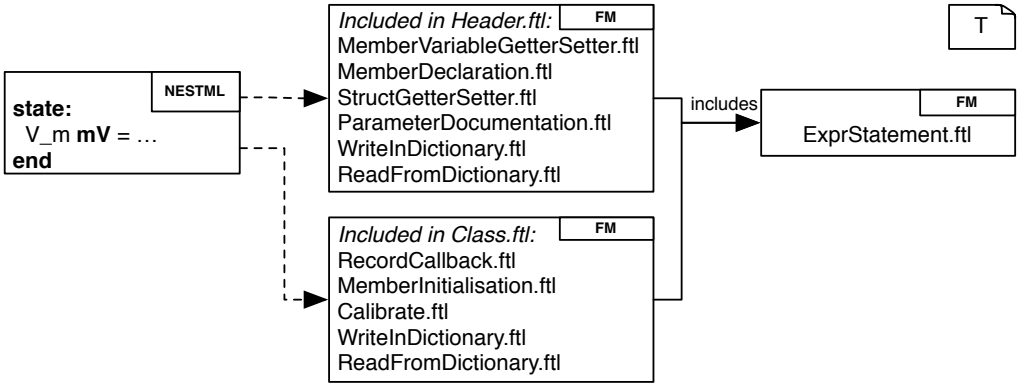


Figure 5.11: Overview over the responsible templates for variable blocks in neurons and components.

Inside the template `neuron.NeuronHeader` the class header of the neuron model is rendered. The class has the same name as the NESTML neuron model and extends the class `nest::Archiving_Node`. It resides in a namespace corresponding to the package name of NESTML neuron model. The structure of the class is based on other NEST neuron models. The template `neuron.NeuronClass` contains the implementation for every function prototype that is defined in the header and contains additional functionality to enable the archiving capabilities of the archiving node.

Every *variable block* of a NESTML neuron model has a corresponding C++ `struct` that groups the variables. The *states* variable block has a `struct` named `State_`, the *parameters* variable block has a `struct` named `Parameter_` and the *internals* variable block has a `struct` named `Variables_`. The class has a private member variable for every `struct`. This allows to easily exchange state or parameter variables, which is often necessary to set depending variables. An overview over the templates responsible for the processing of variables in variable blocks is given in Figure 5.11. These templates are included in either the `neuron.NeuronHeader` or the `neuron.NeuronClass` or both.

These `structs` contain *variable declarations* for every non-alias variable of the corresponding block. Each is rendered by the template `spl.MemberDeclaration`, which uses the same `DeclarationCalculator` as SPL to determine the correct NEST type of that variable. Additionally, a `struct` gets *getter*- and *setter*-functions for all its variables. The template `function.StructGetterSetter` uses the `VariableCalculator` to render the *getter*- and *setter*-functions as *inline* functions inside the `struct` definition.

Besides the *getter*- and *setter*-functions in the `struct`, additional *getter*- and *setter*-functions for any variable in a `struct` are generated on the class level through the template `function.MemberVariableGetterSetter`. The same template renders *alias* variables as *getter*-functions. This allows uniform access to *state*, *parameter* and *internal* variables in the translations of SPL expressions. Listing 5.11 shows generated code for the exemplary *state* variable block in Listing 5.10.

```
1 state:
2         foo units.electric.mV = 13.0      // declare state variable foo
3   alias bar units.electric.mV = foo * 2   // declare alias bar
4 end
```

Listing 5.10: Exemplary *state* variable block.

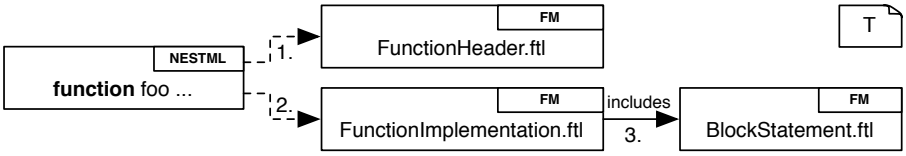


Figure 5.12: Overview over the responsible templates for function definitions in neurons and components.

```
1  struct State_ { // the State_ struct inside the neuron class
2    units::electric::mV_t foo_; // generate non-alias variables
3
4    State_();                          // constructor
5    void get(DictionaryDatum&) const;  // get- and set-functions for SLI compatibility
6    void set(const DictionaryDatum&);  // implementation in the class file
7
8    // generated getter and setter for non-alias variables, e.g. foo
9    inline units::electric::mV_t get_foo() const { return foo_ ; }
10   inline void set_foo(const units::electric::mV_t v) { foo_ = v ; }
11 };
12
13 // inside class definition:
14 State_      S_;  // class has private variable S_ for the struct
15
16 // generate getter and setter for non-alias variables
17 inline units::electric::mV_t get_foo() const { return S_.get_foo() ; }
18 inline void set_foo(const units::electric::mV_t v) { S_.set_foo( v ) ; }
19
20 // and generate getters for alias variables
21 inline units::electric::mV_t get_bar() const { return get_foo() * 2.0; }
22
23 // inside implementation file:
24 <neuron-name>::State_::State_(): foo_( 13.0 ) // generate constructor with
25 {}                                             // initialization of member variables
```

Listing 5.11: Generated code for the variable block in Listing 5.10.

Initialization of variables in these `structs` happens at different places. While non-alias state and parameter variables are initialized in their constructor, which is generated by the template `spl.MemberInitialisation`, the internal variables might rely on properties of the simulation kernel and are thus initialized inside the `calibrate`-function, which is called before the simulation is performed. This code is rendered by the template `function.Calibrate`.

The *functions* defined in a NESTML neuron model are transformed into equivalent, public member functions of the NEST neuron class. In the `neuron.NeuronHeader` template the prototypes of the function are declared with the template `function.FunctionHeader` (1. arrow in Figure 5.12). At the end of the `neuron.NeuronClass` template the implementations of the functions are rendered by the template `function.FunctionImplementation` (2. arrow). Both templates use the `FunctionCalculator` to determine the correct return type. Additionally, the calculator transforms the NESTML parameter list into a string, which contains the parameter list for the C++ function (see line 11 in Listing 5.12). The body of a function is rendered with the template `spl.BlockStatement` (3. arrow). Listing 5.12 shows the template for the function implementation.

```
1  <#--Function implements BodyElement =
2          "function" Name "(" Parameters? ")" (returnType:DottedName)?
3                  BLOCK_OPEN!
4                  Block                  // the function production as a comment
5                  BLOCK_CLOSE!;-->
6
7  // call the calculator that provides the RETURN_TYPE and the PARAMETER_STRING
8  <#if op.callCalculator("nestml.codegen.FunctionCalculator")>
9  ${RETURN_TYPE?replace(".", "::")}          // C++ types have :: name-separator
10 ${nspPrefix}::${ast.getName()}             // the function name needs namespace prefix
11   (${PARAMETER_STRING?replace(".", "::")}) // use provided PARAMETER_STRING
12 {${op.setValue("indent","  ")}             // make correct indentations
13                                           // render body
14   ${op.includeTemplates("nestml.nest.spl.BlockStatement", ast.getBlock())}
15 }
16 </#if>
```

Listing 5.12: The FreeMarker template for function implementations

By definition, a NESTML neuron model can have at most one type of output (context condition N13). During the setup of the connectivity in NEST, each source node checks if the target node accepts a connection with this output type. This check is carried out by the function `check_connection`, which therefore has to be implemented by each NEST neuron that generates output. The function is rendered in the template `neuron.NeuronHeader` as an inline function.

The *emitting of a spike* in NESTML is performed by calling the function `emitSpike` from the predefined component `nestml.Spiking` in the `dynamics` body. To emit a spike event in the NEST simulator involves creating a `SpikeEvent` object and sending it via the network class to all target nodes. To record the spike history the `nest::Archiving_Node` provides the function `set_spiketime`, which should be called for every spike emited. The calculator `FunctionCallCalculator` creates the necessary code whenever the function `emitSpike` is called. Other output types are not implemented yet.

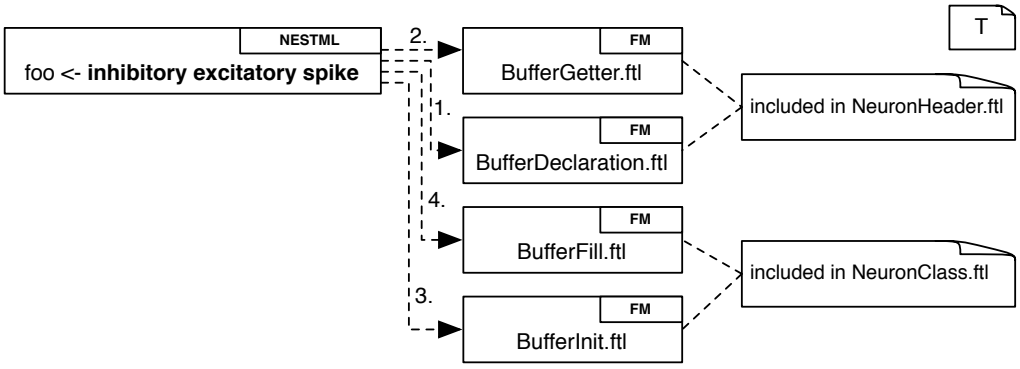


Figure 5.13: Overview over the responsible templates for inputs in neurons.

A NESTML neuron model can have several *input* declarations. A NEST neuron class gets a `RingBuffer` for every input declaration (1. arrow in Figure 5.13) that collects received events of the specified input type. These `RingBuffers` are grouped in the `struct Buffer_` with corresponding *getter*-functions (2. arrow), similar to state, parameter and internal variables. All buffers are initialized in the function `init_buffers_`, for which the code is rendered by the template `buffer.BufferInit` (3. arrow).

```
1  <#if op.callCalculator("nestml.codegen.TimestepDynamicsCalculator")>
2  // time step dynamics has a single parameter of type units.time.ms
3  // the TimestepDynamicsCalculator provides the parameter type and name
4  ${PARAMETER_TYPE?replace(".","::")} ${PARAMETER_NAME}; // specify the dynamics parameter
5  for ( nest::long_t lag = from ; lag < to ; ++lag )     // execute dynamics for every
6  {                                                      // time step in the slice
7    // set parameter to current ms time
8    ${PARAMETER_NAME} = nest::Time(nest::Time::step( lag )).get_ms() + origin.get_ms();
9
10   // render dynamics body
11   ${op.includeTemplates("nestml.nest.spl.BlockStatement", ast.getBlock())}
12 } /* for end */
13 </#if>
```

Listing 5.13: Template for the time step dynamics. Some NEST specific code is omitted.

Incoming events are processed in a `handle` function for the appropriate type, which is created in the template `neuron.NeuronClass`. The code to add an event to the correct buffer is generated by the template `buffer.BufferFill` (4. arrow). As the function `check_connection` suggests, a NEST neuron class has a function that assures that it can handle certain events. This function is called `connect_sender` and is generated for every input type at the end of the template `neuron.NeuronHeader`.

The *dynamics* of a NESTML neuron model is implemented in the `update`-function of a NEST neuron class. The purpose of the `update`-function is to advance the dynamic state of a neuron by a certain time in units of the simulation resolution. The *time step dynamics* of a neuron is supposed to advance the dynamic state of a neuron by one time step of the simulation resolution, hence the code of the `dynamics`-function might have to be executed multiple times in the `update`-function. Figure 5.14 shows the templates involved in generating the neuron dynamics and Listing 5.13 shows the template `function.TimestepDynamics`, which renders the body of the `update`-function for the time step dynamics. The dynamics body is executed for every time step inside the *for*-loop. Other dynamics types are not implemented yet.

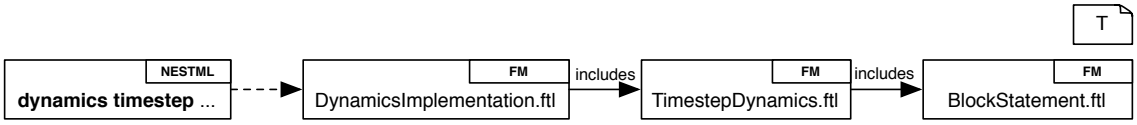


Figure 5.14: Responsible templates for generating the *dynamics* in neurons.

NESTML models can use *components* in two ways. Either they use an individual instance of a component or they use a global instance. For the NEST simulator components are generated as plain C++ classes by the templates `component.ComponentHeader` and `component.ComponentClass` (6. and 7. arrow in Figure 5.10). They have the same structure for functions and state, parameter and internal variables as neurons. Additionally, component classes have a singleton global object that can be accessed via the static function `get_instance` to support the semantics of a global instance of the component. If a neuron uses an individual instance of a component, a member variable and appropriate *getter*-functions are generated in the corresponding class.

In C++ the header of all used types has to be *included* at the top of the source file. Since NESTML allows using fully-qualified names in its files without importing that name, it is not sufficient to only include the headers of types mentioned in the *import*-statements.

The `IncludeCalculator` uses the `IncludeVisitor` to traverse the AST and collect the fully-qualified names of all mentioned types. These names are then used to generate all *includes* for the C++ headers.

The NEST simulator has the feature to *record* arbitrary numeric values of neurons during simulations [PE11]. This is implemented using a specialized `nest::RecordablesMaps` that stores a set of key-callback-pairs for a neuron class. In every recording step the results of the callbacks are stored in the map with their corresponding key. In the implementation file a singleton `nest::RecordablesMap` object for this neuron model is created. The most interesting values of a neuron are its dynamic state, so all state *getter*-functions are registered as callbacks. This registration is rendered by the template `function.RecordCallback`.

Some functions and variables of predefined components have native counterparts in C++ or NEST and the calculators `VarNameCalculator` and `FunctionCallCalculator` have the responsibility, to return the native handle for those functions and variables. For example the component `nestml.Math` has the variable `PI`, but in C++ there is no *math*-class and the value for π is obtained through the constant `M_PI` form the header `cmath`.

The user interface for the NEST simulator is implemented via the *simulation language interpreter* (SLI). SLI is able to gather information and values from neuron models and can change parameters of neurons. This exchange is performed with dictionaries that contain key-value-pairs for the requested information or changes. This design keeps the class interface small and avoids fat interfaces [Str00]. The dictionaries are processed in `get_status`- and `set_status`-functions of a NEST class, which forward the calls to the `get`- and `set`-functions of the state and parameter `structs`. The templates `functions.WriteInDictionary` and `functions.ReadFromDictionary` generate the code to fill and read-out the dictionaries.

The NEST simulator uses a documentation comment at the start of each neuron model class to generate a *html* based documentation for all models during its compilation process. For now, the generation process documents all states and parameters inside this documentation with the template `comments.ParameterDocumentation` (see lines 46 – 87 in Listing D.1). This could be extended to more extensive documentation and other output formats in the future.

Chapter 6

Application Scenario

This chapter first explains the usage of the *NESTMLTool* and how the generated code has to be processed to create a module for the NEST simulator (Section 6.1). Afterwards the *leaky integrate-and-fire* neuron model is derived and the implementation in NESTML is explained (Section 6.2).

6.1 Tool Usage

While the previous sections explained NESTML's design and implementation, this section describes the usage of the tool for developers of neuron models. The file ending *nestml* identifies models defined in the NESTML language. There are no other naming conventions for *nestml*-files and no special folder structure is required.

```
1  NESTML Help Message
2  ------------------------------------------------------------------------------
3  Use like:
4  nestml.jar <Input file/dir> [<options>]
5
6  Options:
7  -help, -h, -?          This help message.
8  -out <outputdir>       Set the directory, to which all generated files are written.
9                         Standard is: output
10 -symtabdir <dir>       Set the directory, to which all symbol table files are written.
11                        Standard is: symtab
12 -mp <modulepath>       Add a directory, in which the tool looks for NESTML models.
13 -prettyprint           Generate pretty printed versions of sources into <outputdir>.
14 -module <ModuleName>   Groups the generated files into a module, if possible.
15 -generate (<Target>)+  Generate output for the specified target.
16                        Possible values are: NEST
```

Listing 6.1: The help message for the NESTML tool.

The class `NESTMLTool` implements a command-line interface for the tool to process *nestml*-files. The Listing 6.1 shows the help-message of this tool, which will be shown, whenever something is wrong with command-line arguments or when one of the help options is stated. The first argument to the tool is the file or directory that should be processed. When models in different directories are reused in one of these files, the

path to those models has to be added to the model-paths with the option *-mp*. Each path has to be added separately with an own *-mp* statement.

The options *-out* and *-symtabdir* can be used to specify directories for generated and symbol table related files, which differ from the defaults. The tool can pretty print the models that should be processed with the option *-prettyprint*. The pretty-printed files are correctly indented and have a standard structure for *nestml* files: first all imports, then all units, components and neurons. Inside a neuron or component first the state, parameter and internal variables, then inputs, outputs and use statements and finally, all functions and the dynamics. The pretty-printed files are written to the *out* directory.

Code for a simulator is generated with the option *-generate*. Different targets can be specified one after another. Currently, the only possible target is *NEST* for code generation for the NEST simulator. All code will be generated inside the *out* directory. If the option *-module* is specified the NEST code is generated inside a folder with the given name under the *out* directory. The additional infrastructure (see Section 5.3.1) for NEST modules will also be generated in that directory. Further processing of the generated module is described in Section 5.3.1.

If no module infrastructure is generated, the neuron model and all auxiliary files have to be copied into the *models*-folder inside of NEST's source directory manually. Then the generated files have to be registered in the file *Makefile.am* of the *models*-folder, so that they are compiled. Next, the new models have to be registered within the file *modelsmodule.cpp* to be available to the simulation kernel. Finally, the NEST simulator has to be re-compiled.

