RWTH Aachen University
Software Engineering Group

Reengineering of NestML with Python

Master Thesis

presented by

Perun, Konstantin

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Prof. Dr. H. Lichter

1st Advisor: Dr. nat. Dimitri Plotnikov

2nd Advisor: Guido Trensch

The present work was submitted to the Chair of Software Engineering

Aachen, April 9, 2018

Eidesstattliche Versicherung

Perun, Konstantin

331109

Name, Vorname

Matrikelnummer (freiwillige Angabe)

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

Reengineering of NestML with Python

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

Mönchengladbach, 01.12.2017

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

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

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

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

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

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Mönchengladbach, 01.12.2017

Ort, Datum

Unterschrift

Kurzfassung

Die NEST Modeling Language (**NestML**, [PRB$^+$16]) ist eine domänenspezifische Model-lierungssprache, die mit dem Ziel entwickelt wurde einfache Modellierung und Erstellung von lauffähigen Modellen für den NEST-Simulator [GD07] zu ermöglichen. Seit der Entste-hung im Jahr 2012 wurde die Sprache und die unterstützende Sammlung an Werkzeugen um neue Konzepte erweitert, während die Programmiersprache Java als Plattform bei-behalten wurde. Bedingt durch die verhältnismäßig geringe Verbreitung von Java in der neurowissenschaftlichen Domäne [MBD$^+$15] führt dies dazu, dass die Wartung und Er-weiterung der Software mit enormen Mehraufwand für Neurowissenschaftler verbunden ist. Dieser Umstand steht im Konflikt zum Grundgedanken von NestML, der Entwick-lung einer modularen und erweiterbaren Modellierungssprache für neurowissenschaftliche Konzepte.

 Mehr als 90% der Kosten bei der Entwicklung und Nutzung von Softwaresystemen entste-hen in der Wartungsphase [MHDH13]. Dieser Umstand macht vorausschauende Planung und Design von Software unabdingbar. Während Fehler und Designentscheidungen, die das Programmieren im Kleinen betreffen, meist durch geeigneten Einsatz von Datenab-straktion und Modularisierung abgemildert werden können, sind Entscheidungen auf der Architektur- und Plattformebene langfristig kostspielig und haben einen Effekt über die gesamte Lebensdauer eines Projektes [Wes02]. Somit gewinnt **Reengineering** von exi-stierender Software immer mehr an Bedeutung, da bestehende Systeme oft auf veralteten Technologien aufbauen, Fehler im grundlegenden Aufbau aufweisen oder nicht länger den gegebenen Anforderungen entsprechen. Hier hat sich die Anpassung der Plattform und der Softwareumgebung als eine mögliche Lösung bewährt [Som15].

 Diese Arbeit wird sich mit dem Überarbeiten und Portieren der existierenden NestML Werkzeugsammlung auf **Python** [van95] als die neue Zielplattform beschäftigen. Diese vor allem in der neurowissenschaftlichen Domäne überwiegend verwendete Programmier-sprache [MBD$^+$15] soll die Nutzung und Einbindung von NestML in bestehende Syste-me erleichtern, eigenständige Erweiterbarkeit durch Neurowissenschaftler ermöglichen und Brückentechnologien überflüssig machen. Um einen hohen Grad an Modularität und Wart-barkeit des Systems zu garantieren wird dabei die **MontiCore Language Workbanch** [KRV10] verwendet und um Python als eine neue Zielplattform für Codegenerierung er-weitert.

Abstract

The NEST Modeling Language (**NestML**, [PRB+16]) is a domain-specific modeling language developed with the aim to provide an easy to use framework for the specification of executable NEST simulator models [GD07]. Since its introduction in the year 2012, many concepts and requirements were integrated into the existing toolchain, while the programming language Java as the underlying platform remained almost untouched, making maintenance and extension of the framework by neuroscientists a disproportionately complex and costly process. This circumstance contradicts the basic principle of NestML, namely to provide a modular and easy to extend modeling language for the neuroscientific domain.

More than 90% of the overall costs arising during the development and usage of software systems originate in the maintenance phase [MHDH13], a circumstance which makes foresighted planning and design of software systems a crucial part of a software's lifecycle. While the effects of errors and bad design in programming in the small can be mostly mitigated by using appropriate concepts, e.g., data abstraction and modularization, wrongheaded decisions concerning the overall architecture or platform make the software's operation costly in the long term and affect the development over its whole life cycle [Wes02]. Here, **reengineering** and especially the changing of the environment or platform of the existing systems is the approach of choice given the fact, that systems often use no longer supported components, contain errors in the overall foundation or simply do not correspond to the existing requirements.

This thesis deals with the reengineering of the NestML tools collection and its migration to Python [van95] as a new target platform. Given Python's popularity in the neuroscientific domain, a migration benefits the usability as well as integration into existing systems, facilitates extensions by neuroscientists and makes usage of bridge technologies unnecessary. In order to accelerate the development and ensure modularity as well as maintainability of the reengineered software, the **MontiCore Language Workbench** [KRV10] will be used and extended by Python as a new target platform for code generation.

Contents

Chapter 1

Introduction

The brain is by far the most complex part of the human body [Nol02]. With approximatively 10^9 - 10^{12} neurons [HH09] it defines humans' behavior and consciousness as well as the perception of the environment. Although its capability to repair damages and anomalies to a certain degree by itself [KSJ$^+$00], external involvement is still often required to prevent natural processes such as Alzheimer or repair damaged tissue. Given the complexity and especially the size of the overall structure, a fundamental and in-detail understanding of processes and structures is essential to conduct a correct treatment. While first approaches to gain the required insights were focused on the extraction of samples, making experiments on living test subjects necessary, new approaches as emerged in the last decades since the introduction of computer systems made this kind of experiments partially obsolete. With computational science as the third pillar of science (besides theoretical and experimental science, [RBF$^+$05]), the behavior and structure of organs and complex systems can be simulated without the need for extraction of tissue or other involvements of living test subjects. Especially the discipline of computational neuroscience was able to gain many insights by simulating the brain of living organisms [DA01].

The overall complexity of the brain, as well as the sheer number of neurons with their interconnections via synapses make efficient simulations and usage of resources necessary to gain insights into such a complex system. Over the years many simulators and simulation environments were developed, from stand-alone solutions with an easy to use interface such as *Neuron* [Car07], through to programming language libraries with a more clear focus on performance and extensibility. The *Neural Simulation Tool* (NEST, [GMP13]) represents such a library implemented in C++ and is currently under development as a part of the EU's *Human Brain Project* [AEM$^+$16]. However, given its underlying platform, a fundamental understanding of programming languages is required to create new simulations and models or extend the existing behavior with new details. Concrete neuron models have therefore to be provided as implementations in C++. Here, *Domain-Specific Modeling Languages* (DSLs, [VDKV00]) have been established as a possible approach to abstract from programming language-specific concepts and focus solely on domain-relevant details. By hiding complex routines on the model in an easy to use framework, DSLs enable the user to use syntax and concepts common to the corresponding domain in order to define arbitrary specifications. The task of deriving equivalent models in the target programming language or environment is delegated to a set of tools, an approach which prevents error-prone and costly transformations by hand.

The *Nest Modeling Language* (NestML, [PRB$^+$16]) was developed with the aim to support the creation of models for the NEST simulator and therefore to facilitate the interaction of neuroscientists with the simulation environment. Given its clear intent, NestML es-

```
1 neuron izhikevich_neuron:
2
3     initial_values:
4         # Membrane potential in mV
5         V mV = -65 mV
6         U_m real = 0
7     end
8
9     equations:
10        V_m' = ( 0.04 * V_m * V_m / mV + 5.0 * V_m +
11               ( 140 - U_m ) * mV + ( (currents + I_e) * GOhm ) ) / ms
12        U_m' = a*(b*V_m-U_m * mV) / (mV*ms)
13    end
14
15    parameters:
16        a real = 0.02
17        b real = 0.2
18        c mV = -65 mV
19        d real = 8.0
20        I_e pA = 0 pA
21        V_min mV = -inf * mV
22    end
23

24    input:
25        spikes mV <- spike
26        currents <- current
27    end
28
29    output: spike
30
31    update:
32        integrate_odes()
33
34        # Add synaptic current
35        V_m += spikes
36
37        V_m = (V_m < V_min)? V_min : V_m
38
39        # threshold crossing
40        if V_m >= 30mV:
41            V_m = c
42            U_m += d
43            emit_spike()
44        end
45    end
46 end
```

usage of typical neuroscientific physical units

initial values for differential equations

differential equations describing the time dependent development

modifiable parameters of the neuron

ports to the neuron

*dynamic behavior of the neuron*

NestML

Figure 1.1: The *Izhikevich Integrate-and-Fire Neuron* model in NestML syntax: The model consists of two major components. The first part describes the static properties of the neuron, including all **initial values** and **parameters** such as thresholds and boundaries. The in- and out-ports of the neuron are defined in the **input** and **output** block. To describe the neuron's development over time, a set of differential **equations** is provided. Each equation states the development of a variable with respect to time and other parameters. The dynamic behavior of the overall neuron is stated as a set of imperative **update** instructions executed in each observable time step.

tablishes a syntax similar to the programming language Python [van95] as often applied in the computational neuroscience [MBD+15], making the reading and writing of models easier while requiring a shorter training period to fully understand all details. In its current state of development, NestML provides all concepts as required for the modeling of *Spiking Point Neurons* [And03]. Modeling of such components is especially interesting given their analogous behavior to real neurons and the inclusion of time as a leading factor. Figure 1.1 demonstrates a model of the *Izhikevich Integrate-and-Fire* neuron [Izh03] declared in NestML syntax.

  Manual creation of software tends to be error-prone and requires an in-depth review process to ensure correctness and efficiency [Wes02]. Moreover, whenever a modification to the DSL and therefore the underlying language is implemented, all components which depend on those specifications have to be rewritten. In order to avoid these problems the concept of *Generative Software Engineering* (GSE, [Rum17, CEC00]) can be employed. This principle represents a more refined approach to an engineering process and requires an abstraction of the existing system to a model describing the very same. Such a *system model* [Rum17] represents a given system in part or in total as a blueprint which can be easily adjusted to new requirements. A set of corresponding tools can then be used to generate a platform-specific code and documentation, making the model the central point of maintenance. Domain-specific modeling languages such as NestML utilize a grammar of the represented language as a blueprint for the generation of code. Adjustments to the represented language are therefore directly integrated into the formal specification of the modeling language, while a set of tools is used to automatically derive the corresponding code base. However, while individual tools generate code which is applicable by itself,

often the problem of incompatible interfaces of used components arises. Generated components of a DSL have to be adjusted by hand in order to be integrated into an existing system - a proceeding which contradicts the basic idea of generative software engineering. To solve this problem the concept of a *Language Workbench* [SSV⁺a] is employed. Instead of generating components such as lexers and parsers by individual tools, a language workbench represents a framework which combines all required tools and processes in a single entity. Moreover, the interfaces of all generated components are automatically adjusted to support integration and interaction, avoiding a modification by hand and enabling an out-of-the-box usage. A language workbench can therefore be used to ease the development of a language even further, by integrating all required tools in a single system. In order to apply all these principles, the state-of-the-art framework *MontiCore* [KRV10] was used for the engineering of the initial NestML framework.