6.2 Leaky Integrate and Fire Neuron Model

The *leaky integrate-and-fire neuron* is an example of a spiking point neuron model, which is often used in modeling studies because of its simplicity and availability of efficient implementations. It is a purely phenomenological model, which models the sub-threshold dynamics of the membrane potential by only a few differential equations that describe the charging of an RC circuit. Upon receiving either spiking or direct electric current input, the membrane charges and a spike event is fired when a certain threshold is crossed. After a spike, the neuron is set inactive (refractory) for a certain time.

Figure 6.1 shows the biophysical basis for this model (A) and the equivalent circuit (B). The neuron membrane consists of a bi-lipid layer, which is impermeable for ions and larger molecules. The built-in ion channels control the flow of ions depending on the membrane potential, while the ion pumps actively keep the membrane potential at its resting level.The membrane separates the inside from the outside and is modeled as a *capacitor*. The channels are modeled as resistors. If an electrical input current I is injected into the cell, it adds further charge on the capacitor, or leaks through the channels in the cell membrane. If the membrane potential exceeds a certain *threshold*, a spike is emitted and the membrane potential is reset to its *resting potential*. During the *refractory* time after emitting a spike the membrane potential is fixed to this *resting potential* and only afterwards starts to integrate again.

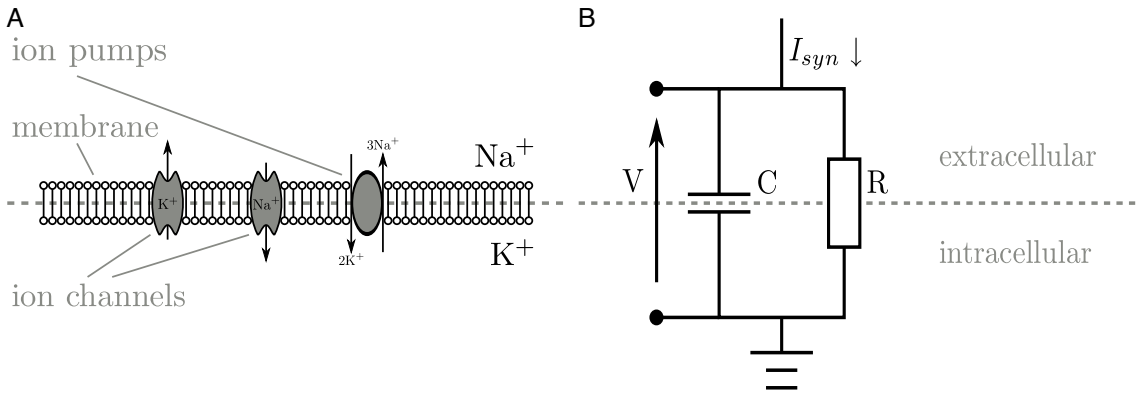


Figure 6.1: The biophysical basis of the leaky integrate-and-fire neuron. (A) A cross section of the cell membrane, with ion channels and pumps. (B) The corresponding electrical circuit. Taken from Eppler [Epp10].

The membrane potential V in the electrical circuit can be described with the differential equation in Equation 6.1 by applying electric current preservation laws [Epp10; Han+10]. All potentials in this model are relative to the *resting potential*, which is set to $0\ mV$. This is an example of an attribute, which makes use of the *alias* mechanism described in Section 4.3: while V is described relative to 0 internally, the user can define an "offset" corresponding to the biophysical reality of the model.

$$C\frac{\delta V}{\delta t} = -\frac{V}{R} + I \quad \Leftrightarrow \quad \frac{\delta V}{\delta t} = -\frac{V}{RC} + \frac{I}{C} \quad \Rightarrow \quad \frac{\delta V}{\delta t} = -\frac{V}{\tau_m} + \frac{I_{syn} + I_{ex}}{C_m} \tag{6.1}$$

$$\begin{aligned} &\frac{\delta\delta\eta}{\delta t\delta t} + 2\alpha\frac{\delta\eta}{\delta t} + \alpha^2\eta = 0,\ \eta(0) = 0,\ \frac{\delta\eta}{\delta t}(0) = \dot{\eta}_0 \\ &\qquad \Rightarrow \quad \eta(t) = \dot{\eta}_0\ t\ e^{-\alpha\ t} \qquad \text{(explicit form)} \\ &\qquad \Rightarrow \quad \eta(t) = \frac{e}{\tau_{syn}}\ t\ e^{-\frac{t}{\tau_{syn}}} \qquad \text{(parameters set)} \end{aligned} \tag{6.2}$$

Resting potential	$V_0 = V_{reset} = 0\ mV$
Membrane time constant	$\tau_m = 10\ ms$
Membrane capacitance	$C_m = 250\ pF$
Spike threshold	$\Theta = 20\ mV$
Refractory period	$t_{ref} = 2\ ms$
External current	$I_{ex} = 0\ mV$
Synaptic time constant	$\tau_{syn} = 2\ ms$
Constant external current	$I_{ex} = 0\ mV$

Table 6.1: Typical parameter values for the *integrate-and-fire* neuron model.

If another neuron emits an action potential, the postsynaptic neuron receives the synaptic input current I_{syn} via the synapse (Section 2.1). One way to model the postsynaptic current is to shape the electric input current with an *alpha*-function. The explicit form of an *alpha*-function η can be seen in Equation 6.2 with $\dot{\eta}_0$ and α chosen such that the *alpha*-function has its peak, when the spike arrives ($t = \tau_{syn}$) and the amplitude of

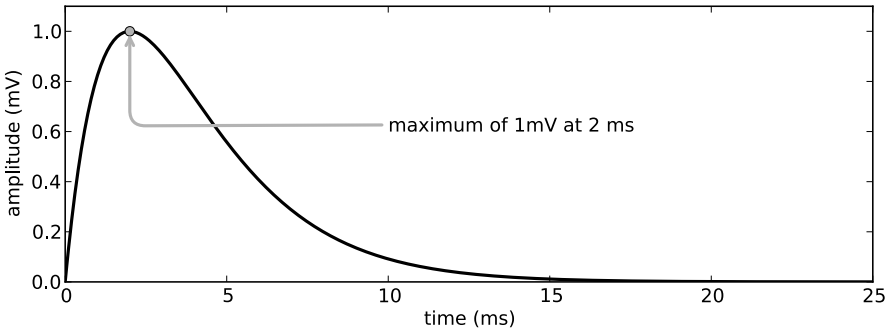


Figure 6.2: The curve of an *alpha*-function from Equation 6.2 with $\tau_{syn} = 2\ ms$.

a synapse of weight one is $1\ mV$ [RD99]. Figure 6.2 shows an example for an *alpha*-function with $\tau_{syn} = 2\ ms$. Table 6.2 contains typical values for the *leaky integrate-and-fire* neuron model.

Exact solutions for the linear differential equations Equation 6.1 and Equation 6.2 can be found through exact integration [RD99; Han+10]. Advancing a system of linear differential equations like

$$\frac{\delta y}{\delta t} = Ay + x$$

on a fixed time grid is given by the iteration formula [RD99]

$$y_{k+1} = e^{A\Delta} y_k + x_{k+1}.$$

The entries of the propagator matrix $e^{A\Delta}$ are taken from the NEST implementation of the *leaky integrate-and-fire* neuron and x_{k+1} consists of the time dependent electric current input from spikes, constant external electric current and other electric current stimuli.

These equations are sufficient to describe the charging and leaking of the membrane, but the concepts *threshold crossing*, *refractory period* and *spike emitting* have to be implemented explicitly. Listing C.1 contains a full NESTML implementation of the *leaky integrate-and-fire* neuron model, along with an in detail description of the used variables and the dynamics structure. The propagator matrix is modeled as *internal* variables. Each of the parameter values in Table 6.2 has a corresponding *parameter* variable. Inside the *dynamics* function the neuron only integrations its membrane potential, if it is not refractory. If the membrane potential crosses the threshold a spike is emitted and the neuron is set refractory.

In a line-of-code (LOC) comparison, the NESTML version has 69 LOC without comments and empty lines. The NEST version without comments and empty lines has about ∼310 LOC (∼180 LOC in source file and ∼130 LOC in the header file), which is more than four times the NEST version. At this point the feature to encapsulate often used functionality into reusable components has not been utilized, which would reduce the model code further.

This model description can be used to generate NEST code (see Section 6.1). The code for the module infrastructure is always very similar and corresponds to the module example in the NEST code base – see Section 5.3.1 for the generated module code. The model uses some of the predefined units, hence the generated code for them is

included for this module – see Section 5.3.2 for the generated unit code. The complete and tidied up header- and implementation-file that are generated for the neuron model can be seen Listing D.1 and Listing D.14, respectively.

Chapter 7

Related Work

This chapter describes modeling languages and processing tools in computational neuroscience field related to NESTML. Section 7.1 presents the *NineML* modeling language and Section 7.2 presents the *NeuroML* modeling language. Both are similar to NESTML in their modeling scope and processing capabilities and are analysed in detail. Section 7.3 presents PyNN, which is a common interface for modeling neuronal networks and controlling various simulators. Finally, Section 7.4 shortly presents other interesting modeling languages and tools.

7.1 NineML

The modeling language *NineML* (Network Interchange for Neuroscience Modeling Language) provides an unambiguous description of spiking neuronal networks for model sharing and reusability. The specification defines a common object model that describes the different elements of a model in a neuronal network (this corresponds to its abstract syntax, see Section 2.3) and uses XML [Bra+08] as its concrete syntax (see Section 2.3). Models in NineML should be simulator agnostic, i.e. they should only provide the common information necessary for all simulators to instantiate network models. For example, they only provide the neuron membrane equation, but do not state how to solve it. The following description of NineML is roughly based on Gorchetchnikov et al. [Gor+11].

Different implementations in Java, Python and Chicken Scheme can import and create the XML descriptions and can generate code to simulate models in different simulators. Currently, it is possible to fully or partly describe:

- spiking neuron models
- post-synaptic membrane electric current mechanisms
- short-term synaptic dynamics
- long-term synaptic modifications

NineML consists of two semantic layers. The *abstract* layer describes the core concepts of a model, along with its mathematical description, parameter and state variables and state update rules. In the *user* layer, on the other hand, the state and parameter variables can be described and default values can be defined. Additionally, unit definitions can be defined in the user layer.

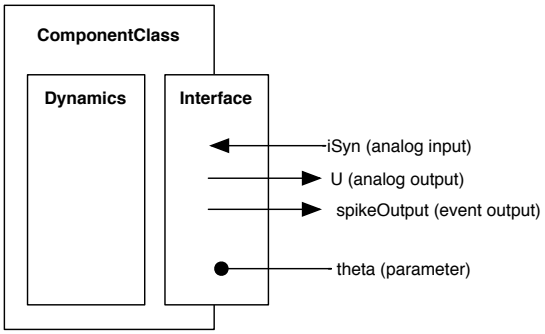


Figure 7.1: Overview over the `ComponentClass` as defined in Listing 7.1. It consists of a `Dynamics` block (an example can be seen in Figure 7.2) and an `Interface` block, which exposes various ports and parameters to other components. The arrows indicate incoming and outgoing analog and event ports. The line with the dot indicates a parameter. Redrawn from Gorchetchnikov et al. [Gor+11].

In the *abstract* layer a `ComponentClass` element represented each network model, e.g. a neuron or a synapse model. The `ComponentClass` is composed of a `Dynamics` block and a set of `Interfaces`. The *Dynamics* block contains the internal dynamics of the model, e.g. state variables and update rules, and the *Interface*s contain the parameters that can be set from the user layer and different ports that allow the `ComponentClass` to communicate with other network elements. Figure 7.1 contains an overview of the `ComponentClass` as defined in Listing 7.1.

```
1  <ComponentClass name="MyNeuronModel">
2      <Parameter  name="theta"                                  dimension="voltage"/>
3      <AnalogPort name="iSyn" mode="reduce" reduce_op="+" dimension="current"/>
4      <AnalogPort name="U"    mode="send"                       dimension="none"/>
5      <EventPort  name="spikeOutput" mode="send"/>
6
7      <Dynamics>
8          <!-- ... -->
9      </Dynamics>
10 </ComponentClass>
```

Listing 7.1: Definition of *MyNeuronModel* in NineML (excerpt) (content of `Dynamics` block are shown in Listing 7.2). Adopted from Gorchetchnikov et al. [Gor+11].

The *Interface*s define what input and output the component exposes to other components (`AnalogPort` and `EventPort`) and what parameters can be set from the *user* layer (`Parameter`). A `Parameter` has a *name* and a *dimension*, line 2 in Listing 7.1. `AnalogPorts` transmit or receive continuous values, e.g. `iSyn` in line 3 receives electric currents. They have a *name*, a *dimension* and a *mode*, which indicates whether it is a sending or receiving port. A *reduce* port, like in line 3, can receive data from multiple sending ports and combines the incoming data with a specified reduce operations. `EventPorts`

transmit or receive discrete events, e.g. they emit or receive action potentials or electric currents from other neuron models or external sources.

Inside the `Dynamics` block the internal state is defined with `StateVariable`s and `Alias`es. The state update rules are modeled with a set of `Regime`s with transitions between them, similar to a *finite-state machine* [Aho+06]. `StateVariable`s, like parameters) have a *name* and a *dimension* (line 2 and 3 in Listing 7.2. `Alias`es represent an alternative handle for a mathematical expression (line 4 – 5) and have a *name* a *dimension* and a mathematical expression (`MathInline`).

```
1  <Dynamics>
2      <StateVariable name="V" dimension="voltage"/>
3      <StateVariable name="U" dimension="voltage per time"/>
4      <Alias name="rv" dimension="none">
5          <MathInline>V*U</MathInline>
6      </Alias>
7      <Regime name="subthresholdRegime">
8          <TimeDerivative variable="U">
9              <MathInline>a*(b*V − U)</MathInline>
10         </TimeDerivative>
11         <TimeDerivative variable="V">
12             <MathInline>0.04*V*V + 5*V + 140.0 − U + iSyn</MathInline>
13         </TimeDerivative>
14         <OnCondition>
15             <Trigger>
16                 <MathInline>V>theta</MathInline>
17             </Trigger>
18             <StateAssignment variable="V" >
19                 <MathInline>c</MathInline>
20             </StateAssignment>
21             <StateAssignment variable="U" >
22                 <MathInline>U+d</MathInline>
23             </StateAssignment>
24             <EventOut port="spikeOutput" />
25         </OnCondition>
26     </Regime>
27 </Dynamics>
```

Listing 7.2: The `Dynamics` of *MyNeuronModel* from Listing 7.1. Adopted from Gorchetchnikov et al. [Gor+11].

A `Dynamics` can have several named `Regime`s, each containing differential equations (`TimeDerivative`, lines 8 – 13) for all `StateVariable`. This means, if the neuron is in a certain `Regime`, its `StateVariable`s advance according to the `TimeDerivate`s of that `Regime` – if no `TimeDerivate` is given for a certain `StateVariable`, it is assumed to be zero. The *transitions* between `Regime`s are defined either by conditions (`OnCondition`, lines 14 – 25) or by events (`OnEvent`) in the source `Regime`. The target `Regime` can be defined with the attribute `target_regime` – if no target is stated, it stays in its `Regime`. The `OnCondition` transition has to define a `Trigger` (lines 15 – 17) and the `OnEvent` transition has to define the responsible input event port. Both can modify `StateVariable` through `StateAssignment`s (lines 18 – 23) and emit events via a specified event port through the `EventOut` statement (line 25).

On the *user* layer each parameter and sending analog port of a component can be fully described and initialized. The abstract layer of the component is defined in the block `definition` (lines 2 – 3 in Listing 7.3). Every parameter and sending analog port

is described via a `property` block with the corresponding *name* (lines 5 – 10). The `quantity` defines the initial value and corresponding unit (lines 6 – 8) and the `note` block allows a short description of the property (line 9).

```
1  <component name="My_neuron_model">
2      <definition language="NineML">
3          http://www.NineML.org/neurons/MyNeuronModel.9ml <!-- abstract layer -->
4      </definition>
5      <property name="theta">
6          <quantity>
7              <value> <scalar>50</scalar> <unit>mV</unit> </value>
8          </quantity>
9          <note><String>Parameter value (spike amplitude)</String></note>
10     </property>
11     ...
12 </component>
```

Listing 7.3: *User* layer definition of *MyNeuronModel* from Listing 7.2. Adopted from Gorchetchnikov et al. [Gor+11].

NineML is more general than NESTML. This has the advantage that a *ComponentClass* is not limited to only model neurons – any kind of network element could possibly be modeled. The drawback is that it is not possible to see, which kind of model is described by the `ComponentClass` and the relation to domain concepts is less visible. NESTML makes it more obvious, whether it is a neuron model or an auxiliary component, uses concrete domain concepts and well known syntax for a procedural description of neuron dynamics. Extensions for NESTML are planned so that it can describe any kind of network model in the future.

The *parameter*, *ports* and *state* variables in NineML have corresponding counterparts in NESTML – the *state* and *parameter* variable blocks and the *input* and *output* statements. However, their definition, unit and default value is split onto the *abstract* and *user* layer, which makes it more difficult to understand the complete model.

To keep NineML descriptions independent of the simulator, it only supports the specification with differential equations to advance the dynamic state of a model. It then decides internally, how the equations are solved and does not allow the specification of an appropriate solver or an algorithm to solve the equation. For models with exact solutions, e.g. the *integrate-and-fire* neuron model from Section 6.2, this does often not yield the optimal code. Viewing the neuron dynamics as a finite-state automaton with regimes and transitions easily allows to visualize the dynamics (e.g. Figure 7.2), but for developing new neuron models a procedural definition of the dynamic is often clearer and allows a more efficient specification of the dynamics, which was an important design decision for NESTML.

Finally, the decision to use XML as concrete syntax is controversial. XML is intended to be easy to create and process by programs [Bra+08] and many processing libraries for XML exist, so no additional lexer and parsers have to be developed. The processing of `MathInline` statements is more difficult, because they do not conform with the XML standard. They have to be processed separately to ensure semantic correctness of the model, e.g. context conditions similar to S02. Further, the verbosity of XML makes writing and reading NineML models rather difficult [Che01].

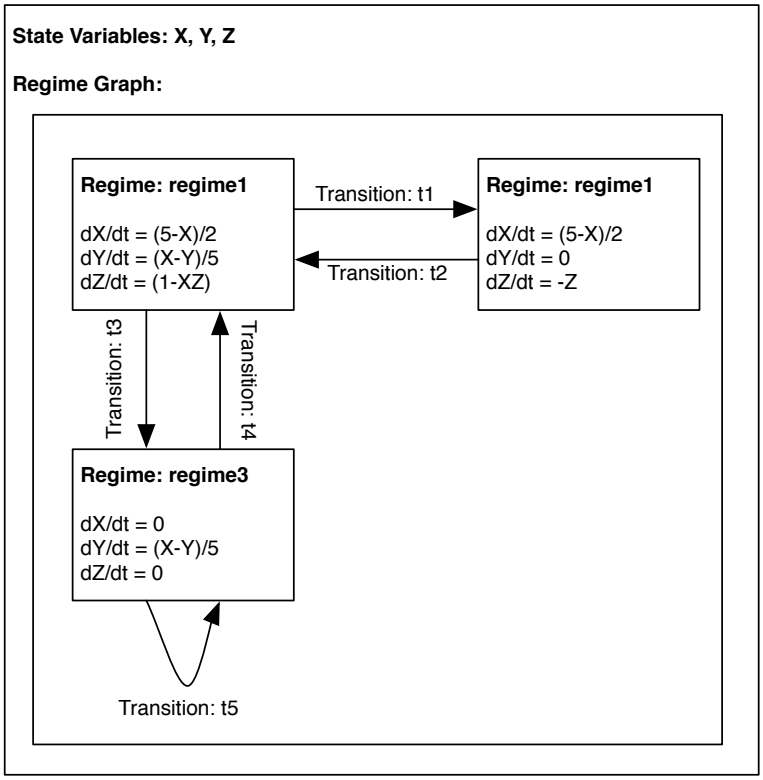


Figure 7.2: Example dynamics of NineML with three state variables, three `Regimes` and directed transitions between the `Regimes`. Redrawn from Gorchetchnikov et al. [Gor+11].

7.2 NeuroML

NeuroML (Neural Open Markup Language) is an open source project based on XML [Bra+08] and intends to be a common description language for biophysically detailed neuronal and network models and should enable interoperability across multiple simulation environments. Neuronal models reconstructed from real neurons can have complex morphologies, descriptions of voltage- and ligand-gated conductances, synaptic mechanisms and the network models contain the positions of cells and synaptic connections in a 3D network structure. Although NeuroML supports the basic integrate-and-fire neuron model, more advanced types of reduced models, such as the exponential integrate and fire or Izhikevich spiking neurons [Izh06], are not yet supported. The description of NeuroML in this section is roughly based on Gleeson et al. [Gle+10].

The language itself is separated into three levels responsible for describing different scales of biological detail, as shown in Figure 7.3. Level 1 describes the *morphology* of a neuronal model with the sub-language *MorphML*, i.e. the number and 3D position of compartments, their size and shape. It also provides mechanisms to state relevant *metadata*. Level 2 uses the *ChannelML* to describe voltage-gated *membrane conductances* together with static and plastic *synaptic conductance processes* and extends Level 1 descriptions by specifying the location and density of these membrane conductances in the cell model. Level 3 describes neuronal networks with 3D locations of

individual neurons (in *populations*), synaptic connections between neurons (in *projections*) and external electrical inputs via the *NetworkML*.

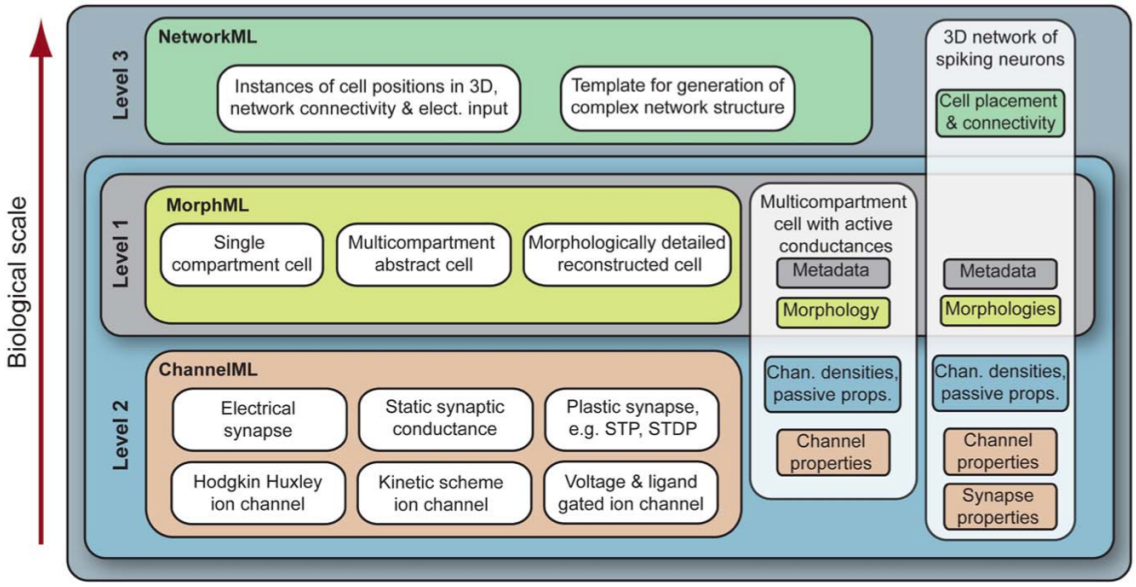


Figure 7.3: Overview over the levels of NeuroML. Taken from Gleeson et al. [Gle+10].

```
<neuroml> <!-- missing: import schemes -->
  <cells>
    <cell name = "HH_Cell">
      <meta:notes>A Simple cell with HH channels</meta:notes>
      <mml:segments>
        <mml:segment id = "0" name = "Soma" cable = "0">
          <mml:proximal x = "0.0" y = "0.0" z = "0.0" diameter = "16.0"/>
          <mml:distal x = "0.0" y = "10.0" z = "0.0" diameter = "16.0"/>
        </mml:segment>
      </mml:segments>

      <mml:cables>
        <mml:cable id = "0" name = "Soma">
          <meta:group>all</meta:group>
          <meta:group>soma_group</meta:group>
        </mml:cable>
      </mml:cables>

      <!-- Adding the biophysical parameters from Level 2 -->
      <biophysics units ="Physiological Units">
        <bio:mechanism name = "NaConductance" type = "Channel Mechanism">
          <bio:parameter name = "gmax" value = "120.0">
            <bio:group>all</bio:group>
          </bio:parameter>
        </bio:mechanism>
        <!-- missing: other mechanics -->
      </biophysics>
    </cell>
  </cells>
</neuroml>
```

Listing 7.4: Excerpt of a cell with *Hodgkin-Huxley* channels defined in MorphML. Adopted from Gleeson et al. [Gle+10].

The *MorphML* XML scheme in Level 1 allows defining the morphology of a cell. Therefore, the definition of a cell (lines 3 – 29 in Listing 7.4) requires a *name* and a list of its *segments*, which can be combined to form *cables*. Each *segment* can further be described by an identifying name and its *proximal* and *distal* dimension and diameter (lines 7 and 8). One or more segment elements can create a cable and define the 3D path of the cable. Associated to Level 1 of NeuroML is the *meta* XML scheme to supply relevant background data, e.g. notes, the original author of the model or a reference to the publication. Furthermore, the cell definition contains the *biophysics* element (lines 20 – 27) from Level 2, which specifies the used membrane channel mechanics and in which cable they are located (line 23).

```
1  <channelml> <!-- missing: import schemes -->
2    <channel_type name="NaConductance">
3      <status value="stable"/>
4
5      <current_voltage_relation cond_law="ohmic" ion="na" default_erev="50"
6                                default_gmax="120">
7        <gate name="m" instances="3">
8          <closed_state id="m0"/>
9          <open_state id="m"/>
10
11         <transition name="alpha" from="m0" to="m" expr_form="exp_linear" rate="1"
12                     scale="10" midpoint="-40"/>
13         <transition name="beta" from="m" to="m0" expr_form="exponential" rate="4"
14                     scale="-18" midpoint="-65"/>
15       </gate>
16
17       <gate name="h" instances="1">
18         <closed_state id="h0"/>
19         <open_state id="h"/>
20         <transition name="alpha" from="h0" to="h" expr_form="exponential" rate="0.07"
21                     scale="-20" midpoint="-65"/>
22         <transition name="beta" from="h" to="h0" expr_form="sigmoid" rate="1"
23                     scale="-10" midpoint="-35"/>
24       </gate>
25     </current_voltage_relation>
26
27   </channel_type>
28 </channelml>
```

Listing 7.5: ChannelML file containing a single Na^+ channel description. Adopted from Gleeson et al. [Gle+10].

Besides the *biophysics* element in the cell description, Level 2 contains the *ChannelML* XML scheme that allows defining the individual conductance mechanisms. It can describe conductance mechanisms that arise at the *channels*, which are distributed over the cell membrane, and conductance mechanisms that arise at *synaptic* contacts. Listing 7.5 shows the definition of an Na^+ channel for a *Hodgkin-Huxley* cell [HH52]. Its name (line 1) corresponds to the name in the *bio:mechanism* element. The *current_voltage_relation* element (lines 5 – 25) lists the relevant *gates* of the conductance, which in turn specify the transitions from opening to closing state and vice versa. To specify a synaptic mechanism, the *synapse_type* element allows parameterizing various typical synaptic conductances, e.g. the *doub_exp_syn* elements describes a conductance change with exponentially rising and decaying time courses (cf. *alpha* shape in Section 6.2).

```
1  <networkml> <!-- missing: import schemes -->
2    <populations>
3      <population name="PopA" cell_type="HH_Cell">
4        <instances size="2">
5          <!-- List all nodes with 3D location -->
6          <instance id="0"> <location x="0"   y="0" z="0"/> </instance>
7          <instance id="1"> <location x="100" y="0" z="0"/> </instance>
8        </instances>
9      </population>
10
11     <population name="PopB" cell_type="HH_Cell">
12       <instances size="3">
13         <!-- List all nodes with 3D location -->
14         <instance id="0"> <location x="0"   y="100" z="0"/> </instance>
15         <instance id="1"> <location x="100" y="100" z="0"/> </instance>
16         <instance id="2"> <location x="200" y="100" z="0"/> </instance>
17       </instances>
18     </population>
19   </populations>
20
21   <projections units="Physiological Units">
22     <projection name="NetworkConnection" source="PopA" target="PopB">
23       <synapse_props synapse_type="DoubleExpSynapse" internal_delay="5"
24                      threshold="-20"/>
25
26       <connections><!-- All connection between cells specified -->
27         <connection id="0" pre_cell_id="0" post_cell_id="1"/>
28         <connection id="1" pre_cell_id="1" post_cell_id="0">
29           <properties weight="0.5"/>
30         </connection>
31       </connections>
32     </projection>
33   </projections>
34
35   <inputs units="SI Units">
36     <!--  adjusted value  -->
37     <input name="RandomInput">
38       <random_stim frequency="50" synaptic_mechanism="DoubleExpSynapse"/>
39       <target population="PopA">
40         <sites>
41           <site cell_id="0"/>
42           <site cell_id="1"/>
43         </sites>
44       </target>
45     </input>
46   </inputs>
47 </networkml>
```

Listing 7.6: A simple network, where instances of cell populations, connections and inputs are specified. Adopted from Gleeson et al. [Gle+10].

With the *NetworkML* in Level 3, the 3D anatomical structure and synaptic connectivity of a network of cells is defined and the external input that is used to drive the network can be specified. Listing 7.6 displays the definition of a simple network with two populations of *HH_Cells*. First, a list of named *population*s are defined (lines 11 – 19). Each cell instance of a population has the same *cell_type* and its 3D *location* has to be specified. Besides this direct definition of instances, it is also possible to define populations with predefined generation algorithms, e.g. place 300 cells randomly inside a

certain 3D region. When the populations are defined, *projections* are used to specify a connecting synapse type and the connections between cells (lines 21 – 33). Finally, the *inputs* element lists the external electrical inputs (lines 35 – 46), e.g. a random input with the *random_stim* element. Further the location, to which cells the inputs should be applied is given with the *target* element for the target population and the *sites* for the actual target cells inside that population.

The NeuroML covers a wider application range than NESTML, since whole neuronal networks can be described with the NetworkML. On this account comparing only Level 1 and Level 2 of NeuroML with NESTML is more appropriate here. NeuroML can define complex single and multi-compartment neuron model with various biophysical mechanisms by using appropriate elements for segments, channel mechanisms or synapse mechanisms. On the one hand, this results in definitions of models that are clear and compact, e.g. by outsourcing and reusing mechanism definitions. On the other hand, the limited set of possible language elements reduces the expressiveness of NeuroML, e.g. models that do not use Hodgkin-Huxley channels, like an integrate-and-fire neuron model, can only be specified, if corresponding elements exist. Defining more mechanisms requires changes to the language definition itself. NESTML in turn can specify a new mechanism via a *component* in a procedural manner and reuse it in *neuron* models.

Since NeuroML also use XML as its concrete syntax, the same arguments as for NineML in Section 7.1 apply. A benefit over the NineML XML scheme is that NeuroML is fully XML conform, e.g. it does not need dedicated processing for `MathInline` expressions.

7.3 PyNN

PyNN is a Python-based interface for specifying and simulating neuronal network models for different simulators [Dav+09]. This allows writing a simulation script once and run it without modification on all supported simulators. Since simulators for neuronal networks have either native interpreters, e.g. *HOC* for NEURON [Hin93] or *SLI* for NEST [GD07], or a Python interface, e.g. Brian [GB08], or both to drive their simulations, PyNN defines a common Python API and the simulators implement this API in terms of their interpreter. By this means, runtime-interaction between PyNN and a supported simulator is possible.

PyNN contains a library of network elements that are common to at least two supported simulators. This includes common neuron models, like single compartment integrate-and-fire neuron model or Hodgkin-Huxley models, common synapse and synaptic plasticity models. It can create individual cells or populations thereof, change their parameters and connect them using different strategies. `Projection`s connect two populations of cells using predefined `Connector`s, e.g. `FixedProbabilityConnector` or `AllToAllConnector`. The `connect` function connects two individual cells, which allows defining own connection algorithms.

When the network is set up, the user specifies the properties, which should be recorded. The function `record` of a population causes the recording of spike times and `record_v` causes the recording of membrane potentials. Finally, the call of the `run` function starts the simulation. When the simulation finishes, calls to the functions `getSpikes` and `get_v` give access to the recorded data. Listing 7.7 shows an example script for PyNN.

In contrast to NESTML it is not possible to define new neuron models with PyNN. It is rather used to define neuronal networks with predefined models and drive a simulation of a network. The scope of PyNN and NeuroML are thus orthogonal, which suggests a possible collaboration between the two in the future: PyNN could allow creation of neuronal network models composed of elements defined in NESTML. For this scenario, the NESTMLTool could be employed to generate the corresponding source code for the selected simulator. The generated code could then be compiled and dynamically loaded into the simulator. This is subject for future investigation and requires support for additional simulators to the NESTMLTool.

```
1  from pyNN.nest2 import *  # import implementation for NEST simulator
2
3  # specify common parameters for cells
4  cell_params = {'tau_m': 20.0, 'v_rest': -49.0, ... }
5  # setup simulator infrastructure
6  setup()
7  # create populations for excitatory and inhibitory,
8  # conductance based IF (integrat-and-fire) cells
9  pE = Population(4000, IF_cond_exp, cell_params, label="Excitatory")
10 pI = Population(1000, IF_cond_exp, cell_params, label="Inhibitory")
11
12 # connect cells with probability 2%, a delay of 0.1ms and different, static weights
13 FPC = FixedProbabilityConnector
14 exc_conn = FPC(0.02, weights=0.004, delays=0.1)
15 inh_conn = FPC(0.02, weights=0.051, delays=0.1)
16
17 # connect populations with above defined connectors
18 e2e = Projection(pE, pE, exc_conn, target='excitatory')
19 e2i = Projection(pE, pI, exc_conn, target='excitatory')
20 i2e = Projection(pI, pE, inh_conn, target='inhibitory')
21 i2i = Projection(pI, pI, inh_conn, target='inhibitory')
22
23 pE.record(1000)              # record spike times of 1000 random cells in pE
24 pI.record()                  # record spike times all cells in pI
25 pE.record_v([pE[0], pE[1]])  # record membrane potential of first two cells in  pE
26
27 run(1000.0)                  # run simulation for 1000ms
28 pI.getSpikes()[:5]           # get first 5 spike times
29 pE.get_v()[:5]               # get first 5 membrane potentials
30 end()                        # finish simulation, release infrastructure
```

Listing 7.7: Example PyNN script that simulates two populations of randomly connected cells. Adopted from Davison et al. [Dav+09].