Despite the high level of applicability of NestML and the quality of the overall software system, there exists a major drawback in the available implementation: Currently, MontiCore supports only the programming language *Java* [AGH00] as a target for the generation of all required DSL components, making Java the underlying platform of NestML and its supporting toolchain. However, Java is not prevalent in the computational neuroscience [MBD⁺15], a circumstance which results in many disadvantages during its usage. On the one hand, it prevents neuroscientists from interacting with the existing implementation of NestML in a white-box manner, making individual modifications and extensions a task for stakeholders outside of the domain. On the other hand, it also complicates the integration of the framework into existing ecosystems. Additional bridge technologies such as *Jython* [JBW⁺10] have to be used to enable interactions with existing tools, a solution which is only applicable or desired in a limited set of cases. All this leads to a situation where NestML has to depend on additional tools to enable an easy installation and usage. In order to deliver all required dependencies, subsystems, and platforms in a single image the virtualization tool *Docker* [doc17] was employed. In addition to enabling the execution on an arbitrary system, it also results in disproportionately many tools and technologies required just in order to provide a compact and easy installation. However, the usage of virtualization tools solves only the problem of required dependencies and platforms. The underlying problem of complex or impossible integration with *PyNN* [DBE⁺08], *PyNEST* [EHM⁺09], *SymPy* [JČMG12] and other tools written in common neuroscientific languages is still prevalent.

Here, the reengineering of an existing system and a migration to a new platform has been established as a possible solution to overcome the above-mentioned problems. While resulting in additional initial effort in order to migrate the existing code base to a new platform, a reengineering of the system represents an investment which pays off in the long run. Without the need for bridge technologies and additional tools, the software becomes easier to maintain, while adjustments and extensions can be directly integrated without depending on platform external components. Moreover, by selecting a platform according to the target audience of the software, here it is possible to involve domain experts into the future development process.

In the case of NestML, the programming language Python [van95] was selected as a new platform for the existing software systems. Although the computational neuroscience domain and especially its software frameworks are built upon several platforms, amongst others the computing environment *MATLAB* [MAT17] and the general purpose programming language C++, a brief survey revealed that Python is able to provide a good trade-off between simplicity and expressiveness as required to migrate the existing toolchain. While MATLAB does not provide several of the required components, including tools for a pro-

cessing of textual models, C++ represents a rich and powerful programming language which requires a certain level of expertise for maintenance and extension of the code base. Here, Python includes all required components and tools while persevering an easy to read syntax and structuring of the software, concepts which benefit an easy to maintain framework. Moreover, several of the components as integrated into NestML are only available in Python, including the symbolic mathematics library SymPy, making Python the language of choice for NestML.

A first, naive approach is the migration of the existing code base by means of completely handwritten code. However, MontiCore enables the generation of all required components, from the model-processing frontend consisting amongst other of a *lexer* and *parser* [Lou], through to subsystems which ensure partial semantical correctness of provided models by means of *symbols tables* and *context conditions* [HMSNR15]. Therefore, to avoid error-prone manual writing of code and ensure an accelerated reengineering process of NestML by means of existing technologies, MontiCore has to be extended and provided with all required modifications to support Python as a new platform for a component generation.

While a sole platform migration of NestML represents a valid approach by modifying the internal structures without altering the external behavior, here it can also be beneficial to re-evaluate the architecture and recollect all requirements to the existing system. Many components, as integrated into the initial implementation, may no longer be required. Other components can be replaced by more efficient or easy to maintain solutions. In conclusion, the process of a technology migration can be interleaved with a refactoring of the existing concepts. In the case of NestML, a reengineering process also requires the reevaluation of existing supporting tools and their applicability in the given situation.

1.1 Research Question

The overall research question of this work is:

> *How can MontiCore be adapted to support a new target platform for component generation exemplified on the use case of NestML?*

The generative approach can only be used up to a certain point, at which problem-specific concepts can no longer be completely specified by a model. The goal is, therefore, to determine at which point a manual implementation has to be used instead of adjusting and parameterizing existing tools. Here, especially the question of the trade-off between parametrization effort and overall reduction of manual work by generating the corresponding elements is important. This work will therefore demonstrate which Python-specific DSL components can be generated by MontiCore, and which parts of the software have to be written by hand. This trade-off becomes even more significant if we consider a possible future life cycle of the software and the intention behind its development. Software which is developed in a joint effort with domain experts or a community is expected to be changed often and adjusted to new requirements. Here, an automated generation of components can be beneficial, making all modifications focused on the design level.

The main contributions of this thesis are:

- How can a reengineering process of a DSL be conducted, exemplified on the concrete use case of the neuroscientific modeling language NestML.

- How the MontiCore Language Workbench can be extended to support a new target platform for component generation.

- Which components of an existing software should be re-generated by the adapted tools and which elements should be rewritten by hand.

1.2 Structure of the Thesis

The remainder of this thesis is structured as follows:

Chapter 1 introduces the overall context of the work, the difficulties and the intended solution.

Chapter 2 provides a concise introduction to the concept of Domain-Specific Modeling Languages (DSL) and a set of components reusable during their construction.

Chapter 3 shows how the model-processing frontend of NestML has been refactored and migrated to Python as a new platform.

Chapter 4 focuses on the NEST code generator and demonstrates how this component was reengineered.

Chapter 5 illustrates, based on a hypothetical use case, how the reengineered framework can be adjusted to new requirements.

Chapter 6 shows how the MontiCore Language Workbench can be extended to support a new target platform for code generation, thus accelerating the integration of new requirements into NestML.

Chapter 7 provides an introduction to the usage of the reimplemented framework.

Chapter 8 concludes the thesis with a summary and an outlook to future work. The core aspects of the thesis are once more discussed.

Chapter 2

Fundamentals

The concepts of refactoring and reengineering require an in-depth understanding of a software's functionality. Here, the key principle is to preserve the outer behavior, while the inner structure of the system is modified to adhere to higher quality standards or other goals. In the case of a domain-specific modeling language, the overall architecture of the processing tools was silently standardized to a component-based and non-monolithic shape. This architecture, as well as the general approach to how a complex DSL-processing framework can be decomposed into a set of individual components and subsystems, will therefore be presented in section 2.1. Subsequently, section 2.2 will introduce a set of basic reengineering concepts and evaluate which supporting components can be used in the use case of NestML.

2.1 Domain-Specific Modeling Languages

A *Domain-Specific Modeling Language* (DSL, [Fow10]) is *a specification language which provides appropriate concepts and notations for the modeling of problems in a specialized domain* [VDKV00]. In contrast to *General-Purpose Languages* (GPL, e.g., UML [Rum11, CE97]), which can be used in a wide range of application fields, but do not provide specific notations and concepts as required for the modeling of very specific problems, DSLs are designed and designated for a specialized task where an in-detail specification of the problem and all its domain-specific concepts is required out of the box. Physical units are a common example of a concept which has to be supported by a modeling language in the field of neuroscience. Although possible, an extension of a general-purpose modeling language by such a specific concept would contradict the overall idea of a GPL, namely to provide an abstract and domain-unspecific concept for the modeling of problems.

  Specifications of problems in a given DSL are often not directly executable but rather of a declarative nature. Consequently, such declarations represent the initial input to a set of tools which process the given model and generate a refined, executable representation. This *DSL-processing workflow* consists of several steps as illustrated in Figure 2.1. Here, one of the key advantages is the use of general-purpose programming languages as a common interface for the generation of code. Generating directives which can be directly executed by the CPU is a process which requires a highly specialized knowledge of soft- and hardware concepts. In order to avoid such a processing of a model, a GPL is instead selected as a target for code generation. On the one hand, the source code as generated from textual models can then be further processed by existing tools as available for the respective programming languages, e.g., editors, verifiers or solvers. On the other
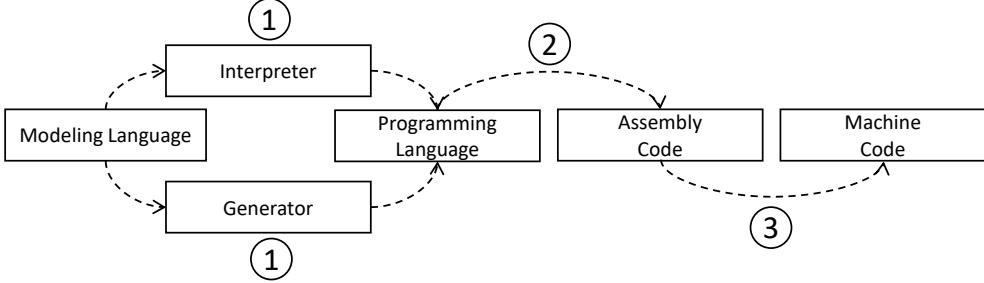
Figure 2.1: An overview of the processing workflow: A model stated in the source modeling language is not executable and has to be processed by a generator or interpreter to compilable code (1). This code represents an equivalent specification in a programming language, e.g., C++ [Sch98]. Subsequently, a compiler generates from the given code a corresponding set of atomic *assembler instructions* (2) [Dun11]. Those instructions are finally processed to an equivalent set of binary coded CPU instructions, representing an executable form of the initial model.

hand, the concern of low-level optimizations, e.g., memory access or handling of loops, is delegated to the platform-specific compilers. Here, the clear single responsibility of the DSL-processing framework is established: For a given textual model declared in the DSL-specific syntax, generate an equivalent and imperative model in a general-purpose programming language. Optimize only domain-specific concepts, e.g., simplify physical units, and leave the remaining parts to the available general-purpose infrastructures.

After a model has been processed by the toolchain, the output can be used together with other existing tools and frameworks for further analysis and optimization, or just compiled to executable code on the corresponding platform. The key concept here is a clear separation of problem-specific frameworks and general-purpose tools. This enables the development of DSLs for arbitrary use cases, where only certain parts of the overall process of making models executable have to be created or modified, while the remaining workflow is used in a black-box manner. The focus, therefore, lies on a DSL-specific collection of tools as required to generate, for a given model in the DSL, an equivalent model in a programming language. However, the processing of a given model requires a sequence of complex steps, from the parsing of a model to an internal computer-processable data structure, though to the generation of code for specific target platforms, e.g., simulators. Here, the decomposition of the monolithic structure of the DSL-processing software into subsystems and assisting components is a common approach for the engineering of a DSL, where the individual complexity of each element becomes accessible as depicted in Figure 2.2.