7.4 Others

Besides the languages described in detail above, there are various other languages and other tools related to neuron modeling, neuronal networks and simulation. Some of them will be shortly introduced in this section.

SBML (Systems Biology Markup Language) [Huc+03] and *CellML* [Cue+03] are XML-based modeling languages. SMBL is used for modeling biochemical reaction networks, like cell signaling pathways, metabolic pathways and gene regulation. It provides tools

that support the user with the creation, import, export, simulation and other processing of SBML models. CellML supports various electrophysiological, mechanical, signal transduction, and metabolic pathway models and has own tools for simulation and creation. Both have a similar scope beyond neuron modeling, and conversion from one model description into the other is possible [Sch+06]. SBML and CellML provide a wider scope than NESTML and are focused on detailed descriptions many biochemical mechanisms. NESTML's specialisation on nerve cells allows more detailed and diverse neuron models.

The Chicken Scheme-based language *nemo* [Rai13] reads in ion channel descriptions and can generate corresponding simulation code for different simulation environments. Currently there is only a prototype implementation and no publication about nemo available. NESTML and nemo follow a similar approach of modeling individual neuron models. While NESTML is rather general and provides a new syntax, nemo is focused on ion channel descriptions and utilize Scheme's extendability to create domain specific languages [Gra93].

Topographica [Bed09] is a Python-based simulator with extensive presentation, analysis and plotting tools. It is able to bridge between multiple simulators to analyze large-scale, detailed models of topographic maps. Therefore, Topographica partitions the cortical surface into topographic, two-dimensional maps, where populations of similar neurons are organized in *Sheets*. Multiple Sheets can be used for each neural area and can be stacked, e.g. to describe the three-dimensional structure of the cortex. A Sheet has to accept and generate Events, has a fixed area and density of neurons and has to be able to generate activity pattern arrays. Once those details for a new Sheet are available, e.g. by simulation with an external simulator, nearly all of Topographica's analysis and plotting code can be used with the new Sheet type. The definition of new Sheets is possible in Topographica itself by wrapping external simulators similar to PyNN (Section 7.3). Topographica and NESTML are related, since both model parts of the brain, but while Topographica models larger areas of the cortex, NESTML is designed to model individual neurons.

The simulators *Brian* [GB08] and *NEURON* [HC97; Hin93], as described in Section 2.2, both have a dedicated modeling language to describe and simulate neuron models. Brian describes neuron models with differentials equation in Python and NEURON uses the *HOC* modeling language. These languages are not usable for other simulators. Both provide a similar scope as NESTML, but are restricted to the corresponding simulator.

Chapter 8

Conclusion

This thesis describes the design and implementation of the domain specific language NESTML and a corresponding processing tool. NESTML allows to model point neuron models in a concise and clear manner (F01). The processing tool checks the models for programmatic correctness (NF06) and can generate code for the NEST simulator to simulate the neuron model within a biological neuronal network (NF05, F15 – F17). The language and the processing tool are developed with the design goal of being extensible (NF04) so that the scope of the language can easily be extended (NF04.1 – NF04.3) and code for more target simulators can be generated (NF05).

While no formal investigations have been carried out yet, we claim that NESTML already now reduces the workload to develop new neuron models. In comparison to C++ models for the NEST simulator, a corresponding NESTML model has only less than four times the amount of code (see Chapter 6). Further, it reduces the work to maintain models for a simulator, since only the code generation has to be adapted instead of every individual model. With the possibility to target more simulators in the future, NESTML models can be reused and findings can easily be reproduced on other simulators.

First we introduced the topic of computational neuroscience and the simulation of neuronal networks, and motivated the development of a new modeling language for neuron models. Next, we provided the fundamental background for this thesis. We described the microscopic building blocks of the brain, especially neurons, and we outlined the interaction between neurons and the translation from actual biological neuronal networks to the simulation of such networks. After that we gave a short introduction into programming and modeling languages, and described compilers in general. We explained the MontiCore framework [Grö+08] in depth, which we used to develop NESTML.

Afterwards, we listed the functional and non-functional requirements for the modeling language NESTML and its processing tool – as identified during interviews with the developers of the NEST simulator [GD07] and with domain experts, as well as extensive research of related modeling languages [Gor+11; Gle+10; GB08; Hin93; Rai13; Cue+03; Huc+03; Bed09].

In the following, we described the design of NESTML and its sub-languages. The *UnitDSL* can model physical units (F06), which in turn can be used in NESTML to model biophysical mechanisms with more detail. The *simple procedural language* (SPL) allows specifying neuron model behavior and biophysical mechanisms in a procedural way (F11)

with a clear and concise syntax (F11.3) similar to Python's. Since Python is widely used in the neuroscience community, we believe NESTML's similarity to Python improves its adoption. NESTML itself makes use of the UnitDSL and the SPL, and can model spiking point neuron models (F01) with dynamic state (F02), parameters (F04) and neuron dynamics for discrete time-step simulations (F03). Functionality can be modularized with functions (F12) and components (F13). Units, components and neuron models are organized in a hierarchical structure (F14). Each of the described languages has various *context conditions*, which ensure programmatic correct models (NF06) and support developers in writing new neuron models (NF02).

Then we covered implementation aspects of NESTML, SPL, UnitDSL and its processing tool. We included a description of their symbol table structure, the currently supported type calculations of expressions and its code generation capabilities. The tool is able to generate neuron models for the NEST simulator (F15) with additional code for auxiliary components (F16) and the complete NEST module infrastructure (F17), which simplifies the use of the generated code. The project layout, MontiCore's grammar format, the encapsulated design for context conditions and code generation makes NESTML and its tool easy to maintain and extend (NF04, NF05).

Finally, we explained the usage of the processing tool and the derivation of a complete neuron model in detail with the example of the *integrate-and-fire* neuron model. We discussed related work and similar modeling languages and tools, and concluded this thesis with an outlook on future work.

8.1 Future Work

NESTML already allows modeling a large set of spiking point neuron models and the processing tool can catch many errors during the static analysis of a NESTML model and generate the models for the NEST simulator. Many extensions to the language and the processing tool are imaginable and an embedding into the computational neuroscience software landscape is desirable.

The ability to model multi-compartment models (NF04.1) is a future extension to NESTML. Components with own membrane potential, inputs and outputs and dynamics can then be used as segments or compartments. Connected in a designated *structure*-statement they model parts of a neuron or of a neuron's dendrite or axon, enabling the use of NEST in a wider neuroscientific community.

It give developers more flexibility in creating and describing neuron models, if NESTML can model the dynamic behavior of neurons in different ways. Spikes and external electric currents can be interpreted as distinct events that arrive at a neuron, so an event-based description of the neuron dynamics could enhance the expressiveness of NESTML (NF04.2). For an example of such neurons, see Hanuschkin et al. [Han+10]. In addition, the dynamics of many neuron models can be described through a set of ordinary differential equations (ODE). It is beneficial, if these could be written down directly in NESTML to describe the neuron dynamics (NF04.3) together with hints about an appropriate method for solving them. The direct use of ODEs in a model description will require the availability of a library of solvers. To this end, NESTML can serve as an easy-to-use interface to external solvers like the ones from the GNU Scientific Library [Gal+03] or CVODE from SUNDAILS [CH96].

Besides neuron models, biological neuronal networks need models for synapses. Naturally, extending NESTML to allow modeling of synapse models is a goal for future work.

Extensions to the processing tool include support for more target simulators (NF05) and more context conditions. An interesting target platform is SpiNNakkar [Pai+13], since it is relatively new and still has only a few neuron models implemented. SpiNNakkar is a computer architecture with an array of ARM9 [Fur00] cores that communicate via packets carried by a custom interconnect fabric. On each core "lives" a set of neurons that communicate via spike-packages. The code size of neuron models is crucial for SpiNNakker, since the cores can only store a limited number of instructions. Another limiting aspect of SpiNNakkar is that the used ARM cores only support integer and fixed-point arithmetic – support for floating-point arithmetic is added via libraries.

Future context conditions, for example, need to check correct usage and reachability of *return*-statements in functions. Type-checking can be extended to test the correct use of physical units in expression: currently it is sufficient that the expression $15.5\frac{mV}{ms} * 5ms$ yields a *real* type, although the actual type would be mV (see Section 5.2).

Incorporating NESTML into PyNN [Dav+09] is another interesting topic for future work, since it can simplify the workflow of specifying and simulating neuronal networks of neurons described in NESTML. The necessary operations to process the NESTML model, generate the code for the specified simulator and compile and load this code into the simulator could be performed by PyNN. When findings should be shared, it is then sufficient, to only distribute the PyNN description.

Finally, usage and usability studies with neuron model developers need to be conducted for NESTML, to further assess the usefulness of NESTML for the day-to-day work of researchers in computational neuroscience.

Bibliography

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques & Tools*. 2nd. Pearson Education, Inc., 2006 (cit. on pp. 14, 18, 81).

[BB98] James M. Bower and David Beeman. *The book of GENESIS (2nd ed.): exploring realistic neural models with the GEneral NEural SImulation System*. Springer-Verlag New York, Inc., 1998 (cit. on p. 11).

[Bed09] James A Bednar. "Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components". In: *Frontiers in Neuroinformatics* 3.8 (2009) (cit. on pp. 89, 91).

[Ber+13] Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, Silvia Pfeiffer, and Ian Hickson. *HTML 5.1 A vocabulary and associated APIs for HTML and XHTML*. Tech. rep. http://www.w3.org/TR/2013/WD-html51-20130528/. W3C, 05/2013 (cit. on p. 12).

[Bra+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Tech. rep. http://www.w3.org/TR/2008/REC-xml-20081126/. W3C, 11/2008 (cit. on pp. 79, 82 sq.).

[Bre+07] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Andrew P. Davison, Sami El Boustani, and Alain Destexhe. "Simulation of networks of spiking neurons: A review of tools and strategies". In: *Journal of Computational Neuroscience* 2007 (2007), pp. 349–398 (cit. on p. 3).

[Cal10] John Calcote. *Autotools: A Practioner's Guide to GNU Autoconf, Automake, and Libtool*. 1st. No Starch Press, 2010 (cit. on p. 60).

[CH96] Scott D. Cohen and Alan C. Hindmarsh. "CVODE, a Stiff/Nonstiff ODE Solver in C". In: *Comput. Phys.* 10.2 (03/1996), pp. 138–143 (cit. on p. 92).

[Che01] James Cheney. "Compressing XML with Multiplexed Hierarchical PPM Models". In: *Proceedings of the Data Compression Conference*. DCC '01. IEEE Computer Society, 2001, pp. 163– (cit. on p. 82).

[Cro+12] Sharon M. Crook, James A. Bednar, Sandra Berger, Robert Cannon, Andrew P. Davison, Mikael Djurfeldt, Jochen Eppler, Birgit Kriener, Steve Furber, Bruce Graham, Hans E. Plesser, Lars Schwabe, Leslie Smith, Volker Steuber, and Sacha van Albada. "Creating, documenting and sharing network models". In: *Network: Computation in Neural Systems* 23.4 (09/2012), pp. 131–149 (cit. on p. 4).

[Cue+03] Autumn A. Cuellar, Catherine M. Lloyd, Poul F. Nielsen, David P. Bullivant, David P. Nickerson, and Peter J. Hunter. "An Overview of CellML 1.1, a Biological Model Description Language". In: *SIMULATION* 79.12 (2003), pp. 740–747 (cit. on pp. 88, 91).

[Dav+09] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. "PyNN: a common interface for neuronal network simulators". In: *Frontiers in Neuroinformatics* 2 (01/2009), p. 10 (cit. on pp. 87 sq., 93).

[DG02] Markus Diesmann and Marc-Oliver Gewaltig. "NEST: An environment for neural systems simulations". In: *In T. Plesser and V. Macho (Eds.), Forschung und wisschenschaftliches Rechnen, Beitrage zum Heinz-Billing-Preis 2001, Volume 58 of GWDG-Bericht*. 2002, pp. 43–70 (cit. on p. 9).

[DRS13] Dániel Dékány, Jonathan Revusky, and Attila Szegedi. *FreeMarker Manual*. Tech. rep. http://freemarker.org/docs/index.html. FreeMarker project, 06/2013 (cit. on pp. 16, 25).

[Epp+09] Jochen M Eppler, Moritz Helias, Eilif Muller, Markus Diesmann, and Marc-Oliver Gewaltig. "PyNEST: a convenient interface to the NEST simulator". In: *Frontiers in Neuroinformatics* 2.12 (2009) (cit. on p. 11).

[Epp10] Jochen Martin Eppler. "Architectures for communication between processes and software layers for a simulator for biological neural networks". PhD thesis. Albert-Ludwigs-Universität Freiburg Technische Fakultät, 12/2010 (cit. on pp. 7 sqq., 11, 75).

[Fic+13] Christoph Ficek, Hans Grönniger, Christoph Herrmann, Holger Krahn, Claas Pinkernell, Holger Rendel, Bernhard Rumpe, Martin Schindler, and Steven Völkel. *MontiCore 2.0.7 - Framework zur Erstellung und Verarbeitung domänenspezischer Sprachen*. Tech. rep. http://www.monticore.de/doc/MCDoku.pdf. RWTH Aachen Software Engineering, 05/2013 (cit. on pp. 15 sq.).

[FS13] Neil Fraser and Ellen Spertus. *blockly – A visual programming editor*. 2013. URL: `https://code.google.com/p/blockly/` (visited on 09/24/2013) (cit. on p. 12).

[Fur00] Steve Furber. *ARM System-on-Chip Architecture*. 2nd. Addison-Wesley Longman Publishing Co., Inc., 2000 (cit. on p. 93).

[Gal+03] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Michael Booth, and Fabrice Rossi. *Gnu Scientific Library: Reference Manual*. Network Theory Ltd., 02/2003 (cit. on p. 92).

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995 (cit. on pp. 16, 19, 22, 54).

[GB08] Dan F M Goodman and Romain Brette. “Brian: a simulator for spiking neural networks in Python”. In: *Frontiers in Neuroinformatics* 2.5 (2008) (cit. on pp. 10, 87, 89, 91).

[GD07] Marc-Oliver Gewaltig and Markus Diesmann. “NEST (NEural Simulation Tool)”. In: *Scholarpedia* 2.4 (2007), p. 1430 (cit. on pp. 5, 9, 59, 87, 91).

[Gho10] Debasish Ghosh. *DSLs in Action*. 1st. Manning Publications Co., 2010 (cit. on p. 11).

[GK02] Wulfram Gerstner and Werner Kistler. *Spiking Neuron Models: An Introduction*. Cambridge University Press, 2002 (cit. on p. 3).

[Gle+10] Padraig Gleeson, Sharon Crook, Robert C. Cannon, Michael L. Hines, Guy O. Billings, Matteo Farinella, Thomas M. Morse, Andrew P. Davison, Subhasis Ray, Upinder S. Bhalla, Simon R. Barnes, Yoana D. Dimitrova, and R. Angus Silver. “NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail”. In: *PLoS Comput Biol* 6.6 (06/2010) (cit. on pp. 83–86, 91).

[Gor+11] Anatoli Gorchetchnikov, Ivan Raikov, Mike Hull, and Yann Le Franc. *Network Interchange for Neuroscience Modeling Language (NineML) – Specification*. Tech. rep. http://software.incf.org/software/nineml/wiki/nineml-specification/. INCF Task Force on Multi-Scale Modeling, 07/2011 (cit. on pp. 79–83, 91).

[Gra93] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice Hall, 09/1993 (cit. on p. 89).

[Grö+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. “MontiCore: a framework for the development of textual domain specific languages”. In: *Companion of the 30th international conference on Software engineering*. ICSE Companion ’08. ACM, 2008, pp. 925–926 (cit. on pp. 5, 91).

[Han+10] Alexander Hanuschkin, Susanne Kunkel, Moritz Helias, Abigail Morrison, and Markus Diesmann. “A general and efficient method for incorporating precise spike times in globally time-driven simulations”. In: *Frontiers in Neuroinformatics* 4.113 (2010) (cit. on pp. 28, 75 sq., 92).

[HC97] M. L. Hines and N. T. Carnevale. “The NEURON Simulation Environment”. In: *Neural Computation* 9.6 (08/1997), pp. 1179–1209 (cit. on pp. 10, 89).

[HH52] A. L. Hodgkin and A. F. Huxley. “A quantitative description of membrane current and its application to conduction and excitation in nerve.” In: *The Journal of physiology* 117.4 (08/1952), pp. 500–544 (cit. on p. 85).

[Hin93] Michael Hines. “NEURON — A Program for Simulation of Nerve Equations”. In: *Neural Systems: Analysis and Modeling*. Springer US, 1993, pp. 127–136 (cit. on pp. 87, 89, 91).

[Huc+03] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, and the rest of the SBML Forum. “The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models”. In: *Bioinformatics* 19.4 (2003), pp. 524–531 (cit. on pp. 88, 91).

[Izh06] Eugene M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting (Computational Neuroscience)*. 1st ed. The MIT Press, 11/2006 (cit. on p. 83).