Before the actual processing of a given model can be executed, the input to the workflow has to be specified. Using the syntax of the modeling language, the problem's specification is denoted with all required details as necessary for an unambiguous definition. Such a declaration demonstrates the key advantages of using a DSL: The specialized syntax is focused on the modeling of specific problems and leaves out information which is known to the computer or can be derived from the context. Moreover, in contrast to keywords and markers as often used in programming languages and general-purpose modeling languages, here, all keywords originate from the domain itself, making the concepts they denote easy to comprehend. Users from the respective domains are therefore directly able to interact with the modeling language without having to have experience with programming languages. Figure 2.3 illustrates a simple yet complete model in the *Calculator Modeling*
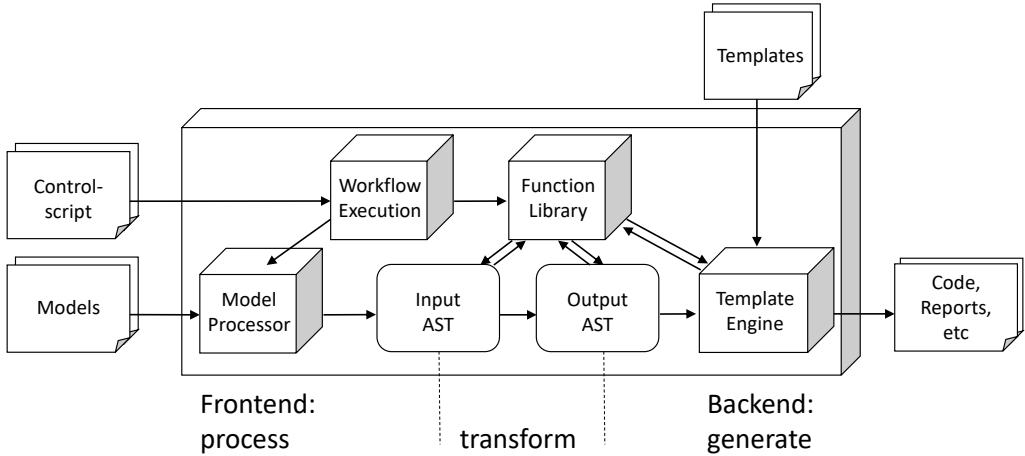
Figure 2.2: The architecture of a DSL [RH17]: The model-processing toolchain consists of three major subsystems and several assisting components. A given model is handed over to the *model-processing frontend* which parses it and creates an internal representation, the *Abstract Syntax Tree* (AST). This representation is then further analyzed and refined by the *transformation and function library*, a collection of components which ensure the overall correctness of the given model and employ subroutines for further modifications and transformations, generating an *output AST*. The processed AST is finally handed over to the *backend* subsystem which generates code in a format as specified in a set of *templates*. The overall process is orchestrated by a *workflow execution* unit whose behavior and individual steps can be customized by a *control script*. The result of the overall process is a set of generated *code*, *reports* and other artifacts.

Language (CML). Although a graphical representation of a model can also be made possible, cf. UML [Rum11], in almost all cases the underlying structure of a stored model is represented in a textual form.

A given model represents the input to the overall processing workflow. In order to create a computer-processable data structure of a textual model, the *workflow execution unit* delegates the handed over model to the *model-processing frontend*. This subsystem consists of several major components:

- A *Grammar* [Lou] which specifies how syntactically correct models are constructed.

- A *Lexer* [Lou] which reads a given model as a stream of characters and generates a stream of token objects, i.e., units which represent individual words in a model.

- A *Parser* [Lou] which consumes a generated token stream and creates an initial internal representation, a *Parse Tree* [Lou], according to the grammar.

- A collection of *Visitors* [Gam95] which analyze a given parse tree and retrieve information as required for further processing. This information can be optionally stored in a refined representation, the *Abstract Syntax Tree* (AST, [Par09]).

Each component is designed and constructed individually, decomposing the model-processing frontend's outer behavior into several, easily distinguishable steps. A grammar is hereby used as the starting point for the construction of a DSL. Denoting the syntax of a modeling language, this component has to be designed and implemented with possible future modifications and adjustments in mind. It is therefore advisable to employ the same standards

9

```
1  calculator test_calculation:
2    declaration:
3      a = 10
4    end
5
6    computation:
7      a = (a**2)/(a-7)
8    end
9  end
```

Calculator Modeling Language → CML

domain-specific keywords

unnecessary details left out

coherent structuring
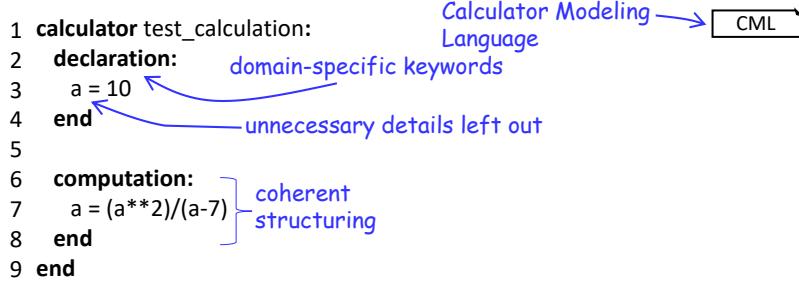
Figure 2.3: A model in the *Calculator Modeling Language* (CML): Each set of calculations is introduced in the *calculator* block and consists of a *declarative* part and a *computational* part. Details, which are not required for an unambiguous declaration, e.g., the type of a variable, are left out. Keywords of the language are selected according to their respective domain, making the purpose of their usage clear.

as known from programming languages, namely modularity and simplicity. A grammar can be composed of arbitrarily many sub-grammars distributed across several artifacts. Here, modularity can be achieved by composing the core grammar from general-purpose sub-grammars, e.g., a grammar defining mathematical expressions and a control structure grammar. This leads to a situation where new languages can be created by means of the *Building Block* approach [M+03], where only details have to be adjusted.

Each grammar is introduced by a set of *Token Definitions*, i.e., words which the declared language expects to interact with in a concrete model. A token declaration defines the sequence of characters, which, if it is detected in a given model, is interpreted as a concrete instance of the corresponding token. Tokens can be defined explicitly, i.e., the sequence of characters which has to match is given in the grammar, or implicitly where a regular expression denotes the properties a word has to have to be recognized as a token of a certain type.

Tokens represent the fundamental units of a grammar and are used in the second type of declarations, namely the *Grammar Rules*. Each rule states how tokens and sub-rules can be combined to create syntactically valid sentences and therefore models in the language. A sequence of tokens is hereby recognized as being constructed according to a given rule if the order, as well as the type of the read-in tokens, correspond to the declaration. Figure 2.4 illustrates the definition of tokens and grammar rules in Antlr syntax [PLW+00] as used for the declaration of the Calculator Modeling Language.

The grammar of a modeling language represents a formal specification constructed according to a grammar definition language, making it an ideal candidate for the input of supporting tools. Those tools can then be used to generate components whose structure and behavior can be completely derived from a given grammar, namely the *Lexer* and *Parser*. Given the strict and unambiguous definition of tokens and grammar rules, a set of supporting tools, e.g., *Antlr* [PLW+00], is able to generate a lexer which reads in a model and decomposes it into a stream of tokens. The task of the lexer is, therefore, to iterate over a given model and hand over a new token of a specified type and value whenever the corresponding definition has been detected. A parser consumes this stream of tokens, identifies a suitable production and constructs an initial internal representation of this rule. Here, most often a tree-like structure, the *parse tree*, is utilized to preserve the structure of the initial model. Figure 2.5 demonstrates how a given model is decomposed into token instances by the lexer, analyzed by the parser according to the grammar rules and finally reconstructed as a parse tree. Each token stores, among other details, the

```
1 grammar CVL;
2
3 Number: [0-9]+;
4
5 PlusLiteral: '+';
6
7 plusExpression: Number PlusLiteral Number;
8 ...
```

Antlr4 grammar → Antlr
implicitly defined token as a valid set
at least one digit
explicitly defined token
a grammar rule for the addition of two values
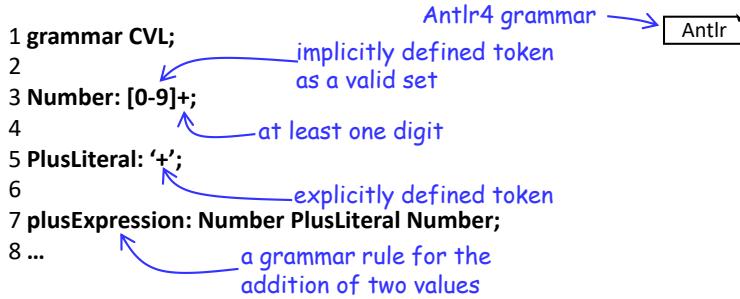
Figure 2.4: An excerpt from the CML Grammar: Utilizing the syntax of Antlr [PLW+00], the grammar denotes explicit and implicit definitions of tokens and their subsequent usage in a grammar rule to define the addition of two values. In order to distinguish tokens and grammar rules, the former are introduced with a capital letter, the latter with a lowercase letter.

CML
...
10 + 2
...

Lexer

token stream

{Number, PlusLiteral, Number}

Parse Tree

plus character token → P

N   N

number token

Grammar Rules
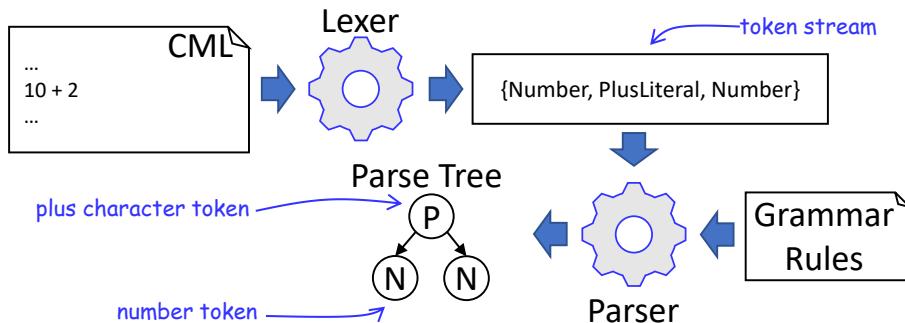
Parser

Figure 2.5: From a model to the parse tree: A model stored in a file is read in by the lexer and decomposed into a stream of tokens; each token representing an atomic unit as defined by the language's grammar and contained in the model. This token stream is handed over to the parser which inspects a given set of grammar rules and selects, if available, an appropriate rule. According to the selected rule, the tokens are rearranged into a tree structure, the parse tree.

actual text as read-in from the model as well as its source position, two properties which are helpful whenever troubleshooting of models is intended.

A parse tree represents an elementary approach for the storage of models in a directly computer-processable structure. However, often additional information or operations need to be stored together with the nodes in the tree, making the immutable structure of the parse tree inapplicable for further processing. In this case an intermediate form of the model is derived, the *Abstract Syntax Tree* (AST). While a given node in the parse tree is represented by a token object as generated during the parsing of a model, a node in the AST is a data structure individually designed by hand. Each type of an AST node is represented by a class which stores arbitrary information and provides additional methods for required interactions, e.g., data retrieval. All details required to create a node in the AST are derived by means of visitors, which implement a traversal routine as well as the corresponding information collecting operations on the parse tree. Although the creation of an AST can be implemented as a part of the parsing process, cf. MontiCore [KRV10], a clear separation of the model-parsing and AST-creating routines can be beneficial in terms of maintainability and modularity, making the traversal and information derivation process on the parse tree a modular and exchangeable component. Figure 2.6 outlines
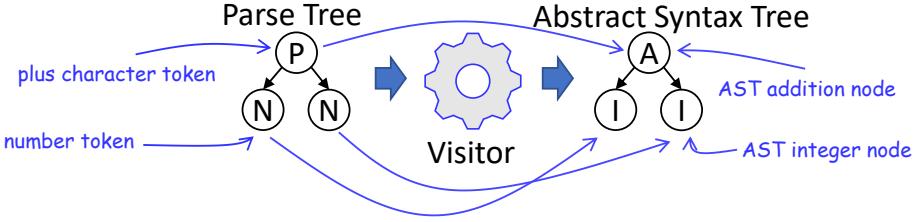
Figure 2.6: The construction of an AST: A *parse tree* represents an immutable data structure with a limited set of stored information. In order to store more model-related details, an *abstract syntax tree* is initialized by utilizing a *visitor* which traverse a parse tree and retrieves all required details.

how a visitor is used to create an AST from a given parse tree. The abstract syntax tree serves as the final representation of the textual model before being further processed by the next subsystem of the DSL-processing tools collection.