[Ker+08] Rex A. Kerr, Thomas M. Bartol, Boris Kaminsky, Markus Dittrich, Jen-Chien Jack Chang, Scott B. Baden, Terrence J. Sejnowski, and Joel R. Stiles. “Fast Monte Carlo Simulation Methods for Biological Reaction-Diffusion Systems in Solution and on Surfaces”. In: *SIAM J. Sci. Comput.* 30.6 (10/2008), pp. 3126–3149 (cit. on p. 11).

[Kra09] Holger Krahn. “MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering”. PhD thesis. RWTH Aachen University, 12/2009 (cit. on pp. 5, 11, 19 sq.).

[Mal+04] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. “Scratch: A Sneak Preview”. In: *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*. C5 ’04. IEEE Computer Society, 2004, pp. 104–109 (cit. on p. 12).

[MDG08] Abigail Morrison, Markus Diesmann, and Wulfram Gerstner. “Phenomenological models of synaptic plasticity based on spike timing.” In: *Biological Cybernetics* 98.6 (2008), pp. 459–478 (cit. on p. 3).

[Mül08] Kevin Müller. “Design and Implementation of an SQL Grammar for the MontiCore Framework”. MA thesis. Technical University Carolo-Wilhelmina in Braunschweig, 04/2008 (cit. on pp. 12, 14).

[OGS08] Bryan O’Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. 1st. O’Reilly Media, Inc., 2008 (cit. on p. 13).

[Oli06] Travis E. Oliphant. *Guide to NumPy*. Trelgol Publishing, 2006 (cit. on p. 10).

[OMG11a] OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1*. Object Management Group, 08/2011. URL: `http://www.omg.org/spec/UML/2.4.1` (cit. on p. 102).

[OMG11b] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. Object Management Group, 08/2011. URL: `http://www.omg.org/spec/UML/2.4.1` (cit. on p. 102).

[Pai+13] E. Painkras, L.A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D.R. Lester, A.D. Brown, and S.B. Furber. “SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation”. In: *Solid-State Circuits, IEEE Journal of* 48.8 (2013), pp. 1943–1953 (cit. on p. 93).

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007 (cit. on p. 16).

[PE11] Hans Plesser and Jochen Eppler. “Flexible and efficient recording of neuronal properties in large network simulations: The NEST Multimeter”. In: *BMC Neuroscience* 12.Suppl 1 (2011) (cit. on p. 72).

[PH13] Terence Parr and Sam Harwell. *ANTLR*. 2013. URL: `http://www.antlr.org/` (visited on 11/25/2013) (cit. on p. 16).

[Rai13] Ivan Raikov. *nemo – The Chicken Scheme wiki*. 2013. URL: `http://wiki.call-cc.org/eggref/4/nemo?action=show&rev=28864` (visited on 11/11/2013) (cit. on pp. 11, 89, 91).

[RB08] Subhasis Ray and Upinder S Bhalla. “PyMOOSE: interoperable scripting in Python for MOOSE”. In: *Frontiers in Neuroinformatics* 2.6 (2008) (cit. on p. 11).

[RD11] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011 (cit. on p. 34).

[RD99] Stefan Rotter and Markus Diesmann. “Exact digital simulation of time-invariant linear systems with applications to neuronal modeling”. In: *Biological Cybernetics* 81.5-6 (1999), pp. 381–402 (cit. on p. 76).

[Sch+06] Maria J. Schilstra, Lu Li, Joanne Matthews, Andrew Finney, Michael Hucka, and Nicolas Le Novère. “CellML2SBML: conversion of CellML into SBML”. In: *Bioinformatics* 22.8 (2006), pp. 1018–1020 (cit. on p. 89).

[SR91] Dan Shafer and Dean A. Ritz. *Practical Smalltalk : using Smalltalk/V*. Springer-Verlag, 1991 (cit. on p. 12).

[Ste12] Jens Stephani. “Erweiterung von MontiCore zur Verwendung von Attributen”. MA thesis. RWTH Aachen University, 04/2012 (cit. on pp. 16, 23, 25).

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. 3rd. Addison-Wesley Longman Publishing Co., Inc., 2000 (cit. on p. 72).

[Sum08] Mark Summerfield. *Programming in Python 3: A Complete Introduction to the Python Language*. 1st. Addison-Wesley Professional, 2008 (cit. on p. 13).

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, Fourth Edition*. Addison-Wesley Longman, Amsterdam, 2011 (cit. on p. 19).

[Ven13] Bill Venners. *ScalaTest*. 2013. URL: `http://www.scalatest.org/` (visited on 09/20/2013) (cit. on p. 11).

[Völ11] Steven Völkel. “Kompositionale Entwicklung domänenspezifischer Sprachen”. PhD thesis. Technical University Carolo-Wilhelmina in Brauschweig, 04/2011 (cit. on pp. 12, 21 sq., 54).

Appendix A

Abbreviations

Some of the diagrams in this thesis contain special comments that explain the formal meaning of the elements in the diagram or explain, what kind of element is used. Additionally, the text contains some abbreviations of longer and often used terms. This chapter shortly explains each of the used abbreviations.

GPL General Purpose Languages, see Section 2.3.

DSL Domain Specific Languages, see Section 2.3.

AST Abstract Syntax Tree, see Section 2.4.

ODE Ordinary Differential Equation.

This diagram is a UML class diagram according to OMG [OMG11a; OMG11b].

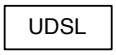
This diagram is a UML class diagram according to OMG [OMG11a; OMG11b].

This diagram shows an abstract syntax tree as described in Section 2.4. Possibly, the nodes can have attributes next them as described in Section 2.5.4.

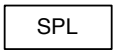
This diagram shows the calling hierarchy of FreeMarker templates with the triggering element. The elements of this diagram have additional abbreviations explaining, which kind of element they are.

This element represents a FreeMarker template.

This element represent the usage of a certain command-line argument.



This element represents a model written in the UnitDSL.



This element represents a production from the SPL.

This element represents a production from the NESTML.

Appendix B

Language Grammars

```
1  package nestml.literals;
2
3  grammar Literals {
4
5      options {
6          parser lookahead=3
7          lexer lookahead=3
8          nostring noident nows noslcomments
9      }
10
11     interface IPackage;
12     ast IPackage = packageName:DottedName;
13
14     APackage implements IPackage = "package" packageName:DottedName;
15
16     token BLOCK_OPEN = ":";
17
18     token BLOCK_CLOSE = "end";
19
20     DottedName = names:Name ("." names:Name)*;
21     ast DottedName =
22         method public String toString() {
23             return com.google.common.base.Joiner.on(".").join(getNames());
24         };
25
26     token Name
27         options{testLiterals=true;} = // check Literals first
28         ('a'..'z' | 'A'..'Z' | '_' | '$')
29         ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$')*;
30
31
32     // allow EOL to be a expression delimiter
33     token WS = (' ' | '\t' | '\f' )+ {_ttype = Token.SKIP;};
34     token EOL = ( options {generateAmbigWarnings=false;}:
35                     "\r\n" // DOS
36                   | '\r' // Macintosh
37                   | '\n' // Unix
38                 ) { newline();};
```

Listing B.1: The Literals grammar. Many parts are omitted, since they are equal to `mc.literals.Literals`. (Part 1 of 2)

```
39      token SL_COMMENT = "//" (~('\n'|'\r'))* ('\n'!|'\r'!('\n'!)?)?
40          {newline();}{
41      if (getCompiler() != null) {
42        Token _token_comment = makeToken(_ttype);
43        _token_comment.setText(new String(text.getBuffer(), _begin,
44            text.length() - _begin));
45        mc.ast.Comment _comment = new mc.ast.Comment(_token_comment.getText());
46        _comment.set_SourcePositionStart(new mc.ast.SourcePosition(
47            _token_comment.getLine(), _token_comment.getColumn()));
48        _comment.set_SourcePositionEnd(
49            mc.antlr.MC_LLkParser.computeEndPosition(_token_comment));
50        getCompiler().addComment(_comment);
51      }};
52  }
```

Listing B.2: The Literals grammar. Many parts are omitted, since they are equal to `mc.literals.Literals`. (Part 2 of 2)

```
1   package nestml.unit;
2
3   grammar Unit extends nestml.literals.Literals {
4       options {
5         nostring noident nows noslcomments
6       }
7
8       ast Unit = method public String toString() {
9               return org.apache.commons.lang.
10                  builder.ToStringBuilder.reflectionToString(this);
11          };
12
13      Package = APackage BLOCK_OPEN Units BLOCK_CLOSE;
14
15      Units = (SL_COMMENT! | EOL!)* UnitLine
16              (((SL_COMMENT! | EOL!)+ "unit") => ((SL_COMMENT! | EOL!)+ UnitLine)
17               | SL_COMMENT!
18               | EOL!)*;
19
20      UnitLine = Unit (options {greedy=true;}:";" Unit)* (";")?;
21
22      Unit = "unit" unitName:Name ([real:"Real"]|[integer:"Integer"])
23              ([leftInclusive:"["]|[leftExclusive:"("])
24                  (from:SignedNumericLiteral|[fromInf:"-inf"])
25                  "..."
26                  (to:SignedNumericLiteral|[toInf:"inf"])
27              ([rightInclusive:"]"]|[rightExclusive:")"]);
28  }
```

Listing B.3: The grammar for the UnitDSL.

```
1  package nestml.spl;
2
3  /* Small Procedual Language */
4  grammar SPL extends nestml.literals.Literals {
5      options {
6        nostring noident nows noslcomments
7      }
8
9      concept attributes {
10         syn typeEntry: /nestml.spl.ets.entries.SPLTypeEntry;
11
12         global SymbolTable: /interfaces2.helper.SymbolTableInterface;
13         global Level: /mc.ProblemReport.Type;
14      }
15
16     SPLFile = Block;
17
18     Block = (
19             ("if"|"for"|"while"|Name|DottedName) => Stmt
20             | SL_COMMENT! | EOL!
21             )*;
22
23     Stmt = Simple_Stmt (SL_COMMENT! | EOL! | EOF)
24          | Compound_Stmt;
25
26     Compound_Stmt = IF_Stmt
27                   | FOR_Stmt
28                   | WHILE_Stmt;
29
30     Simple_Stmt = Small_Stmt (options {greedy=true;}:";" Small_Stmt)* (";")?;
31
32     Small_Stmt = (DottedName "=") => Assignment
33                | (DottedName "(") => FunctionCall
34                | Declaration
35                | ReturnStmt;
36
37     Assignment = DottedName "=" Expr;
38
39     Declaration = vars:Name ("," vars:Name)* type:DottedName ( "=" Expr )?;
40
41     ReturnStmt = "return" (Expr | Test);
42
43     IF_Stmt = "if" Test BLOCK_OPEN Block ELIF_Clause* (ELSE_Clause)? BLOCK_CLOSE;
44
45     ELIF_Clause = "elif" Test BLOCK_OPEN Block;
46
47     ELSE_Clause = "else" BLOCK_OPEN Block;
48
49     FOR_Stmt = "for" var:Name "in" from:Expr "..." to:Expr
50                    ("step" step:SignedNumericLiteral)?
51                    BLOCK_OPEN Block BLOCK_CLOSE;
52
53     WHILE_Stmt = "while" Test BLOCK_OPEN Block BLOCK_CLOSE;
54
55     Test = OR_Test;
56
57     OR_Test = AND_Test ("or" AND_Test)*;
```

Listing B.4: The grammar for the SPL. (Part 1 of 2)

```
58      AND_Test = NOT_Test ("and" NOT_Test)*;
59
60      NOT_Test = "not" NOT_Test
61               | (Expr ("<"|"<="|"=="|"!="|"<>"|">="|">")) => Comparison
62               | (DottedName "(") => FunctionCall
63               | DottedName // bool variable
64               | BooleanLiteral; // true & false
65
66      Comparison = "(" Test ")"
67                 | left:Expr op:["<"|"<="|"=="|"!="|"<>"|">="|">"] right:Expr;
68
69      Expr = XOR_Expr ("|" XOR_Expr)*;
70
71      XOR_Expr = AND_Expr ("^" AND_Expr)*;
72
73      AND_Expr = SHIFT_Expr ("&" SHIFT_Expr)*;
74
75      SHIFT_Expr = ARITH_Expr (SHIFT_ExprEnd)*;
76      SHIFT_ExprEnd = sign:["<<"|">>"] ARITH_Expr;
77
78      ARITH_Expr = Term (ARITH_ExprEnd)*;
79      ARITH_ExprEnd = sign:["+"|"-"] Term;
80
81      Term = Factor (TermEnd)*;
82      TermEnd = sign:["*"|"/"|"%"] Factor;
83
84      Factor = sign:["+"|"-"|"~"] Factor
85             | Power;
86
87      Power = Atom (options {greedy=true;}:"**" Factor)?;
88
89      Atom = "(" Expr ")"
90           | (DottedName "(") => FunctionCall
91           | DottedName // variables
92           | NumericLiteral
93           | StringLiteral
94           | BooleanLiteral;
95
96      FunctionCall = DottedName "(" ArgList ")";
97
98      ArgList = (args:Expr ("," args:Expr)*)?;
99  }
```

Listing B.5: The grammar for the SPL. (Part 2 of 2)

```
1  package nestml;
2
3  grammar NESTML extends nestml.literals.Literals {
4      options {
5        nostring noident nows noslcomments
6        lexer lookahead = 5
7      }
8
9      concept attributes {
10         global SPLConnector: /nestml.spl._ast.SPLToolConcreteStorageConnector;
11
12         syn typeEntry: /nestml.ets.entries.NESTMLTypeEntry;
13      }
14
15     external Block;
16     external Declaration;
17     external UnitLine;
18
19     Package = (SL_COMMENT! | EOL!)*
20               APackage BLOCK_OPEN!
21                 ( SL_COMMENT! | EOL! | Statment)*
22               BLOCK_CLOSE!
23               (SL_COMMENT! | EOL!)*;
24
25     Statment = ("import" | "unit") => (Simple_Stmt (SL_COMMENT! | EOL! | EOF))
26              | Compound_Stmt;
27
28     Compound_Stmt = Neuron
29                   | Component;
30
31     Simple_Stmt = Import
32                 | UnitLine;
33
34     Import = "import" DottedName ([isStar:".*"])? (";")?;
35     ast Import = method public String toString() {
36         if (isStar) {
37             return "import␣" + getDottedName() + ".*";
38         }
39         return "import␣" + getDottedName();
40     };
41
42     Neuron = "neuron" Name Body;
43
44     Component = "component" Name Body;
45
46     interface BodyElement;
47
48     / Body = BLOCK_OPEN! ( SL_COMMENT! | EOL! | BodyElement)* BLOCK_CLOSE!;
49
50     USE_Stmt implements BodyElement = "use" name:DottedName "as" alias:Name;
```

Listing B.6: The grammar for NESTML. (Part 1 of 2)

```
51      Var_Block implements BodyElement =
52          ([state:"state"]|[para:"parameter"]|[internal:"internal"])
53              BLOCK_OPEN
54                  ( (("alias")? Name) =>
55                      AliasDecl (options {greedy=true;}:";" AliasDecl)* (";")?
56                    | SL_COMMENT! | EOL!)*
57              BLOCK_CLOSE;
58
59      AliasDecl = ([alias:"alias"])? Declaration;
60
61      Input implements BodyElement = "input"
62              BLOCK_OPEN!
63                  (InputLine | SL_COMMENT! | EOL!)*
64              BLOCK_CLOSE!;
65
66      InputLine = Name "<-" InputType* ([spike:"spike"]|[current:"current"]);
67
68      InputType = ([inh:"inhibitory"]|[exc:"excitatory"]);
69
70      Output implements BodyElement =
71                      "output" BLOCK_OPEN! ([spike:"spike"]|[current:"current"]) ;
72
73      Structure implements BodyElement = "structure"
74                  BLOCK_OPEN!
75                      (StructureLine | SL_COMMENT! | EOL!)*
76                  BLOCK_CLOSE!;
77
78      StructureLine = compartments:DottedName ("-" compartments:DottedName);
79
80      Function implements BodyElement =
81              "function" Name "(" Parameters? ")" (returnType:DottedName)?
82                 BLOCK_OPEN!
83                 Block
84                 BLOCK_CLOSE!;
85
86      Parameters = Parameter ("," Parameter)*;
87      ast Parameters = method public String toString() {
88          return com.google.common.base.Joiner.on(",␣").join(getParameter());
89      };
90
91      Parameter = Name type:DottedName;
92      ast Parameter = method public String toString() {
93          return name + "␣" + type.toString();
94      };
95
96      Dynamics implements BodyElement = "dynamics" (MinDelay | TimeStep)
97                                        "(" Parameters? ")"
98                                          BLOCK_OPEN! Block BLOCK_CLOSE!;
99
100     MinDelay = "minDelay";
101
102     TimeStep = "timestep";
103 }
```

Listing B.7: The grammar for NESTML. (Part 2 of 2)

Appendix C

Example NESTML neuron

For comparison, the implementation structure is similar to the *leaky integrate-and-fire* neuron in NEST[1]. The variable `y0` represents the external electric current stimulus, `y1` and `y2` are necessary for the alpha shape post-synaptic current calculations (see Section 6.2), and `y3` represents the relative membrane potential. The variables in the *parameter* block have been described previously. The variables starting with *P* in the *internals* block represent the propagator matrix.

Inside the *dynamics* function the *threshold crossing* is implemented in lines 68 – 73: if the membrane potential `y3` exceeds the threshold `Theta`, the neuron is set refractory (line 69), i.e. the counter `r` is set to the number of simulation steps corresponding to the refractory time, the membrane potential is set to its reset value `delta_V_reset` (line 70) and a spike is emitted (line 72). Lines 53 – 58 implement the membrane charging and the refractory period: if the neuron is not refractory, i.e. the counter `r` is zero, the membrane potential can be charged (line 55), otherwise the counter `r` is decreased.