Here, an encapsulation of the model-processing frontend as a single unit benefits the overall maintainability and re-usability of the toolchain. The task of this subsystem is solely to create a modifiable internal representation of the model. The concrete checks and modifications can then be implemented in other components. Moreover, all modifications concerning the grammar of the language are located in a single component, making them transparent to other subsystems. Here, information hiding is achieved by utilizing the AST creating routine as the interface to the frontend.

Up to now the lexer and parser have ensured syntactical correctness of a processed model, since otherwise either an invalid token or no corresponding grammar rule would have been found. However, for the sake of an overall correctness of a given model, also the semantical soundness has to be ensured. The concept of soundness is hereby represented by a broad set of rules which have to be fulfilled by a given model. Starting with properties which always have to hold, e.g., that all used variables are defined, through to more complex and domain-specific properties, e.g., that no assignments are made to read-only ports, it is possible to specify a list of situations where no correctness of generated results can be guaranteed. This idea originates from the concept of *Design by Contract* [Mey02], where models which fulfill a set of pre-conditions are guaranteed to be processed to correct counter pieces which fulfill a set of post-conditions, e.g., a correct syntactical structure on the target platform. In order to ensure those properties, a so-called *Symbol Table* [HMSNR15, Fow10] and a set of *Context Conditions* [Rum17] are implemented. The concepts as implemented in these components provide a possible solution to the context-insensitive nature of *Context Free Grammars* [BW84], a class of grammars and grammar specification languages where rules are not able to take their context, i.e., neighboring elements, into account.

The symbol table serves as a central unit, storing all context-related information of a model. All details which are implicitly contained in the AST are made explicit by analyzing the tree and creating for each declared element a *symbol* [Fow10], and for each block of statements a *scope* [Fow10] with arbitrarily many symbols and sub-scopes. By storing this information in a tree or map-like structure, it can be easily derived which elements have been defined in which scopes, which properties the declared elements have, and in conclusion which conflicts exist in the initial model. Figure 2.7 illustrates how a symbol table is constructed from a textual model.

Although some context-related properties of a given model, e.g., the question whether elements have been redeclared, can be checked during the construction of the symbol table, other require an overall complete symbol table to ensure a correct context. Representing an
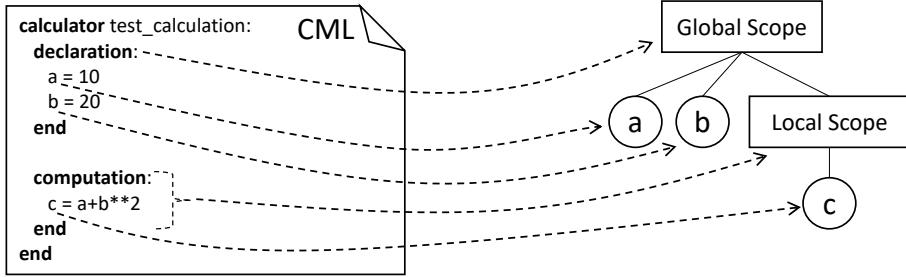
Figure 2.7: Construction of a symbol table: The *declaration* block defines a global scope for variables. All variables defined in this scope are therefore available in all sub-scopes. The *computation* scope defines a local scope as used to compute new values. A local scope with its defined variables is embedded in the global one.

individual component, a set of context conditions is used to detect broken contexts or warn the user of potential mistakes. Here, domain- and task-related rules can be implemented to restrict the set of semantically correct models. Context conditions are implemented by defining two properties: The type of elements in a model whose correctness should be checked, and an individual definition of correctness in the corresponding context. For instance, a context condition could traverse a given abstract syntax tree, and whenever an expression has been detected, check according to the previously created symbol table if all referenced variables have been defined.

Having an in terms of syntax and the specified context conditions correct model, the second component as contained in the transformation and function library can be executed to restructure the model to a different form. Arbitrary and use case-specific processes and transformations can be implemented in order to modify the internal representation according to the given requirements. For instance, in the context of CML a transformation substitutes redundant computations of the same value by a reference to a single intermediate variable containing this value, making the computation required to solve the problem even faster. This process is enabled by the mutable structure of the AST which can be easily adjusted or extended, demonstrating the advantage of using an AST over the immutable parse tree. All parts of the AST can be replaced or removed, while new elements can be easily attached. However, due to its complexity, most often an automated generation of the transformation is not possible. In conclusion, all transformations on the model have to be implemented completely by hand, making a manual review process as well as involvement of domain experts mandatory.

A correct and, in some cases, optimized and extended AST represents the final instance before being generated to a target format. The approach to generate an output can be classified according to certain specifications [Sch17], namely whether its generation has been conducted by using a meta model or not. In this context, the meta model is one or more artifacts specifying the syntax of the target platform. In the case of DSLs, most often a programming language is selected as the target. However, modeling of the target by means of a meta model unnecessarily complicates the process, considering that only a limited set of concepts of the target platform is utilized, making many parts of the meta model unnecessary. Thus, in the case of NestML, a simple *model to text* [Sch17] approach is used, where the target is generated without using a meta model as an assisting component. The most common approach to generate a target-specific and persistent representation of
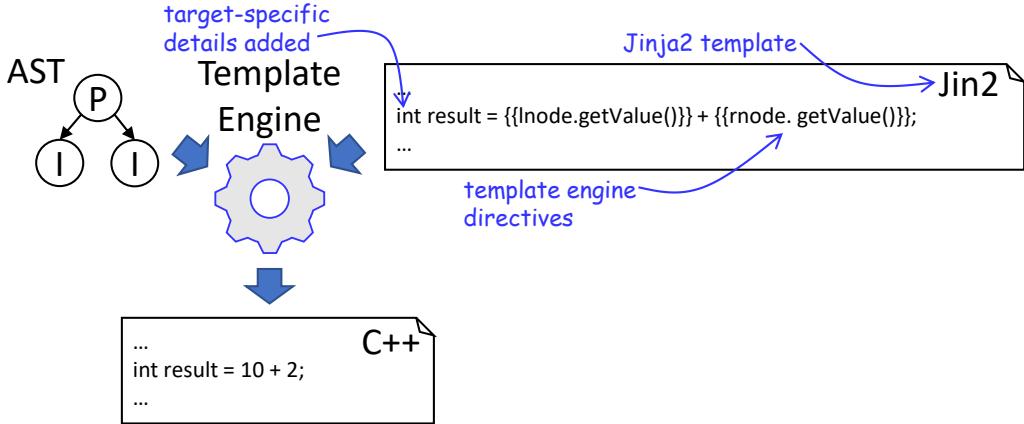
Figure 2.8: The code-generating Backend: In order to make modifications to the internal representation persistent and store the results in a required format, a *Generator Engine*, e.g., *Jinja2* [Ron08], is employed. This component iterates over a *template* written in the respective declaration language and replaces placeholders by details as contained in the AST. Embedded directives to the generator engine are executed and their result used, while the remaining part is simply copied. The result is a model-specific file adapted to the required target platform.

the modified AST is the usage of a *Generator Engine*[1][GBR04]. Utilizing a *template*, this component inspects a handed over AST and replaces placeholders and directives with the corresponding elements from the AST, making this approach ideal for use cases where the output can be schematically described. Figure 2.8 illustrates an excerpt from the template as used to generate models of the calculator modeling language to compilable C++ code. As with the model-processing frontend, it can be beneficial to encapsulate a target platform-specific backend in a single subsystem. The interface to this subsystem is represented by the collection of methods used to generate the model in a specific format. By following this concept, it is easily possible to extend a given DSL-processing framework with new target platforms, where only the backend with its target-specific components has to be implemented.

The compilable code represents the overall result as generated by the backend. This code can now be used with conventional tools, compilers and editors to create the corresponding executable binaries. Figure 2.9 demonstrates the generated results. In conclusion, the task of a DSL toolchain is not simply to transform a model to a target-specific format, but also to enrich it by additional details, among others types and concrete operators.

Given the modular structure of the DSL-processing toolchain, it is easily possible to insert or delete steps in the processing of a given model. The workflow introduced above represents one possible implementation and may, therefore, differ in certain scenarios. In some cases, where correctness of models can be assumed, it is not required to utilize a symbol table and the corresponding context conditions. Whenever models do not have to be modified, it may be beneficial to directly interact with the parse tree instead of creating a modifiable AST. In situations where certain details of the target model cannot be derived by the toolchain, it is also possible to enable the injection of handwritten code. All these scenarios underline the necessity of a modular system rather than a monolithic structure of the toolchain.

[1]also referred to as *Template Engine* [Rum17]
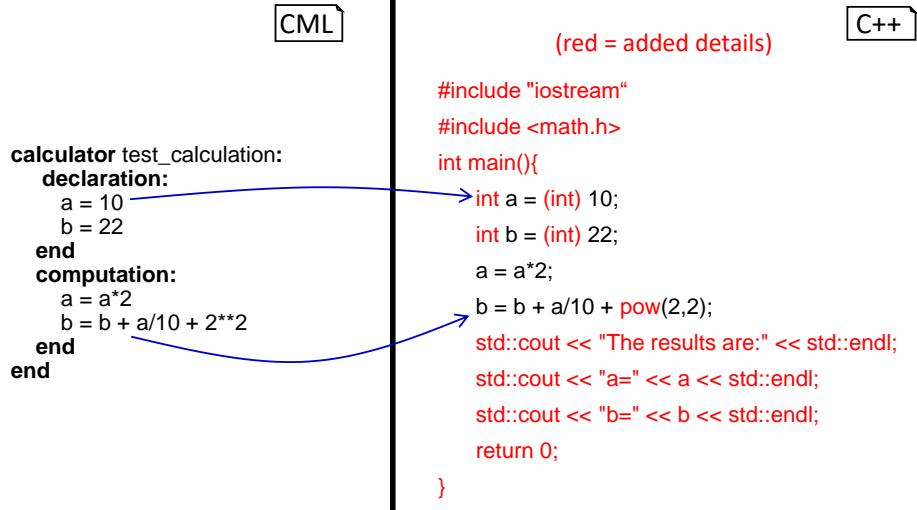
Figure 2.9: A comparison of models: The initial model in the *calculator modeling language* is processed by the DSL framework to compilable C++ code. In order to enable compilers to process the given model, the model is generated in target-specific format and enriched by required standard properties, e.g., brackets and libraries. The backend derives details regarding the type of the variables and enriches the generated model by respective specifications. Finally, instructions for a visualization of results are added.

In conclusion, while a black-box view on the DSL processing tools indicates a complex and monolithic system, a white-box view demonstrates that all individual components, starting from a lexer and parser, through to the generator engine, have an easy to implement task. Although the presented fundamentals demonstrate the general approach for construction of a DSL, a basic understanding of the presented concepts can also be beneficial whenever certain parts of an existing system have to be modified. All approaches, tools, and details as presented in this section will be utilized in chapter 3 in order to reengineer and reimplement the existing components of NestML on a new platform.

2.2 Methodology and Reuse of Components

The trend of rising size and complexity of software as observable since the introduction of the first computer systems [Hel96] makes research in the field of software reuse approaches a crucial activity in order to ensure the stability and correctness of products. Over the years many different techniques and possibilities to reuse existing concepts, code bases and components have been developed and tested. MontiCore, cf. chapter 6, demonstrates how components can be generated from a user-supplied system model and, therefore, implements the concept of *program generators* [Hem93]. *Design patterns* [VHJG95] on the architecture and implementation level represent a different approach to reuse existing knowledge during the creation of new software. Here, instead of providing concrete code which can be integrated into the project, a general idea is given how certain components have to be designed in order to achieve a higher level of maintainability and modularity. The pipelining of a read-in model as implemented in the overall DSL approach, cf. section 2.1, represents a modified instance of the *pipes and filter* pattern [BHS07], while all AST classes generated by MontiCore [RH17] contain an implementation of the *builder* pattern [BHS07]. The utilization of patterns represents a reuse approach which has to

```python
1  def p_expression(p):
2      """
3      astExpression : astExpression PLUS astExpression
4                    | number;
5      """
6      if type(p[1]) == int:
7          p[0] = ASTNumericLiteral. makeLiteral(p[1])
8      elif len(p) == 3 and p[2] == ‚+':
9          p[0] = ASTExpr.makeTerm(p[1], p[3])
10         p[0].isUnaryPlus = True
11
```

grammar rules defined as comments

Python

processing according to the grammar in comments

AST creation during parsing

Figure 2.10: Definition of a grammar rule in *Ply* [pyt17b]: Each rule is defined as a function in Python syntax. A definition of the rule is stated in BNF form [MR] in the comment section of the function. The corresponding body defines the routine on elements as executed whenever a stream of tokens is detected to be constructed according to the grammar rule. Here, the grammar as stated in the comment is inspected by the method and details are retrieved.

be regarded independently of the concrete domain and use case, making it a technique which should always be applied during the engineering of software. The remaining part of this chapter will, therefore, focus on more specific techniques, approaches, and tools which can be employed to accelerate and ensure certain quality standards of the reengineered NestML framework as introduced in chapter 3.

Chapter 1 outlined how MontiCore can be used to generate a set of components representing the overall infrastructure required to create a model-processing frontend. Here, no additional tools and libraries are required to be able to process a given model to the corresponding internal representation. In contrast to the generated AST classes which are completely independent of concrete tools, the generated lexer and parser require the Antlr runtime environment [PLW+00], making it a hard dependency whenever the generated code shall be used. Besides MontiCore and in consequence Antlr, many other lexer and parser generators can be employed. The Calculator Modeling Language (CML) as introduced in section 2.1 has been used to conduct a brief survey on the usability of other tools which can be applied to reuse existing technology. For this purpose, two additional lexer and parser generators were tested for their applicability.

The *Ply* [pyt17b] lexer and parser generator represents a Python implementation of the *Lex and Yacc* [LMB92] tools. Here, the overall grammar of a language is defined as Python code and is therefore not contained in a separate artifact. Moreover, each token has to be provided with a definition and an executable function in order to achieve a specific processing. Figure 2.10 visualizes a single definition of a rule as used to enable the parser to recognize an addition of two values. The grammar rule, as well as the corresponding processing, is defined in a single method. The use case of CML has shown that while Ply provides a highly customizable behavior where many components can be individually adapted to specific needs, it also requires an in-depth understanding of the general lexer and parser concepts to apply all required principles correctly. Moreover, given the monolithic definition of the grammar, where all components are defined in a single Python artifact, this approach prevents an easily maintainable structure of the software where elements can be adjusted individually. Modularity of the components on the artifact level can therefore not be enforced.

```python
1 import pyparsing as pp
2
3 number = pp.Word(pp.nums)
4 expression = pp.Or([number, expression + pp.Literal(`+`) + expression])
5
```

definition of rules as function calls — Python

predefined token definitions

alternatives provided as lists

Figure 2.11: Definition of a grammar rule in *PyParsing* [McG07]: Each rule is defined as the return value of a function call. A set of predefined, common token definitions can be reused in order to reduce the overall grammar. Alternatives, optionality, and repetition is marked by a call to the corresponding function, e.g., the *Or* function call to declare a set of alternatives.

PyParsing [McG07] is yet a different tool which was tested for its applicability. Here, grammar and parser rules are defined as Python code and consist of calls to individual functions representing token definitions and sub-rules. Each rule corresponds to the return value of a specific function. Figure 2.11 illustrates the definition of a rule as used to the define addition of two values. This approach represents a valid alternative to the definition of a grammar as a composition of rules and tokens in a specific syntax and as a separate artifact. Moreover, by using calls to predefined functions, no user-defined routines have to be written, making a composition of grammars and rules easy to achieve. Nonetheless, while this approach may be applicable in the case where small grammars represent the whole required syntax of a DSL, it becomes less maintainable and modifiable whenever a larger grammar with more rules and complex right-hand sides is required, making it less appropriate for modeling languages with complex structures. Moreover, several features as included in Antlr, e.g., additional channels for streams of tokens, are hard to implement and maintain in PyParsing. Token stream channels are used to filter out certain specification of the read-in model, e.g., white lines, by handing them over to a different stream. Although not crucial, such a feature is beneficial whenever a certain handling, e.g., processing of source model comments, has to be implemented.

In conclusion, we see that MontiCore and Antlr represent the best alternative for the generation of a lexer, a parser as well as the AST infrastructure. Besides providing an expressive concept for a definition of grammars, it also enables the user to directly influence the generated lexer and parser with handwritten code. Moreover, by encapsulating the definition in a separate file, a clear separation of concerns and single responsibility is achieved. Here, the grammar defines the *what*, while the generated lexer and parser the *how*, instead of mixing both concepts in a single artifact. Although Ply and PyParsing can be used to generate or implement a lexer and parser, these libraries do not feature a concept for the generation of additional components as required to store and interact with a read-in model, among others the corresponding AST data structure and visitors. MontiCore and its concept for the generation of components is therefore a clear favorite for the engineering of a DSL.

As discussed in section 2.1, most components required in the transformation and function library cannot be generated or reused and consequently have to be implemented by hand. In the case of NestML, only a subsystem for the processing and storage of physical units as often required in the computational neuroscience has been found to be applicable. For this purpose, two existing implementations were tested.

Sympy [JČMG12] and especially its *Physics* package represents a possible implementation

for the handling of physical units. Each unit is stored as an object consisting of several properties describing its structure. A physical unit is represented by a base, e.g. the electric potential in volt, and the corresponding magnitude definition. By encapsulating those properties in a single object and providing an overwritten behavior for the standard arithmetic operations, this module enables the calculation and derivation of new physical units, a concept which is often required whenever type checking of expressions is intended. However, one of the key drawbacks of using Sympy's physics module is the handling of equality checks for combined and complex units. While atomic parts of a given expression in a neuron model tend to have simple units, e.g., a single variable storing a value in millivolt, compound expressions often results in complex, combined units. The selected unit system should be able to derive new units by combining existing ones. However, while SymPy supports such a handling, the concept of equality is lost. Derived units such as *newton* and the combination of their base units $kg * m/s^2$ are not recognized as being equal. This circumstance prevents a valid type checking of expressions whenever physical units are involved.

The *AstroPy* [ART$^+$13] module implements a similar approach by storing units as composable objects. Besides providing a similar concept for arithmetic operations, it is also possible to define new physical systems, i.e., systems where a specified set of units is regarded as the base units. Moreover, each unit object stores an additional set of properties as often required during a type checking routine. Utilizing these properties, it is easily possible to derive for a given unit all available, equivalent representations. In conclusion, the aforementioned problem of inequality between *newton* and $kg*m/s^2$ is no longer given. Instead, the underlying base units of a compound one are compared, thus equality for arbitrary combinations can be ensured. The clear separation of the units and magnitudes enables a type checking system to regard units which only differ in a prefix as being equal. AstroPy and its underlying units type system are therefore the most fitting solution for processes where derivation of new units and equality checks represent the main goal.

The last component which can often be reused during the engineering of a DSL is a generator engine as employed in the code generator. Here, a vast amount of solutions exists, from light-weight and fast renderers for web pages, through to complex and expressive engines for the generation of compilable code. However, most often engines only differ in detail and provide a similar set of functionality. A brief survey revealed that all concepts required for the reengineering of NestML are available in almost all common generator engines. Here, *Cheetah* [che17], *Jinja2* [jin17] and *Tenjin* [ten17] were tested. As demonstrated in Figure 2.12, all three components utilize an almost equal syntax and instruction set. In conclusion, the reasons for deciding in favor of Jinja2 was the fact, that it provides the most in-detail documentation of the tool's usage and implementation. Moreover, Jinja2 is used in a wide range of projects, amongst others by *Mozilla* [moz17] and *SourceForge* [sou17], thus a long-term support of the tool is expected.

All components introduced in this chapter will be used in chapter 3 and chapter 4 in order to employ the reuse of existing components and follow the building block approach. However, we will also present an additional set of techniques and patterns as applied in the reengineered NestML framework.

```
1 #for $j in $getDecl()                                              Cheetah
2 int ${j.getName().prettyPrint()} = (int)${j.getExpr().prettyPrint()};
3 #end for
```

```
1 {% for j in getDecl() %}                                            Jinja
2 int {{ j.getName().prettyPrint()}} = (int){{ j.getExpr().prettyPrint()}};
3 {% endfor %}
```

```
1 <?py for j in i.getDecl(): ?>                                      Tenjin
2 int ${j.getName().prettyPrint()} = (int)${j.getExpr().prettyPrint()};
3 <?py #endfor ?>
```

Figure 2.12: A comparison of templates as used to generate CML models to compilable C++ code. Here, all three engines *Cheetah* [che17], *Jinja* [jin17] and *Tenjin* [ten17] utilize an almost equal syntax and provide a similar set of instructions. Details are only found in the corresponding implementation and execution time of the engines.

19

Chapter 3

The model-processing Frontend

The previous chapter introduced the overall architecture of a DSL, all required model-processing steps as well as a set of reusable components. These fundamentals will now be used to reengineer the existing NestML framework and perform a platform migration to Python. In this chapter we will demonstrate how the model-processing frontend has been reengeered. To this end, it is first necessary to parse a textual model to an internal representation by means of a *lexer* and *parser*. Section 3.1 introduces this subsystem together with a collection of AST classes and the *ASTBuilderVisitor*, a component which extracts an AST representation from a given *parse tree*. Subsequently, the *CommentCollectorVisitor* and its underlying process responsible for the extraction of comments from the source model and their correct storing in the AST is demonstrated, making the generation of self-documenting models possible. Having a model's AST, it remains to check its semantical correctness. For this purpose, section 3.2 will first introduce a data structure for storing of context-related details, namely the *Symbol* classes. Here, we also show how modeled data types can be represented and stored. In order to provide a basic set of constants and functions predefined in NestML, the *predefined* subsystem is implemented. With a parsed model stored in an AST and a structure for storing context information, the frontend proceeds to collect context details of the model. Demonstrated in section 3.3 together with the *SymbolTable* and a set of *context conditions*, the *ASTSymbolTableVisitor* ensures semantical correctness. After all context conditions have been checked, the frontend's model processing is complete. All steps outlined above are orchestrated by the *ModelParser* class which represents the interface to the model-processing frontend. The chapter is concluded in section 3.4 by an introduction to the set of assisting components. Figure 3.1 subsumes the concepts demonstrated in this chapter. In order to avoid ambiguity, we refer to the reengineered framework as *PyNestML*.