[1] Compare models/iaf_neuron.h and models/iaf_neuron.cpp in the NEST simulator version 2.2.2 .

```
1  package models:
2    import units.electrics.*
3    import units.time.*
4
5    neuron iaf_neuron:
6      state:
7        y0 mV                    // external current
8        y1 mV                    // y1 and y2 are used in order to
9        y2 mV                    // calculate alpha shape
10       y3 mV                    // relative membrane potential
11       r  integer               // refractory steps to go
12       alias V_m mV = y3 + E_L // actual Membrane potential.
13     end
14
15     parameter:
16       C_m       pF = 250       // Capacity of the membrane.
17       tau_m     ms = 10        // Membrane time constant.
18       tau_syn   ms = 2         // Time constant of synaptic current.
19       t_ref     ms = 2         // Refractory period.
20       E_L       mV = -70       // actual Resting potential.
21       delta_V_reset mV = -70 - E_L  // relative Resting potential. => 0
22       Theta     mV = -55 - E_L  // relative threshold
23       I_e       pA = 0         // constant external current.
24
25       // some aliases
26       alias V_th mV = Theta + E_L  // actual Threshold.
27       alias  V_reset mV = delta_V_reset + E_L // Reset value of the membrane potential.
28     end
29
30     internal:
31       h    ms   = Time.resolution()     // get simulation resolution
32       // propagator matrix
33       P11 real = Math.E ** (-h / tau_syn)
34       P22 real = P11
35       P33 real = Math.E ** (-h / tau_m)
36       P21 real = h * P11
37       P30 real = 1 / C_m * (1 - P33) * tau_m
38       P31 real = 1 / C_m * ((P11 - P33) / (-1/tau_syn + 1/tau_m) - h * P11) /
39                                                          ↪(-1/tau_m + 1/tau_syn)
40       P32 real = 1 / C_m * (P33 - P11) / (-1/tau_m - -1/tau_syn)
41
42       PSCInitialValue  mV = 1 * Math.E / tau_syn
43       RefractoryCounts integer = Time.steps(t_ref) // convert time into steps
44     end
45
46     input:
47       spikeBuffer   <- inhibitory excitatory spike // put all spikes into one buffer
48       currentBuffer <- current          // put external current stimulus into  buffer
49     end
50
51     output: spike                  // iaf_neuron emits spikes
```

Listing C.1: Implementation of the *integrate-and-fire* neuron in NESTML. (Part 1 of 2)

```
52      dynamics timestep(t ms):       // advance state by one time-step h to t
53        if r == 0: // not refractory
54          y3 = P30 * (y0 + I_e) + P31 * y1 + P32 * y2 + P33 * y3
55        else:
56          r = r - 1
57        end
58
59        // alpha shape PSCs
60        y2 = P21 * y1 + P22 * y2
61        y1 = y1 * P11
62
63        // Apply spikes delivered in step t
64        y1 = y1 + PSCInitialValue * spikeBuffer.getSum(t)
65
66        // threshold crossing
67        if y3 >= Theta:
68          r = RefractoryCounts
69          y3 = delta_V_reset
70
71          Spiking.emitSpike()
72        end
73
74        // set new input current
75        y0 = currentBuffer.getSum(t);
76      end
77
78      // setter functions for alias-variables
79      function set_V_th(v mV):
80        Theta = v - E_L
81      end
82
83      function set_V_reset(v mV):
84        delta_V_reset = v - E_L
85      end
86
87      function set_V_m(v mV):
88        y3 = v - E_L
89      end
90    end
91  end
```

Listing C.2: Implementation of the *integrate-and-fire* neuron in NESTML. (Part 2 of 2)

Appendix D

Generated NEST Code (neuron)

```
1  /* generated from model models.iaf_neuron*/
2  /* generated by template nestml.nest.neuron.NeuronHeader*/
3  /*
4   *  iaf_neuron.h
5   *
6   *  This file is part of NEST.
7   *
8   *  Copyright (C) 2004 The NEST Initiative
9   *
10  *  NEST is free software: you can redistribute it and/or modify
11  *  it under the terms of the GNU General Public License as published by
12  *  the Free Software Foundation, either version 2 of the License, or
13  *  (at your option) any later version.
14  *
15  *  NEST is distributed in the hope that it will be useful,
16  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
17  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18  *  GNU General Public License for more details.
19  *
20  *  You should have received a copy of the GNU General Public License
21  *  along with NEST.  If not, see <http://www.gnu.org/licenses/>.
22  *
23  */
24
25 #ifndef MODELS_IAF_NEURON_H
26 #define MODELS_IAF_NEURON_H
27
28 #include "nest.h"
29 #include "event.h"
30 #include "archiving_node.h"
31 #include "connection.h"
32 #include "universal_data_logger.h"
33 #include "dictdatum.h"
34
35 #include "units/unitless/real.h"
36 #include "units/electrics/mV.h"
37 #include "units/time/ms.h"
38 #include "ring_buffer.h"
39 #include "units/electrics/pA.h"
```

Listing D.1: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 1 of 13)

```
40 #include <cmath>
41 #include "units/unitless/integer.h"
42 #include "units/electrics/pF.h"
43
44 namespace models {
45
46   /* BeginDocumentation
47 Name: iaf_neuron .
48
49 Description:
50   Empty. TODO
51
52 Parameters:
53   y0              nest::double_t -  external current In mV_t.
54   y1              nest::double_t -  y1 and y2 are used in order to In mV_t.
55   y2              nest::double_t -  calculate alpha shape In mV_t.
56   y3              nest::double_t -  relative membrane potential In mV_t.
57   r               nest::long_t -  refractory steps to go In integer_t.
58   V_m             nest::double_t -  Membrane potential. In mV_t.
59
60   C_m             nest::double_t -  Capacity of the membrane. In pF_t.
61   tau_m           nest::double_t -  Membrane time constant. In ms_t.
62   tau_syn         nest::double_t -  Time constant of synaptic current. In ms_t.
63   t_ref           nest::double_t -  Refractory period. In ms_t.
64   E_L             nest::double_t -  Resting potential. In mV_t.
65   delta_V_reset   nest::double_t -  relative Resting potential. => 0 In mV_t.
66   Theta           nest::double_t -  elative threshold In mV_t.
67   I_e             nest::double_t -  constant external current. In pA_t.
68   V_th            nest::double_t -  some aliases In mV_t.
69   V_reset         nest::double_t -  Reset value of the membrane potential. In mV_t.
70
71
72 Remarks:
73   Empty
74
75 References:
76   Empty
77
78 Sends: nest::SpikeEvent
79
80 Receives: Spike, Current, DataLoggingRequest
81
82 Author:
83   TODO
84
85 SeeAlso:
86   Empty
87 */
88
89 class iaf_neuron : public nest::Archiving_Node
90 {
91 public:
92     /**
93      * The constructor is only used to create the model prototype in the model manager.
94      */
95     iaf_neuron();
```

Listing D.2: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 2 of 13)

```
96      /**
97       * The copy constructor is used to create model copies and instances of the model.
98       * @node The copy constructor needs to initialize the parameters and the state.
99       *       Initialization of buffers and interal variables is deferred to
100      *       @c init_buffers_() and @c calibrate().
101      */
102     iaf_neuron(const iaf_neuron&);
103
104     /**
105      * Import sets of overloaded virtual functions.
106      * This is necessary to ensure proper overload and overriding resolution.
107      * @see http://www.gotw.ca/gotw/005.htm.
108      */
109     using nest::Node::connect_sender;
110     using nest::Node::handle;
111
112     /**
113      * Used to validate that we can send nest::SpikeEvent to desired target:port.
114      */
115     nest::port check_connection(nest::Connection&, nest::port);
116
117     /**
118      * @defgroup mynest_handle Functions handling incoming events.
119      * We tell nest that we can handle incoming events of various types by
120      * defining @c handle() and @c connect_sender() for the given event.
121      * @{
122      */
123     void handle(nest::SpikeEvent &);        //! accept spikes
124     void handle(nest::CurrentEvent &);      //! accept input current
125     void handle(nest::DataLoggingRequest &);//! allow recording with multimeter
126
127     nest::port connect_sender(nest::SpikeEvent&, nest::port);
128     nest::port connect_sender(nest::CurrentEvent&, nest::port);
129     nest::port connect_sender(nest::DataLoggingRequest&, nest::port);
130     /** @} */
131
132     // SLI communication functions:
133     void get_status(DictionaryDatum &) const;
134     void set_status(const DictionaryDatum &);
135
136     // Generate function header
137     /* generated by template nestml.nest.function.FunctionHeader*/
138     void set_V_th (units::electrics::mV_t v);
139 /* generated by template nestml.nest.function.FunctionHeader*/
140     void set_V_reset (units::electrics::mV_t v);
141 /* generated by template nestml.nest.function.FunctionHeader*/
142     void set_V_m (units::electrics::mV_t v);
143
144
145     /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
146     inline units::electrics::mV_t get_y0() const { return S_.get_y0() ; }
147     inline void set_y0(const units::electrics::mV_t v) { S_.set_y0( v ) ; }
148
149 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
150     inline units::electrics::mV_t get_y1() const { return S_.get_y1() ; }
151     inline void set_y1(const units::electrics::mV_t v) { S_.set_y1( v ) ; }
```

Listing D.3: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 3 of 13)

```
152 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
153     inline units::electrics::mV_t get_y2() const { return S_.get_y2() ; }
154     inline void set_y2(const units::electrics::mV_t v) { S_.set_y2( v ) ; }
155
156 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
157     inline units::electrics::mV_t get_y3() const { return S_.get_y3() ; }
158     inline void set_y3(const units::electrics::mV_t v) { S_.set_y3( v ) ; }
159
160 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
161     inline units::unitless::integer_t get_r() const { return S_.get_r() ; }
162     inline void set_r(const units::unitless::integer_t v) { S_.set_r( v ) ; }
163
164 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
165     inline units::electrics::mV_t get_V_m() const { return get_y3() + get_E_L(); }
166
167
168     /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
169     inline units::electrics::pF_t get_C_m() const { return P_.get_C_m() ; }
170     inline void set_C_m(const units::electrics::pF_t v) { P_.set_C_m( v ) ; }
171
172 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
173     inline units::time::ms_t get_tau_m() const { return P_.get_tau_m() ; }
174     inline void set_tau_m(const units::time::ms_t v) { P_.set_tau_m( v ) ; }
175
176 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
177     inline units::time::ms_t get_tau_syn() const { return P_.get_tau_syn() ; }
178     inline void set_tau_syn(const units::time::ms_t v) { P_.set_tau_syn( v ) ; }
179
180 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
181     inline units::time::ms_t get_t_ref() const { return P_.get_t_ref() ; }
182     inline void set_t_ref(const units::time::ms_t v) { P_.set_t_ref( v ) ; }
183
184 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
185     inline units::electrics::mV_t get_E_L() const { return P_.get_E_L() ; }
186     inline void set_E_L(const units::electrics::mV_t v) { P_.set_E_L( v ) ; }
187
188 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
189     inline units::electrics::mV_t get_delta_V_reset() const
190         { return P_.get_delta_V_reset() ; }
191     inline void set_delta_V_reset(const units::electrics::mV_t v)
192         { P_.set_delta_V_reset( v ) ; }
193
194 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
195     inline units::electrics::mV_t get_Theta() const { return P_.get_Theta() ; }
196     inline void set_Theta(const units::electrics::mV_t v) { P_.set_Theta( v ) ; }
197
198 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
199     inline units::electrics::pA_t get_I_e() const { return P_.get_I_e() ; }
200     inline void set_I_e(const units::electrics::pA_t v) { P_.set_I_e( v ) ; }
201
202 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
203     inline units::electrics::mV_t get_V_th() const { return get_Theta() + get_E_L(); }
204
205 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
206     inline units::electrics::mV_t get_V_reset() const
207         { return get_delta_V_reset() + get_E_L(); }
```

Listing D.4: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 4 of 13)

```
208     /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
209     inline units::time::ms_t get_h() const { return V_.get_h() ; }
210     inline void set_h(const units::time::ms_t v) { V_.set_h( v ) ; }
211
212 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
213     inline units::unitless::real_t get_P11() const { return V_.get_P11() ; }
214     inline void set_P11(const units::unitless::real_t v) { V_.set_P11( v ) ; }
215
216 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
217     inline units::unitless::real_t get_P22() const { return V_.get_P22() ; }
218     inline void set_P22(const units::unitless::real_t v) { V_.set_P22( v ) ; }
219
220 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
221     inline units::unitless::real_t get_P33() const { return V_.get_P33() ; }
222     inline void set_P33(const units::unitless::real_t v) { V_.set_P33( v ) ; }
223
224 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
225     inline units::unitless::real_t get_P21() const { return V_.get_P21() ; }
226     inline void set_P21(const units::unitless::real_t v) { V_.set_P21( v ) ; }
227
228 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
229     inline units::unitless::real_t get_P30() const { return V_.get_P30() ; }
230     inline void set_P30(const units::unitless::real_t v) { V_.set_P30( v ) ; }
231
232 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
233     inline units::unitless::real_t get_P31() const { return V_.get_P31() ; }
234     inline void set_P31(const units::unitless::real_t v) { V_.set_P31( v ) ; }
235
236 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
237     inline units::unitless::real_t get_P32() const { return V_.get_P32() ; }
238     inline void set_P32(const units::unitless::real_t v) { V_.set_P32( v ) ; }
239
240 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
241     inline units::electrics::mV_t get_PSCInitialValue() const
242         { return V_.get_PSCInitialValue() ; }
243     inline void set_PSCInitialValue(const units::electrics::mV_t v)
244         { V_.set_PSCInitialValue( v ) ; }
245
246 /* generated by template nestml.nest.function.MemberVariableGetterSetter*/
247     inline units::unitless::integer_t get_RefractoryCounts() const
248         { return V_.get_RefractoryCounts() ; }
249     inline void set_RefractoryCounts(const units::unitless::integer_t v)
250         { V_.set_RefractoryCounts( v ) ; }
251
252
253
254     /* generated by template nestml.nest.function.BufferGetter*/
255     inline nest::RingBuffer& get_spikeBuffer() { return B_.get_spikeBuffer() ; }
256 /* generated by template nestml.nest.function.BufferGetter*/
257     inline nest::RingBuffer& get_currentBuffer() { return B_.get_currentBuffer() ; }
258
259
260 private:
261
262     //! Reset parameters and state of neuron.
```

Listing D.5: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 5 of 13)

```
263     //! Reset state of neuron.
264     void init_state_(const Node& proto);
265
266     //! Reset internal buffers of neuron.
267     void init_buffers_();
268
269     //! Initialize auxiliary quantities, leave parameters and state untouched.
270     void calibrate();
271
272     //! Take neuron through given time interval
273     void update(nest::Time const &, const nest::long_t, const nest::long_t);
274
275     // The next two classes need to be friends to access the State_ class/member
276     friend class nest::RecordablesMap<iaf_neuron>;
277     friend class nest::UniversalDataLogger<iaf_neuron>;
278
279     /**
280      * Dynamic state of the neuron.
281      *
282      * These are the state variables that are advanced in time by calls to
283      * @c update(). In many models, some or all of them can be set by the user
284      * through @c SetStatus. The state variables are initialized from the model
285      * prototype when the node is created. State variables are reset by @c
286      * ResetNetwork.
287      *
288      * @note State_ need neither copy constructor nor @c operator=(), since
289      *       all its members are copied properly by the default copy constructor
290      *       and assignment operator. Important:
291      *       - If State_ contained @c Time members, you need to define the
292      *         assignment operator to recalibrate all members of type @c Time . You
293      *         may also want to define the assignment operator.
294      *       - If State_ contained members that can not copy themselves, such
295      *         as C-style arrays, you need to define the copy constructor and
296      *         assignment operator to copy those members.
297      */
298     struct State_ {
299         /* generated by template nestml.nest.spl.MemberDeclaration*/
300         units::electrics::mV_t y0_;
301
302         /* generated by template nestml.nest.spl.MemberDeclaration*/
303         units::electrics::mV_t y1_;
304
305         /* generated by template nestml.nest.spl.MemberDeclaration*/
306         units::electrics::mV_t y2_;
307
308         /* generated by template nestml.nest.spl.MemberDeclaration*/
309         units::electrics::mV_t y3_;
310
311         /* generated by template nestml.nest.spl.MemberDeclaration*/
312         units::unitless::integer_t r_;
313
314         State_();
315
316         /** Store state values in dictionary. */
317         void get(DictionaryDatum&) const;
```