3.1   Lexer, Parser and AST classes

As introduced in section 2.1, the first step during the processing of a textual model is the creation of an internal representation by means of an AST. For this purpose, it is first necessary to implement a *lexer* and *parser* which read in a textual model and create a respective *parse tree*. However, the parse tree represents an immutable data structure where no data retrieval and modification operations are provided, making required transformations and interactions difficult. Consequently, a refined representation in the from of an AST has to be derived. It is therefore necessary to implement a collection of AST classes used to store individual elements of the AST. In order to retrieve all required in-
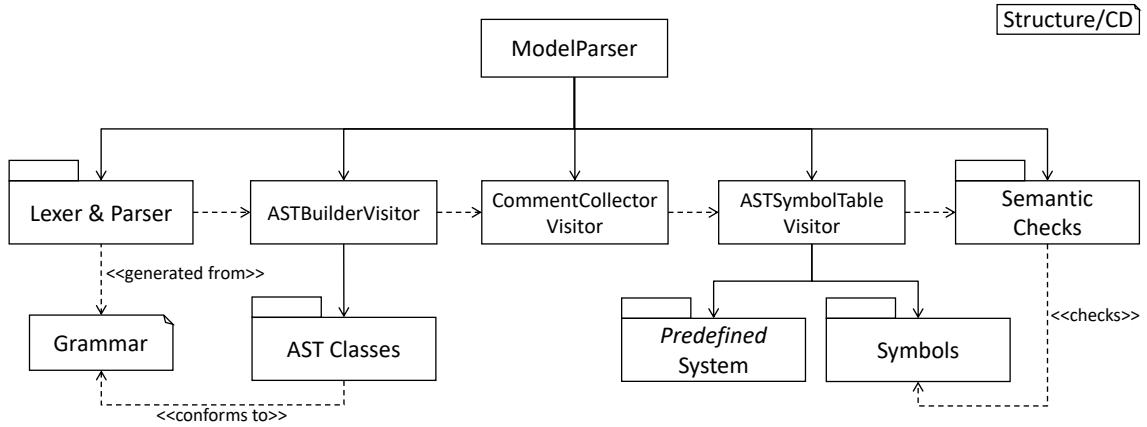
Figure 3.1: Overview of the model-processing Frontend: The *lexer* and *parser* process a textual model to the corresponding *parse tree* and can be completely generated from a grammar artifact. The *ASTBuilderVisitor* is responsible for the initialization of a model's AST, employing classes which conform to the DSL's grammar. After the AST has been constructed, the *CommentCollectorVisitor* collects and stores all comments stated in the source model. The *ASTSymbolTableVisitor* subsequently collects context information of the model by utilizing *Symbols* and the *predefined* subsystem. *Semantic Checks* conclude the processing by checking the model for semantical correctness. All steps are orchestrated by the *ModelParser*.

formation from the parse tree and instantiate a respective AST, the *ASTBuilderVisitor* is implemented. The result is a model's AST which can be used for further checks and modifications. All these steps are encapsulated in the orchestrating *ModelParser* class. Figure 3.2 provides an overview of the components as introduced in this section.

Although possible, *lexer* and *parser* are usually not implemented by hand but rather generated from their respective grammar. Section 2.1 demonstrated several existing tools and approaches for component generation. In the case of PyNestML, *Antlr* was selected to define the grammar and generate the lexer and parser. For this purpose, it is first necessary to create the grammar of the language. Fortunately, the grammar artifact as found in NestML can be completely reused and has only to be adapted. Appendix A demonstrates the reworked grammar as used in PyNestML. Although modular and easy to understand, PyNestML's grammar is still an artifact of several hundreds lines of code. In the following we will therefore use a simplified working example as depicted in Figure 3.3. The grammar is hereby an artifact structured according to Antlr's syntax and defines which rules and tokens the language accepts, cf. section 2.1. All concepts as introduced for the working example are implemented analogously for the complete grammar.

Starting from the grammar, Antlr is used to generate the respective lexer and parser, making an error-prone implementation by hand unnecessary. Consequently, these components can be used in a black-box manner, where only the interface is of interest. The generated lexer expects a file or string to parse, and returns the respective token stream. As Figure 3.8 illustrates, storing and interacting with the stream of tokens can be beneficial whenever a derivation of additional details in the initial model is required, e.g., the model comments. The token stream is handed over to the parser which creates a parse tree representation of the model according to the grammar rules. Both steps as well as the derivation of an AST are encapsulated in the *ModelParser* class whose *parseModel* be-
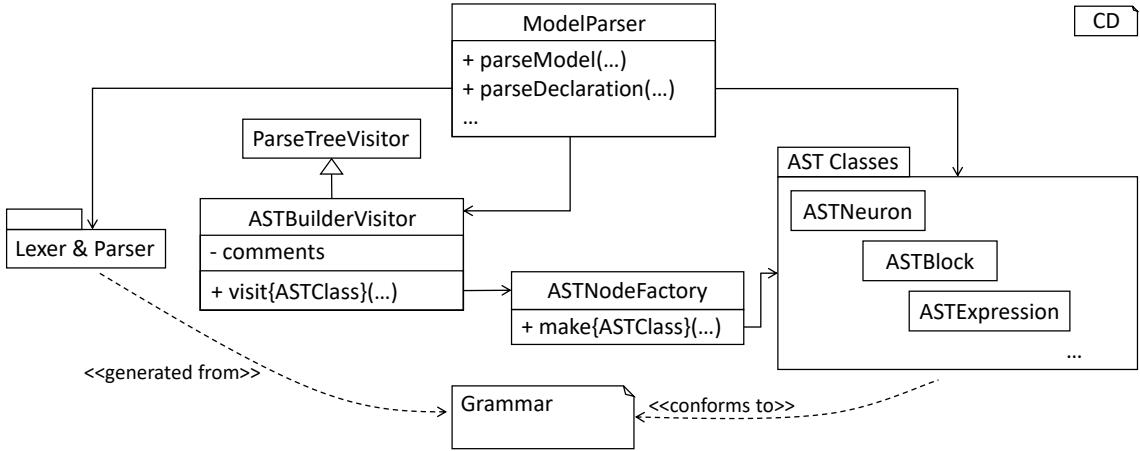
Figure 3.2: Overview of the lexer, parser and the AST classes: The grammar represents the artifact from which the lexer and parser are generated. Moreover, the *ASTBuilderVisitor* class extends the generated *ParseTreeVisitor* class and transforms the handed over parse tree to the respective AST. The *ASTNodeFactory* features a set of operations for node initialization. The *ModelParser* encapsulates all processes and can be used to parse complete models or single statements.


```
1  grammar PyNestML;                                                    Antlr
2
3  //Tokens                                                 everything between # and end of line
4  COMMENT: ('#' (~('\n' |'\r' ))*) -> channel(2);         put on a different token channel
5  NAME : ( [a-zA-Z] | '_' | '$' )( [a-zA-Z] | '_' | [0-9] | '$' )*;
6  INTEGER : [1-9][0-9]* | '0';
7  BOOLEAN: 'True' | 'False';                              string, integer and boolean literals
8
9  //Grammar Rules
10 neuron: 'neuron' NAME (block)* EOF;                     each neuron model introduced by the neuron's name
11 block: NAME ':' (assignment | declaration)* 'end';
12 assignment: NAME '=' expression;                        blocks of assignments and declarations
13 declaration: NAME type ('=' expression)?;
14 type: 'integer' | 'boolean' | 'string';
15 expression:  simpleExpression
16            | expression ('*'|'/') expression            arithmetic combinations of terminals
17            | expression ('+'|'-') expression;
18 simpleExpression: NAME | BOOLEAN | INTEGER;
```

Figure 3.3: A simplified grammar: Each neuron model is introduced by the keyword *neuron* and the neuron's name. A model is composed of an arbitrary number of *blocks* consisting of a name and a set of *declarations* and *assignments*. Declarations consist of a name, the data type and a value defining expression, while assignments only utilize a left-hand side name and a value providing expression. *Expressions* are either simple, i.e., a string, boolean or integer literal, or arithmetic combinations of other expressions.

23
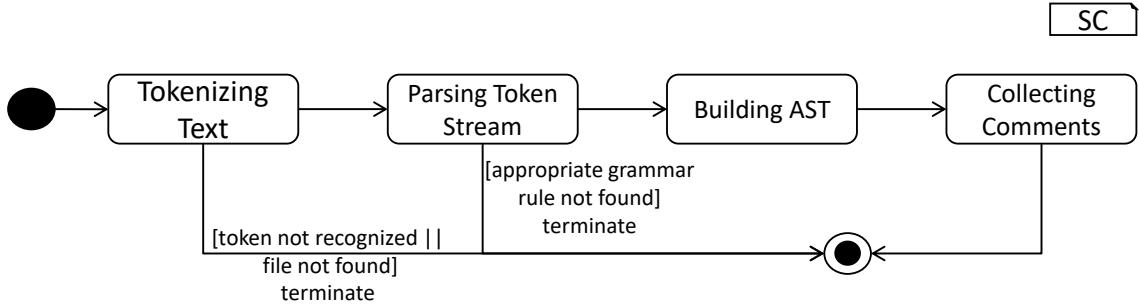
Figure 3.4: The model-parsing process: First, a model is decomposed into a stream of token objects. If a literal in the model is not constructed according to the token definitions, the process is terminated and the problem reported. Otherwise, the token stream is handed over to the parser which constructs a parse tree by taking the grammar rules into account. For sequences of tokens which are not constructed according to a grammar rule, an error is reported and the process terminated. A constructed parse tree is handed over to the *ASTBuilderVisitor* which constructs the respective AST. Finally, all comments are retrieved and stored.

havior is illustrated in Figure 3.4. Besides complete models, it is also often of interest to parse single instructions or expressions from a given string, e.g., for AST-to-AST transformations. The *ModelParser* class therefore provides parsing methods for each production in the grammar artifact, which can then be used to parse the respective element directly from a given string. In all cases, first, the parse tree is created by means of the generated lexer and parser. Subsequently, the further on introduced *ASTBuilderVisitor* is used to derive a respective AST representation.

AST classes couple fields for all required values with data retrieval and modification operations, cf. Figure 3.5. The abstract *ASTNode* class represents the base class which is extended by all concrete node classes. It implements features which are common for all concrete nodes, namely the *source location* of the element, a *comment* field as well as a reference to the respective *scope* of the element, cf. section 3.3. Moreover, it prescribes abstract methods which have to be implemented by all subclasses: The *equals* method can be used to check whether two objects are equal in terms of their properties, while an overwritten ``__str__`` method returns a human-readable form of the element. The concrete *accept* method is used by the further on introduced visitors in order to interact with the object.

A source location is an object of the *SourceLocation* class. By encapsulating this property in a separate class it is possible to provide a set of common utility. Among others the following two methods were implemented: The *before* function checks whether the current source location in the model is before a handed over one, while the *encloses* function indicates whether one source location encloses a different one.

Concrete AST classes are implemented according to the DSL's grammar. Explicit terminals such as the plus symbol are indicated by boolean fields, e.g., storing *true* whenever a respective terminal has been used. Implicitly declared terminals, e.g., *NAME*, are stored with the values stated in the textual model. References to sub-productions such as the *simple expression* are treated in the same manner, although here a reference to the initialized AST node of the sub-production is stored. Besides standard functionality for the retrieval of data, each AST class inherits and implements all operations as declared in the abstract *ASTNode* class. Figure 3.6 illustrates how the *ASTExpression* and *ASTSimpleExprssion*

24
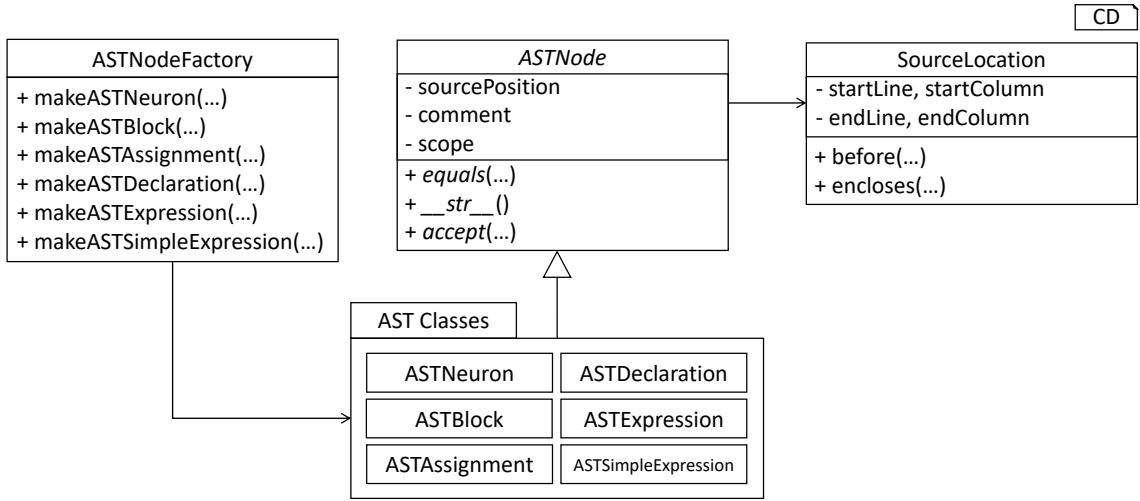
Figure 3.5: Overview of the AST classes: The *ASTNode* represents a base class for all concrete AST classes. Each AST node stores a reference to a *SourceLocation* object, representing the position in the textual model where the element has been defined. The *ASTNodeFactory* is used to create new instances of AST nodes.

classes are constructed from the respective production in the grammar.

Due to Python's missing concept of method overloading, it is not possible to define several standard constructors for a single AST class. This problem is tackled by means of the *factory* pattern [Gam95]. For each instantiable node, the *ASTNodeFactory* class defines one or more operations which can be invoked to return a new object of the respective class, cf. Figure 3.5. By providing all functions with a distinct name, method overloading is avoided.

The *ASTBuilderVisitor* class implements a parse tree visiting process which initializes the respective AST representation, cf. Figure 3.2. As demonstrated in Figure 3.7, the processing encapsulated in this class visits all nodes in a model's parse tree and creates AST nodes with the retrieved information. The parse tree stores all terminals, e.g., numeric values, as strings. For token classes which model value classes, e.g., strings or numeric values, their values are stored in correctly typed attributes of the AST. For each field of a parse tree node, the *ASTBuilderVisitor* therefore checks whether a value is available, e.g., a stated numeric literal. In cases where a value has been provided, it is retrieved, correctly casted and stored in the AST node. For non-terminals, the procedure is executed recursively by calling the *visit* method. The result is an initialized AST.

Although not crucial for the correct generation of a model implementation, comments as contained in the source model can be beneficial whenever an inspection of generated code is necessary. Here, it is often intended to retain source comments. As declared in Appendix A, the lexer hands all elements embedded in comment tags over to a different token channel. Each comment is delegated to the comment channel, where all comment tokens are stored and retrieved whenever required. In order to extract and transfer comments from tokens to their respective AST nodes, the *CommentCollectorVisitor* has been implemented, cf. Figure 3.8. It inspects the token stream and retrieves all comments which belong to the corresponding node. For this purpose, the *CommentCollectorVisitor* stores a reference to the initial token stream. Moreover, four methods are provided: The *getComment* function represents the orchestrating method and is used to invoke the col-

25
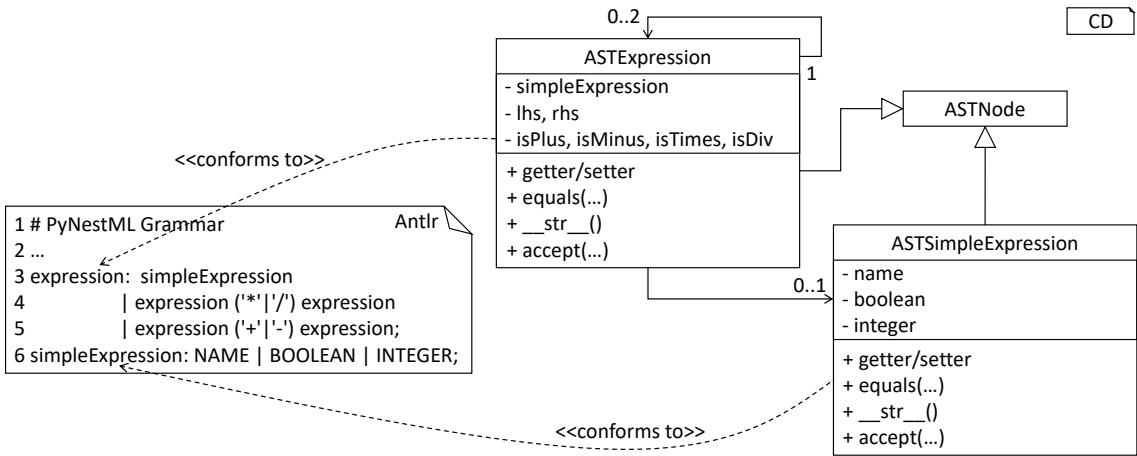
Figure 3.6: From Grammar to AST Classes: Each production in the grammar is used to construct a new AST class. For each terminal and referenced sub-rule, an attribute is created. A set of operations provides functionality for the visualization of nodes, data retrieval, and manipulation.

Figure 3.7: The *ASTSimpleExpression* node creating method: With the overall structure of the DSL in mind, this method is constructed to directly store correctly typed values. The position of the element in the model is retrieved and stored in a new *SourceLocation* object. Finally, a new AST node is created by the respective factory method.
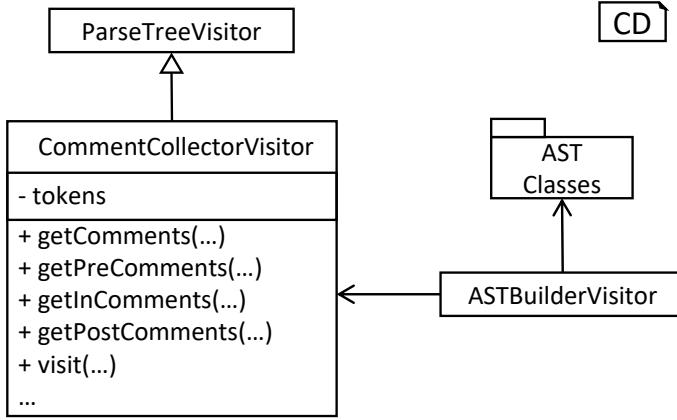
Figure 3.8: The *CommentCollectorVisitor*: The visitor implements a process for the collection of comments in arbitrary nodes of the parse tree. In order to simplify the processing, merely the *visit* method has to be called. This method delegates the work to the *getComments* function and finally returns all collected comments. The comment collector extends the *ParseTreeVisitor* and is called within the *ASTBuilderVisitor* whenever an AST is constructed.
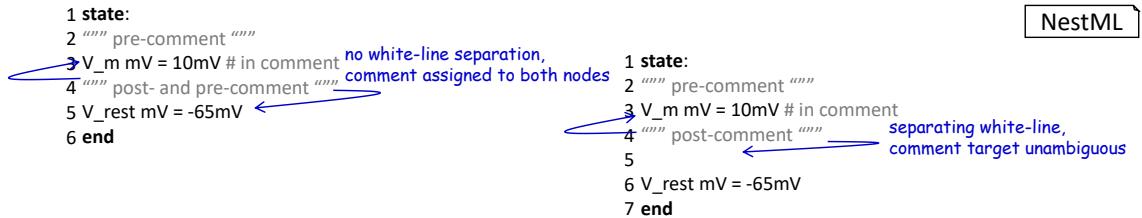
Figure 3.9: Illustration of the comment-processing routine: The target of a comment is recognized unambiguously if a separating white-line is inserted, otherwise the comment is added to both enclosing nodes.

lection of all pre-comments (stated before a statement or block), the in-comments (single line comments in the same line) and finally the post-comments stated after a statement or block in the textual model. In the following we exemplify the processing of pre-comments, the same procedure is applied analogously for the collecting of in- and post-comments. It should be noted that detection of a comment's target is ambiguous. For instance, in a situation where two statements with a single comment in between are given without any white-line separating one or the other, it is not possible to determine whether it represents a post-comment of the first statement or the pre-comment of the second one. The following simple yet sufficient concept has been developed: In order to highlight a comment as belonging to a certain element, it is necessary to separate the comment by means of a white-line as demonstrated in Figure 3.9. In the case that no white-line is injected, the comment is handed over to the previous and subsequent element. The user is therefore able to denote which comments belong to which element by inserting additional newlines.

The processing of pre-comments is implemented in the following manner: First, the *CommentCollectorVisitor* checks whether the processed node represents the first element in the artifact (e.g., the first definition of a neuron). In this case, the number of white-lines before the element is not relevant and all preceding comments are stored together with the node. Otherwise, starting from the position of the current context, the token stream is

inspected in a reversed order. In the case that a normal element token (e.g., the declaration of a variable) is detected, the loop is terminated since the next element has been reached. If a comment token is detected, then it is put on a stack. Such a handling is required in order to detect whether the comment belongs to the currently handled node, or represents an in-comment of the previous node. If an empty line is detected, then all tokens on the stack are stored in the list of returned comments. Whenever two subsequent white-line tokens have been detected (thus a separating white-line), the overall process is terminated. The visitor returns the collected list of comments in a reversed order to preserve the initial ordering. This process is executed analogously for post-comments. However, here it is not necessary to reverse the list or the token stream. A inverse traversal of the token stream is only necessary to detect where a pre-comment has been terminated. In the case of in-comments, no special handling is implemented. Instead it is simply checked whether before the next end-of-line marker a comment token is contained. To make comments more readable, the *replaceDelimeters* function removes all comment delimiters from the comment string.