Listing D.6: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 6 of 13)

```
318         /**
319          * Set state values from dictionary.
320          */
321         void set(const DictionaryDatum&);
322
323
324         /* generated by template nestml.nest.function.StructGetterSetter*/
325         inline units::electrics::mV_t get_y0() const { return y0_ ; }
326         inline void set_y0(const units::electrics::mV_t v) { y0_ = v ; }
327
328 /* generated by template nestml.nest.function.StructGetterSetter*/
329         inline units::electrics::mV_t get_y1() const { return y1_ ; }
330         inline void set_y1(const units::electrics::mV_t v) { y1_ = v ; }
331
332 /* generated by template nestml.nest.function.StructGetterSetter*/
333         inline units::electrics::mV_t get_y2() const { return y2_ ; }
334         inline void set_y2(const units::electrics::mV_t v) { y2_ = v ; }
335
336 /* generated by template nestml.nest.function.StructGetterSetter*/
337         inline units::electrics::mV_t get_y3() const { return y3_ ; }
338         inline void set_y3(const units::electrics::mV_t v) { y3_ = v ; }
339
340 /* generated by template nestml.nest.function.StructGetterSetter*/
341         inline units::unitless::integer_t get_r() const { return r_ ; }
342         inline void set_r(const units::unitless::integer_t v) { r_ = v ; }
343
344
345     };
346
347     /**
348      * Free parameters of the neuron.
349      *
350      * These are the parameters that can be set by the user through @c SetStatus.
351      * They are initialized from the model prototype when the node is created.
352      * Parameters do not change during calls to @c update() and are not reset by
353      * @c ResetNetwork.
354      *
355      * @note Parameters_ need neither copy constructor nor @c operator=(), since
356      *       all its members are copied properly by the default copy constructor
357      *       and assignment operator. Important:
358      *       - If Parameters_ contained @c Time members, you need to define the
359      *         assignment operator to recalibrate all members of type @c Time . You
360      *         may also want to define the assignment operator.
361      *       - If Parameters_ contained members that can not copy themselves,such
362      *         as C-style arrays, you need to define the copy constructor and
363      *         assignment operator to copy those members.
364      */
365     struct Parameters_ {
366         /* generated by template nestml.nest.spl.MemberDeclaration*/
367         units::electrics::pF_t C_m_;
368
369         /* generated by template nestml.nest.spl.MemberDeclaration*/
370         units::time::ms_t tau_m_;
371
372         /* generated by template nestml.nest.spl.MemberDeclaration*/
373         units::time::ms_t tau_syn_;
```

Listing D.7: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 7 of 13)

```
374         /* generated by template nestml.nest.spl.MemberDeclaration*/
375         units::time::ms_t t_ref_;
376
377         /* generated by template nestml.nest.spl.MemberDeclaration*/
378         units::electrics::mV_t E_L_;
379
380         /* generated by template nestml.nest.spl.MemberDeclaration*/
381         units::electrics::mV_t delta_V_reset_;
382
383         /* generated by template nestml.nest.spl.MemberDeclaration*/
384         units::electrics::mV_t Theta_;
385
386         /* generated by template nestml.nest.spl.MemberDeclaration*/
387         units::electrics::pA_t I_e_;
388
389         /** Initialize parameters to their default values. */
390         Parameters_();
391
392         /** Store parameter values in dictionary. */
393         void get(DictionaryDatum&) const;
394
395         /** Set parameter values from dictionary. */
396         void set(const DictionaryDatum&);
397
398         /* generated by template nestml.nest.function.StructGetterSetter*/
399         inline units::electrics::pF_t get_C_m() const { return C_m_ ; }
400         inline void set_C_m(const units::electrics::pF_t v) { C_m_ = v ; }
401
402 /* generated by template nestml.nest.function.StructGetterSetter*/
403         inline units::time::ms_t get_tau_m() const { return tau_m_ ; }
404         inline void set_tau_m(const units::time::ms_t v) { tau_m_ = v ; }
405
406 /* generated by template nestml.nest.function.StructGetterSetter*/
407         inline units::time::ms_t get_tau_syn() const { return tau_syn_ ; }
408         inline void set_tau_syn(const units::time::ms_t v) { tau_syn_ = v ; }
409
410 /* generated by template nestml.nest.function.StructGetterSetter*/
411         inline units::time::ms_t get_t_ref() const { return t_ref_ ; }
412         inline void set_t_ref(const units::time::ms_t v) { t_ref_ = v ; }
413
414 /* generated by template nestml.nest.function.StructGetterSetter*/
415         inline units::electrics::mV_t get_E_L() const { return E_L_ ; }
416         inline void set_E_L(const units::electrics::mV_t v) { E_L_ = v ; }
417
418 /* generated by template nestml.nest.function.StructGetterSetter*/
419         inline units::electrics::mV_t get_delta_V_reset() const
420             { return delta_V_reset_ ; }
421         inline void set_delta_V_reset(const units::electrics::mV_t v)
422             { delta_V_reset_ = v ; }
423
424 /* generated by template nestml.nest.function.StructGetterSetter*/
425         inline units::electrics::mV_t get_Theta() const { return Theta_ ; }
426         inline void set_Theta(const units::electrics::mV_t v) { Theta_ = v ; }
```

Listing D.8: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 8 of 13)

```
427 /* generated by template nestml.nest.function.StructGetterSetter*/
428         inline units::electrics::pA_t get_I_e() const { return I_e_ ; }
429         inline void set_I_e(const units::electrics::pA_t v) { I_e_ = v ; }
430     };
431
432     /**
433      * Internal variables of the neuron.
434      * These variables must be initialized by @c calibrate, which is called before
435      * the first call to @c update() upon each call to @c Simulate.
436      * @node Variables_ needs neither constructor, copy constructor or assignment
437      *       operator, since it is initialized by @c calibrate(). If Variables_
438      *       has members that can not destroy themselves, Variables_ will need a
439      *       destructor.
440      */
441     struct Variables_ {
442         /* generated by template nestml.nest.spl.MemberDeclaration*/
443         units::time::ms_t h_;
444
445         /* generated by template nestml.nest.spl.MemberDeclaration*/
446         units::unitless::real_t P11_;
447
448         /* generated by template nestml.nest.spl.MemberDeclaration*/
449         units::unitless::real_t P22_;
450
451         /* generated by template nestml.nest.spl.MemberDeclaration*/
452         units::unitless::real_t P33_;
453
454         /* generated by template nestml.nest.spl.MemberDeclaration*/
455         units::unitless::real_t P21_;
456
457         /* generated by template nestml.nest.spl.MemberDeclaration*/
458         units::unitless::real_t P30_;
459
460         /* generated by template nestml.nest.spl.MemberDeclaration*/
461         units::unitless::real_t P31_;
462
463         /* generated by template nestml.nest.spl.MemberDeclaration*/
464         units::unitless::real_t P32_;
465
466         /* generated by template nestml.nest.spl.MemberDeclaration*/
467         units::electrics::mV_t PSCInitialValue_;
468
469         /* generated by template nestml.nest.spl.MemberDeclaration*/
470         units::unitless::integer_t RefractoryCounts_;
471
472         /* generated by template nestml.nest.function.StructGetterSetter*/
473         inline units::time::ms_t get_h() const { return h_ ; }
474         inline void set_h(const units::time::ms_t v) { h_ = v ; }
475
476 /* generated by template nestml.nest.function.StructGetterSetter*/
477         inline units::unitless::real_t get_P11() const { return P11_ ; }
478         inline void set_P11(const units::unitless::real_t v) { P11_ = v ; }
479
480 /* generated by template nestml.nest.function.StructGetterSetter*/
481         inline units::unitless::real_t get_P22() const { return P22_ ; }
482         inline void set_P22(const units::unitless::real_t v) { P22_ = v ; }
```

Listing D.9: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 9 of 13)

```
483 /* generated by template nestml.nest.function.StructGetterSetter*/
484         inline units::unitless::real_t get_P33() const { return P33_ ; }
485         inline void set_P33(const units::unitless::real_t v) { P33_ = v ; }
486
487 /* generated by template nestml.nest.function.StructGetterSetter*/
488         inline units::unitless::real_t get_P21() const { return P21_ ; }
489         inline void set_P21(const units::unitless::real_t v) { P21_ = v ; }
490
491 /* generated by template nestml.nest.function.StructGetterSetter*/
492         inline units::unitless::real_t get_P30() const { return P30_ ; }
493         inline void set_P30(const units::unitless::real_t v) { P30_ = v ; }
494
495 /* generated by template nestml.nest.function.StructGetterSetter*/
496         inline units::unitless::real_t get_P31() const { return P31_ ; }
497         inline void set_P31(const units::unitless::real_t v) { P31_ = v ; }
498
499 /* generated by template nestml.nest.function.StructGetterSetter*/
500         inline units::unitless::real_t get_P32() const { return P32_ ; }
501         inline void set_P32(const units::unitless::real_t v) { P32_ = v ; }
502
503 /* generated by template nestml.nest.function.StructGetterSetter*/
504         inline units::electrics::mV_t get_PSCInitialValue() const
505             { return PSCInitialValue_ ; }
506         inline void set_PSCInitialValue(const units::electrics::mV_t v)
507             { PSCInitialValue_ = v ; }
508
509 /* generated by template nestml.nest.function.StructGetterSetter*/
510         inline units::unitless::integer_t get_RefractoryCounts() const
511             { return RefractoryCounts_ ; }
512         inline void set_RefractoryCounts(const units::unitless::integer_t v)
513             { RefractoryCounts_ = v ; }
514     };
515
516     /**
517      * Buffers of the neuron.
518      * Ususally buffers for incoming spikes and data logged for analog recorders.
519      * Buffers must be initialized by @c init_buffers_(), which is called before
520      * @c calibrate() on the first call to @c Simulate after the start of NEST,
521      * ResetKernel or ResetNetwork.
522      * @node Buffers_ needs neither constructor, copy constructor or assignment
523      *       operator, since it is initialized by @c init_nodes_(). If Buffers_
524      *       has members that can not destroy themselves, Buffers_ will need a
525      *       destructor.
526      */
527     struct Buffers_ {
528         Buffers_(iaf_neuron&);
529         Buffers_(const Buffers_ &, iaf_neuron&);
530
531             /* generated by template nestml.nest.buffer.BufferDeclaration*/
532         nest::RingBuffer spikeBuffer_;
533                             //!< Buffer incoming spikes through delay, as sum
534
535             /* generated by template nestml.nest.buffer.BufferDeclaration*/
536         nest::RingBuffer currentBuffer_;
537                             //!< Buffer incoming currents through delay, as sum
```

Listing D.10: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 10 of 13)

```
538         /** Logger for all analog data */
539         nest::UniversalDataLogger<iaf_neuron> logger_;
540
541         /* generated by template nestml.nest.function.BufferGetter*/
542         inline nest::RingBuffer& get_spikeBuffer() { return spikeBuffer_ ; }
543 /* generated by template nestml.nest.function.BufferGetter*/
544         inline nest::RingBuffer& get_currentBuffer() { return currentBuffer_ ; }
545     };
546
547     /**
548      * @defgroup pif_members Member variables of neuron model.
549      * Each model neuron should have precisely the following four data members,
550      * which are one instance each of the parameters, state, buffers and variables
551      * structures. Experience indicates that the state and variables member should
552      * be next to each other to achieve good efficiency (caching).
553      * @note Devices require one additional data member, an instance of the @c Device
554      *       child class they belong to.
555      * @{
556      */
557     Parameters_ P_;  //!< Free parameters.
558     State_      S_;  //!< Dynamic state.
559     Variables_  V_;  //!< Internal Variables
560     Buffers_    B_;  //!< Buffers.
561
562     //! Mapping of recordables names to access functions
563     static nest::RecordablesMap<iaf_neuron> recordablesMap_;
564     /** @} */
565 }; /* neuron iaf_neuron */
566 } /* namespace models */
567
568 inline
569 nest::port models::iaf_neuron::check_connection(nest::Connection& c,
570                                                 nest::port receptor_type)
571 {
572   // You should usually not change the code in this function.
573   // It confirms that the target of connection @c c accepts @c nest::SpikeEvent on
574   // the given @c receptor_type.
575   nest::SpikeEvent e;
576   e.set_sender(*this);
577   c.check_event(e);
578   return c.get_target()->connect_sender(e, receptor_type);
579 }
580
581 inline
582 nest::port models::iaf_neuron::connect_sender(nest::SpikeEvent&,
583                                               nest::port receptor_type)
584 {
585   // You should usually not change the code in this function.
586   // It confirms to the connection management system that we are able
587   // to handle @c SpikeEvent on port 0. You need to extend the function
588   // if you want to differentiate between input ports.
589   if (receptor_type != 0)
590     throw nest::UnknownReceptorType(receptor_type, get_name());
591   return 0;
592 }
```

Listing D.11: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 11 of 13)

```
593 inline
594 nest::port models::iaf_neuron::connect_sender(nest::CurrentEvent&,
595                                                nest::port receptor_type)
596 {
597   // You should usually not change the code in this function.
598   // It confirms to the connection management system that we are able
599   // to handle @c CurrentEvent on port 0. You need to extend the function
600   // if you want to differentiate between input ports.
601   if (receptor_type != 0)
602     throw nest::UnknownReceptorType(receptor_type, get_name());
603   return 0;
604 }
605
606 inline
607 nest::port models::iaf_neuron::connect_sender(nest::DataLoggingRequest& dlr,
608                          nest::port receptor_type)
609 {
610   // You should usually not change the code in this function.
611   // It confirms to the connection management system that we are able
612   // to handle @c DataLoggingRequest on port 0.
613   // The function also tells the built-in UniversalDataLogger that this node
614   // is recorded from and that it thus needs to collect data during simulation.
615   if (receptor_type != 0)
616     throw nest::UnknownReceptorType(receptor_type, get_name());
617
618   return B_.logger_.connect_logging_device(dlr, recordablesMap_);
619 }
620
621
622 inline
623 void models::iaf_neuron::get_status(DictionaryDatum &d) const
624 {
625   P_.get(d);
626
627   /* generated by template nestml.nest.function.WriteInDictionary*/
628   def<units::electrics::mV_t>(d, "V_th", get_V_th());
629 /* generated by template nestml.nest.function.WriteInDictionary*/
630   def<units::electrics::mV_t>(d, "V_reset", get_V_reset());
631
632   S_.get(d);
633
634   /* generated by template nestml.nest.function.WriteInDictionary*/
635   def<units::electrics::mV_t>(d, "V_m", get_V_m());
636   (*d)[nest::names::recordables] = recordablesMap_.get_list();
637 }
638
639 inline
640 void models::iaf_neuron::set_status(const DictionaryDatum &d)
641 {
642   Parameters_ ptmp = P_;  // temporary copy in case of errors
643   ptmp.set(d);                       // throws if BadProperty
644
645   /* generated by template nestml.nest.function.ReadFromDictionary*/
646   units::electrics::mV_t tmp_V_th;
647   updateValue<units::electrics::mV_t>(d, "V_th", tmp_V_th);
648   set_V_th(tmp_V_th);
```

Listing D.12: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 12 of 13)

```
649 /* generated by template nestml.nest.function.ReadFromDictionary*/
650   units::electrics::mV_t tmp_V_reset;
651   updateValue<units::electrics::mV_t>(d, "V_reset", tmp_V_reset);
652   set_V_reset(tmp_V_reset);
653
654
655   State_       stmp = S_;  // temporary copy in case of errors
656   stmp.set(d);                          // throws if BadProperty
657
658   /* generated by template nestml.nest.function.ReadFromDictionary*/
659   units::electrics::mV_t tmp_V_m;
660   updateValue<units::electrics::mV_t>(d, "V_m", tmp_V_m);
661   set_V_m(tmp_V_m);
662
663   // if we get here, temporaries contain consistent set of properties
664   P_ = ptmp;
665   S_ = stmp;
666 }
667
668 #endif /* #ifndef MODELS_IAF_NEURON_H */
```

Listing D.13: Generated header of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 13 of 13)

```
1  /* generated from model models.iaf_neuron*/
2  /* generated by template nestml.nest.neuron.NeuronClass*/
3  /*
4   *  iaf_neuron.cpp
5   *
6   *  This file is part of NEST.
7   *
8   *  Copyright (C) 2004 The NEST Initiative
9   *
10  *  NEST is free software: you can redistribute it and/or modify
11  *  it under the terms of the GNU General Public License as published by
12  *  the Free Software Foundation, either version 2 of the License, or
13  *  (at your option) any later version.
14  *
15  *  NEST is distributed in the hope that it will be useful,
16  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
17  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18  *  GNU General Public License for more details.
19  *
20  *  You should have received a copy of the GNU General Public License
21  *  along with NEST.  If not, see <http://www.gnu.org/licenses/>.
22  *
23  */
24
25 #include "exceptions.h"
26 #include "network.h"
27 #include "dict.h"
28 #include "integerdatum.h"
29 #include "doubledatum.h"
30 #include "dictutils.h"
31 #include "numerics.h"
32 #include "universal_data_logger_impl.h"
```

Listing D.14: Generated implementation of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 1 of 8)

```
33 #include <limits>
34
35 #include "models/iaf_neuron.h"
36
37 /* ----------------------------------------------------------------
38  * Recordables map
39  * ---------------------------------------------------------------- */
40
41 nest::RecordablesMap<models::iaf_neuron> models::iaf_neuron::recordablesMap_;
42
43 namespace nest
44 {
45   // Override the create() method with one call to RecordablesMap::insert_()
46   // for each quantity to be recorded.
47   template <>
48   void RecordablesMap<models::iaf_neuron>::create()
49   {
50     // use standard names whereever you can for consistency!
51     /* generated by template nestml.nest.function.RecordCallback*/
52     insert_("y0", &models::iaf_neuron::get_y0);
53 /* generated by template nestml.nest.function.RecordCallback*/
54     insert_("y1", &models::iaf_neuron::get_y1);
55 /* generated by template nestml.nest.function.RecordCallback*/
56     insert_("y2", &models::iaf_neuron::get_y2);
57 /* generated by template nestml.nest.function.RecordCallback*/
58     insert_("y3", &models::iaf_neuron::get_y3);
59 /* generated by template nestml.nest.function.RecordCallback*/
60     insert_("V_m", &models::iaf_neuron::get_V_m);
61   }
62 }
63
64 /* ----------------------------------------------------------------
65  * Default constructors defining default parameters and state
66  * ---------------------------------------------------------------- */
67
68 models::iaf_neuron::Parameters_::Parameters_()
69   /* generated by template nestml.nest.spl.MemberInitialisation*/
70   : C_m_( 250 ) // pF
71   , tau_m_( 10 ) // ms
72   , tau_syn_( 2 ) // ms
73   , t_ref_( 2 ) // ms
74   , E_L_( -70 ) // mV
75   , delta_V_reset_( -70 - get_E_L() ) // mV
76   , Theta_( -55 - get_E_L() ) // mV
77   , I_e_( 0 ) // pA
78 {}
79
80 models::iaf_neuron::State_::State_()
81   /* generated by template nestml.nest.spl.MemberInitialisation*/
82   : y0_() // mV
83   , y1_() // mV
84   , y2_() // mV
85   , y3_() // mV
86   , r_() // integer
87 {}
```

Listing D.15: Generated implementation of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 2 of 8)

```
88  /* ----------------------------------------------------------------
89   * Parameter and state extractions and manipulation functions
90   * ---------------------------------------------------------------- */
91
92  void
93  models::iaf_neuron::Parameters_::get(DictionaryDatum &d) const
94  {
95    /* generated by template nestml.nest.function.WriteInDictionary*/
96    def<units::electrics::pF_t>(d, "C_m", get_C_m());
97
98  /* generated by template nestml.nest.function.WriteInDictionary*/
99    def<units::time::ms_t>(d, "tau_m", get_tau_m());
100
101 /* generated by template nestml.nest.function.WriteInDictionary*/
102   def<units::time::ms_t>(d, "tau_syn", get_tau_syn());
103
104 /* generated by template nestml.nest.function.WriteInDictionary*/
105   def<units::time::ms_t>(d, "t_ref", get_t_ref());
106
107 /* generated by template nestml.nest.function.WriteInDictionary*/
108   def<units::electrics::mV_t>(d, "E_L", get_E_L());
109
110 /* generated by template nestml.nest.function.WriteInDictionary*/
111   def<units::electrics::mV_t>(d, "delta_V_reset", get_delta_V_reset());
112
113 /* generated by template nestml.nest.function.WriteInDictionary*/
114   def<units::electrics::mV_t>(d, "Theta", get_Theta());
115
116 /* generated by template nestml.nest.function.WriteInDictionary*/
117   def<units::electrics::pA_t>(d, "I_e", get_I_e());
118 }
119
120 void
121 models::iaf_neuron::Parameters_::set(const DictionaryDatum& d)
122 {
123   /* generated by template nestml.nest.function.ReadFromDictionary*/
124   units::electrics::pF_t tmp_C_m;
125   updateValue<units::electrics::pF_t>(d, "C_m", tmp_C_m);
126   set_C_m(tmp_C_m);
127
128 /* generated by template nestml.nest.function.ReadFromDictionary*/
129   units::time::ms_t tmp_tau_m;
130   updateValue<units::time::ms_t>(d, "tau_m", tmp_tau_m);
131   set_tau_m(tmp_tau_m);
132
133 /* generated by template nestml.nest.function.ReadFromDictionary*/
134   units::time::ms_t tmp_tau_syn;
135   updateValue<units::time::ms_t>(d, "tau_syn", tmp_tau_syn);
136   set_tau_syn(tmp_tau_syn);
137
138 /* generated by template nestml.nest.function.ReadFromDictionary*/
139   units::time::ms_t tmp_t_ref;
140   updateValue<units::time::ms_t>(d, "t_ref", tmp_t_ref);
141   set_t_ref(tmp_t_ref);
```

Listing D.16: Generated implementation of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 3 of 8)

```
142  /* generated by template nestml.nest.function.ReadFromDictionary*/
143    units::electrics::mV_t tmp_E_L;
144    updateValue<units::electrics::mV_t>(d, "E_L", tmp_E_L);
145    set_E_L(tmp_E_L);
146
147  /* generated by template nestml.nest.function.ReadFromDictionary*/
148    units::electrics::mV_t tmp_delta_V_reset;
149    updateValue<units::electrics::mV_t>(d, "delta_V_reset", tmp_delta_V_reset);
150    set_delta_V_reset(tmp_delta_V_reset);
151
152  /* generated by template nestml.nest.function.ReadFromDictionary*/
153    units::electrics::mV_t tmp_Theta;
154    updateValue<units::electrics::mV_t>(d, "Theta", tmp_Theta);
155    set_Theta(tmp_Theta);
156
157  /* generated by template nestml.nest.function.ReadFromDictionary*/
158    units::electrics::pA_t tmp_I_e;
159    updateValue<units::electrics::pA_t>(d, "I_e", tmp_I_e);
160    set_I_e(tmp_I_e);
161  }
162
163  void
164  models::iaf_neuron::State_::get(DictionaryDatum &d) const
165  {
166    /* generated by template nestml.nest.function.WriteInDictionary*/
167    def<units::electrics::mV_t>(d, "y0", get_y0());
168
169  /* generated by template nestml.nest.function.WriteInDictionary*/
170    def<units::electrics::mV_t>(d, "y1", get_y1());
171
172  /* generated by template nestml.nest.function.WriteInDictionary*/
173    def<units::electrics::mV_t>(d, "y2", get_y2());
174
175  /* generated by template nestml.nest.function.WriteInDictionary*/
176    def<units::electrics::mV_t>(d, "y3", get_y3());
177
178  /* generated by template nestml.nest.function.WriteInDictionary*/
179    def<units::unitless::integer_t>(d, "r", get_r());
180  }
181
182  void
183  models::iaf_neuron::State_::set(const DictionaryDatum& d)
184  {
185    /* generated by template nestml.nest.function.ReadFromDictionary*/
186    units::electrics::mV_t tmp_y0;
187    updateValue<units::electrics::mV_t>(d, "y0", tmp_y0);
188    set_y0(tmp_y0);
189
190  /* generated by template nestml.nest.function.ReadFromDictionary*/
191    units::electrics::mV_t tmp_y1;
192    updateValue<units::electrics::mV_t>(d, "y1", tmp_y1);
193    set_y1(tmp_y1);
```

Listing D.17: Generated implementation of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 4 of 8)

```
194 /* generated by template nestml.nest.function.ReadFromDictionary*/
195   units::electrics::mV_t tmp_y2;
196   updateValue<units::electrics::mV_t>(d, "y2", tmp_y2);
197   set_y2(tmp_y2);
198
199 /* generated by template nestml.nest.function.ReadFromDictionary*/
200   units::electrics::mV_t tmp_y3;
201   updateValue<units::electrics::mV_t>(d, "y3", tmp_y3);
202   set_y3(tmp_y3);
203
204 /* generated by template nestml.nest.function.ReadFromDictionary*/
205   units::unitless::integer_t tmp_r;
206   updateValue<units::unitless::integer_t>(d, "r", tmp_r);
207   set_r(tmp_r);
208 }
209
210 models::iaf_neuron::Buffers_::Buffers_(iaf_neuron &n)
211   : logger_(n)
212 {}
213
214 models::iaf_neuron::Buffers_::Buffers_(const Buffers_ &, iaf_neuron &n)
215   : logger_(n)
216 {}
217
218
219 /* ----------------------------------------------------------------
220  * Default and copy constructor for node
221  * ---------------------------------------------------------------- */
222
223 models::iaf_neuron::iaf_neuron()
224   : Archiving_Node(),
225     P_(),
226     S_(),
227     B_(*this)
228 {
229   recordablesMap_.create();
230 }
231
232 models::iaf_neuron::iaf_neuron(const iaf_neuron& n)
233   : Archiving_Node(n),
234     P_(n.P_),
235     S_(n.S_),
236     B_(n.B_, *this)
237 {}
238
239 /* ----------------------------------------------------------------
240  * Node initialization functions
241  * ---------------------------------------------------------------- */
242
243 void
244 models::iaf_neuron::init_state_(const Node& proto)
245 {
246   const iaf_neuron& pr = downcast<iaf_neuron>(proto);
247   S_ = pr.S_;
248 }
```

Listing D.18: Generated implementation of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 5 of 8)

```
249 void
250 models::iaf_neuron::init_buffers_()
251 {
252   /* generated by template nestml.nest.buffer.BufferInit*/
253   get_spikeBuffer().clear(); //includes resize
254
255   /* generated by template nestml.nest.buffer.BufferInit*/
256   get_currentBuffer().clear(); //includes resize
257
258   B_.logger_.reset(); // includes resize
259   Archiving_Node::clear_history();
260 }
261
262 void
263 models::iaf_neuron::calibrate()
264 {
265   B_.logger_.init();
266
267   /* generated by template nestml.nest.function.Calibrate*/
268   set_h( nest::Time::get_resolution().get_ms() );
269 /* generated by template nestml.nest.function.Calibrate*/
270   set_P11( std::pow( M_E , ( - get_h() / get_tau_syn() ) ) );
271 /* generated by template nestml.nest.function.Calibrate*/
272   set_P22( get_P11() );
273 /* generated by template nestml.nest.function.Calibrate*/
274   set_P33( std::pow( M_E , ( - get_h() / get_tau_m() ) ) );
275 /* generated by template nestml.nest.function.Calibrate*/
276   set_P21( get_h() * get_P11() );
277 /* generated by template nestml.nest.function.Calibrate*/
278   set_P30( 1 / get_C_m() * ( 1 - get_P33() ) * get_tau_m() );
279 /* generated by template nestml.nest.function.Calibrate*/
280   set_P31( 1 / get_C_m() * (
281           ( get_P11() - get_P33() ) / ( - 1 / get_tau_syn() - - 1 / get_tau_m() )
282         - get_h() * get_P11() ) / ( - 1 / get_tau_m() - - 1 / get_tau_syn() ) );
283 /* generated by template nestml.nest.function.Calibrate*/
284   set_P32( 1 / get_C_m() * ( get_P33() - get_P11() ) /
285             ( - 1 / get_tau_m() - - 1 / get_tau_syn() ) );
286 /* generated by template nestml.nest.function.Calibrate*/
287   set_PSCInitialValue( 1 * M_E / get_tau_syn() );
288 /* generated by template nestml.nest.function.Calibrate*/
289   set_RefractoryCounts( nest::Time(nest::Time::ms( get_t_ref() )).get_steps() );
290
291 }
292
293 /* ----------------------------------------------------------------
294  * Update and spike handling functions
295  * ---------------------------------------------------------------- */
296
297 void
298 models::iaf_neuron::update(nest::Time const & origin, const nest::long_t from,
299                            const nest::long_t to)
300 {
301   /* generated by template nestml.nest.function.DynamicsImplementation*/
302   /* generated by template nestml.nest.function.TimestepDynamics*/
303   assert(to >= 0 && (nest::delay) from < nest::Scheduler::get_min_delay());
304   assert(from < to);
```

Listing D.19: Generated implementation of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 6 of 8)

```
305   units::time::ms_t t;
306   for ( nest::long_t lag = from ; lag < to ; ++lag )
307   {
308     t = nest::Time(nest::Time::step( lag )).get_ms() + origin.get_ms();
309     /* generated by template nestml.nest.spl.BlockStatement*/
310     if (get_r() == 0) {
311       set_y3( get_P30() * ( get_y0() + get_I_e() ) + get_P31() * get_y1()
312                 + get_P32() * get_y2() + get_P33() * get_y3() );
313     } else {
314       set_r( get_r() - 1 );
315     } /* if end */
316
317     set_y2( get_P21() * get_y1() + get_P22() * get_y2() );
318     set_y1( get_y1() * get_P11() );
319     set_y1( get_y1() + get_PSCInitialValue() * get_spikeBuffer().get_value(
320               nest::Time(nest::Time::ms( t - origin.get_ms() )).get_steps()) );
321
322     if (get_y3() >= get_Theta()) {
323       set_r( get_RefractoryCounts() );
324       set_y3( get_delta_V_reset() );
325
326       set_spiketime(nest::Time::step(origin.get_steps()+lag+1));
327       nest::SpikeEvent se;
328       network()->send(*this, se, lag) ;
329
330     } /* if end */
331
332     set_y0( get_currentBuffer().get_value(
333             nest::Time(nest::Time::ms( t - origin.get_ms() )).get_steps()) );
334
335     // voltage logging
336     B_.logger_.record_data(origin.get_steps()+lag);
337   } /* for end */
338 }
339
340 void
341 models::iaf_neuron::handle(nest::SpikeEvent & e)
342 {
343   assert(e.get_delay() > 0);
344
345   const double_t weight = e.get_weight();
346   const double_t multiplicity = e.get_multiplicity();
347
348   /* generated by template nestml.nest.buffer.BufferFill*/
349   // excitatory && inhibitory
350   get_spikeBuffer().add_value(e.get_rel_delivery_steps( network()->get_slice_origin()),
351                 weight * multiplicity );
352 }
353
354 void
355 models::iaf_neuron::handle(nest::CurrentEvent& e)
356 {
357   assert(e.get_delay() > 0);
358
359   const double_t current=e.get_current();
360   const double_t weight=e.get_weight();
```

Listing D.20: Generated implementation of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 7 of 8)

```
361 /* generated by template nestml.nest.buffer.BufferFill*/
362   get_currentBuffer().add_value(e.get_rel_delivery_steps(
363                 network()->get_slice_origin()), weight * current );
364 }
365
366 void
367 models::iaf_neuron::handle(nest::DataLoggingRequest& e)
368 {
369   B_.logger_.handle(e);
370 }
371
372 /* ----------------------------------------------------------------
373  * Additional functions
374  * ---------------------------------------------------------------- */
375
376 /* generated by template nestml.nest.function.FunctionImplementation*/
377 void
378 models::iaf_neuron::set_V_th (units::electrics::mV_t v)
379 {
380   /* generated by template nestml.nest.spl.BlockStatement*/
381   set_Theta( v - get_E_L() );
382 }
383
384 /* generated by template nestml.nest.function.FunctionImplementation*/
385 void
386 models::iaf_neuron::set_V_reset (units::electrics::mV_t v)
387 {
388   /* generated by template nestml.nest.spl.BlockStatement*/
389   set_delta_V_reset( v - get_E_L() );
390 }
391
392 /* generated by template nestml.nest.function.FunctionImplementation*/
393 void
394 models::iaf_neuron::set_V_m (units::electrics::mV_t v)
395 {
396   /* generated by template nestml.nest.spl.BlockStatement*/
397   set_y3( v - get_E_L() );
398 }
```

Listing D.21: Generated implementation of the *integrate-and-fire* neuron in NESTML in Listing C.1. (Part 8 of 8)

Appendix E

Generated NEST Code (unit)

```
1  package units:
2      unit sampleUnit Real ( -13.67 ... 17e4 ]
3  end
```

Listing E.1: The sampleUnit model in NESTML.

```
1  /*
2   *  sampleUnit.h
3   *
4   *  This file is part of NEST.
5   *
6   *  Copyright (C) 2004 The NEST Initiative
7   *
8   *  NEST is free software: you can redistribute it and/or modify
9   *  it under the terms of the GNU General Public License as published by
10  *  the Free Software Foundation, either version 2 of the License, or
11  *  (at your option) any later version.
12  *
13  *  NEST is distributed in the hope that it will be useful,
14  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
15  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  *  GNU General Public License for more details.
17  *
18  *  You should have received a copy of the GNU General Public License
19  *  along with NEST.  If not, see <http://www.gnu.org/licenses/>.
20  *
21  */
22
23 #ifndef UNITS_SAMPLEUNIT_H
24 #define UNITS_SAMPLEUNIT_H
25
26 #include "nest.h"
27
28 namespace units {
29
30 // unit sampleUnit Real ( -13.67 ... 17e4 ]
31 typedef nest::double_t sampleUnit_t;
```

Listing E.2: Generated header of the sampleUnit in Listing E.1. (Part 1 of 2)

```
1  /*
2   * Checks, whether the argument is in the range of a sampleUnit.
3   * @return: 0 if argument not in range
4   *          1 otherwise
5   */
6  bool check_sampleUnit(const sampleUnit_t &);
7
8  } /* namespace units */
9
10 #endif
11 /* #ifndef UNITS_SAMPLEUNIT_H */
```

Listing E.3: Generated header of the sampleUnit in Listing E.1. (Part 2 of 2)

```
1  /*
2   *  sampleUnit.cpp
3   *
4   *  This file is part of NEST.
5   *
6   *  Copyright (C) 2004 The NEST Initiative
7   *
8   *  NEST is free software: you can redistribute it and/or modify
9   *  it under the terms of the GNU General Public License as published by
10  *  the Free Software Foundation, either version 2 of the License, or
11  *  (at your option) any later version.
12  *
13  *  NEST is distributed in the hope that it will be useful,
14  *  but WITHOUT ANY WARRANTY; without even the implied warranty of
15  *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  *  GNU General Public License for more details.
17  *
18  *  You should have received a copy of the GNU General Public License
19  *  along with NEST.  If not, see <http://www.gnu.org/licenses/>.
20  *
21  */
22
23 #include "units/sampleUnit.h"
24
25 bool
26 units::check_sampleUnit(const units::sampleUnit_t &var)
27 {
28     // assume var is inside the range
29     bool result = true;
30     // test, if var is smaller than left limit
31     if (var <= -13.67 )
32     {
33         result = false;
34     }
35
36     // test, if var is bigger than right limit
37     if (var > 17e4 )
38     {
39         result = false;
40     }
41
42     return result;
43 }
```

Listing E.4: Generated implementation of the sampleUnit in Listing E.1