Separating the model-parsing and comment-collecting subprocesses leads to an even clearer separation of concerns and benefits maintainability. New types of comment tags can be easily implemented without the need to modify the AST builder. All modifications are therefore focused in the *CommentCollectorVisitor*, while the initial grammar is kept programming language-agnostic. The comment collecting operation is invoked during the initialization of an individual AST node in the AST builder.

This section introduced the model-parsing process which constructs the AST from a textual model. Here, we first introduced the starting point of each DSL, namely the grammar artifact, and subsequently outlined how the implementation of a lexer and parser by hand can be avoided by means of Antlr. Instead, these components were generated and embedded into PyNestML. Due to the missing typing and assisting methods in the parse tree as returned by the parser, a set of AST classes was implemented and introduced in detail. Each class represents a data structure which is used to store details as retrieved from the parse tree. To this end, the *ASTBuilderVisitor* class and its AST initializing approach were demonstrated. The result of steps introduced above is a parsed model represented through an AST. Finally, the *CommentCollectorVisitor* demonstated how comments in source models can be collected and stored. Although not crucial for creation of correct target artifacts, comments can still be beneficial troubleshooting the generated code.

3.2   Symbol and Typing System

Continuing with an initialized AST, PyNestML proceeds to start collect information regarding the context. For this purpose, we first establish a data structure for the storage of context related details by means of symbol. Subsequently we demonstrate how predefined properties of PyNestML are integrated by means of the *predefined* subsystem. Finally, we show how types of expressions and declarations can be derived.

Chapter 2.1 demonstrated how *symbols* can be used to store details of pre- and user-defined functions and variables. The abstract *Symbol* class represents a base class for arbitrary symbols. It features attributes which are common for all concrete symbol types, amongst others a *reference* to the AST node used to create the symbol, the *scope* in which the element is located, the *name* of the symbol and a *comment*. Besides common data encapsulation methods, only the *isDefinedBefore* method is provided. This method checks whether a symbol has been defined before a certain *source location* and is used
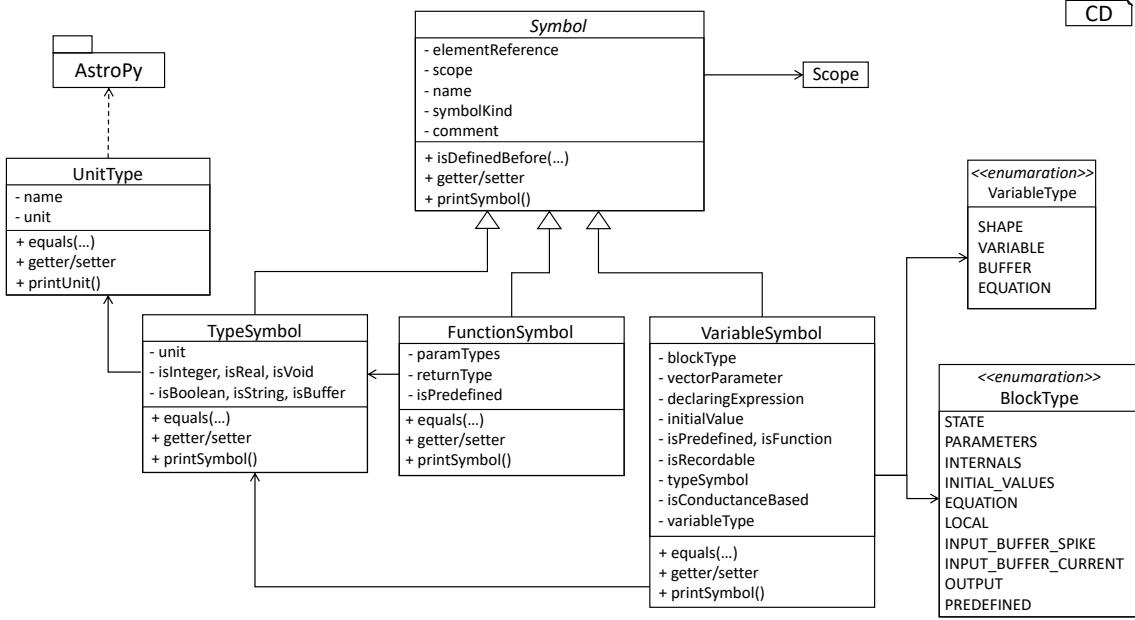
Figure 3.10: The *Symbol* subsystem: The abstract *Symbol* class prescribes common properties. This class is implemented by the *TypeSymbol* to represent concrete types. *Function-Symbol* and *VariableSymbol* store declared functions and variables. For more modularity, the *UnitType* class is used as around the *AstroPy* unit system [ART+13]. *VariableType* and *BlockType* represent enumerations of possible types of variables and blocks.

during semantical checks, cf. section 3.3. Figure 3.10 provides an overview of classes as implemented in PyNestML to enable a storage of semantics and types.

A *TypeSymbol* represents a type as used in declarations and function signatures, and can be either a primitive or a physical unit. In its current state, the type system supports the primitive types *integer*, *real*, *void*, *boolean* and *string*. Whether a type is a primitive is represented by a boolean field for each type, while physical units are stored as references to the corresponding *UnitType* objects. The *UnitType* class is a simple wrapper for the *AstroPy* unit system as introduced in section 2.2 and is used to couple an *AstroPy* unit object [ART+13] with a processable *name* as well as *equality*- and data-access operations. The final attribute of the *TypeSymbol* class is a boolean indicator whether a buffer or non-buffer type is represented. As indicated in Appendix A, *spike* buffers can be declared with an arbitrary data type. As we will demonstrate in chapter 4, the backend utilizes different approaches for the generation of buffer and non-buffer types.

The *VariableSymbol* class represents the second type of symbols. Each *VariableSymbol* object symbolizes a variable or constant as defined in the source model. It stores the type of block in which it has been declared as an element of the *BlockType* enumeration type. According to the grammar, each variable symbol can be defined in a *state* block, the *parameters* or *internals* block, the *initial values* or *equations* block. Moreover, given the fact that ports are regarded as variables with stored values, the block types *input buffer current*, *input buffer spike* and *output* are provided. Finally, the type system is able to mark variables as being declared in a *local* block, e.g., a user-defined *function* block or the *update* block, or as a predefined element of PyNestML, e.g., the global time variable *t*. The type of a block in which the element has been declared is required for the correct generation of target platform-specific code as introduced in chapter 4. PyNestML

29
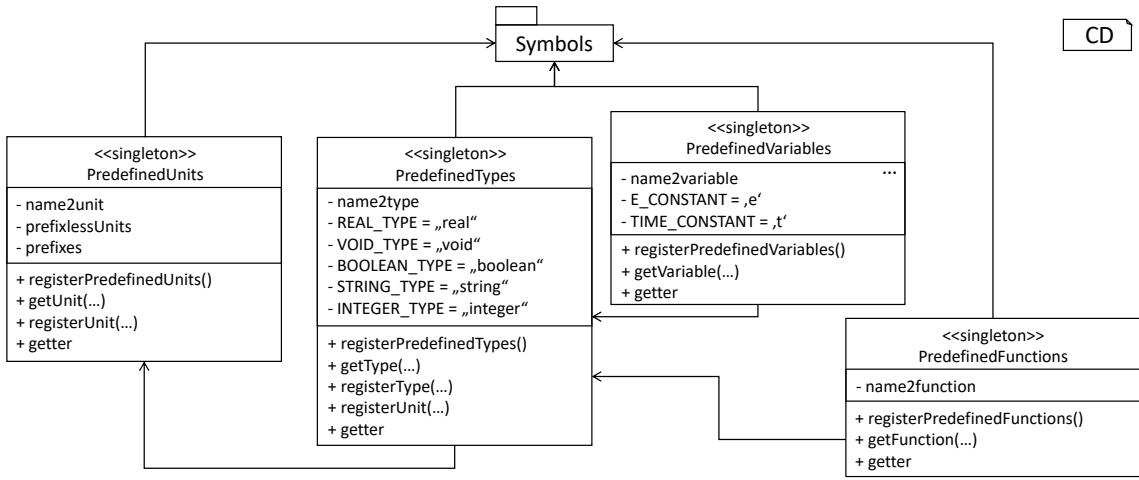
Figure 3.11: The *predefined* subsystem: By utilizing the *Symbol* classes, a collection of *UnitType* objects is created representing physical units. Together with primitive data types, these units are encapsulated in *type symbols* and stored in the *PredefinedTypes* collection, before being used in *PredefinedVariables* and *PredefinedFunctions*.

marks variables defined in the *equations* block as being *shapes* or *equations*. Variables defined in the input block are marked as being a *buffer*, while all other elements are simple *variables*. To this end, the *VariableType* enumeration type is implemented. By utilizing such a specification it is easily possible to sort symbols according to the property they represent. A corresponding getter function can then be used to retrieve buffers or shapes as required in semantical checks and code generation, cf. section 3.3 and chapter 4. The remaining attributes represent a collection of characteristics which are common for declared elements: A variable symbol can have a *vector parameter* indicating that a vector variable is given. The boolean fields *is-predefined*, *is-function* and *is-recordable* indicate whether the elements have been marked by keywords in the source model or represent predefined concepts, i.e., an element which is always available in PyNestML as in the case of the global time variable t. The *is-conductance-based* marks buffers with the unit type *Siemens*[1], while the *type symbol* stores a reference to an object representing the type of the variable. The *declaring expression* as well as the *initial value* attributes are used in the context of equations. The *declaring expression* field stores a reference to the expression denoting how new values of the equation have to be computed. Analogously the *initial value* stores the starting value of a differential equation. In the case that a non-equation symbol is stored, the *declaring expression* is used to simply store a right-hand side expression.

The *FunctionSymbol* is the last type of symbol and stores references to pre- and user-defined functions. Consequently, each symbol consists of a *name* of the function, the return type represented by a type symbol and a list of parameter type symbols. A boolean field indicates whether the corresponding function is predefined or not. In contrast to the variable symbol, function symbols do not feature further specifications or characteristics, e.g., the type of block in which they have been defined. Consequently, only a basic set of data access operations is provided.

In order to initialize a basic collection of types, variables and symbols, the *predefined*

---

[1]conductance-based buffers are processed differently during code generation in NEST
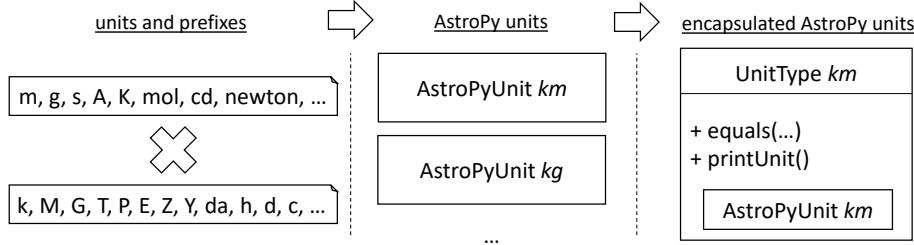
Figure 3.12: Instantiation of SI units with *AstroPy* [ART⁺13]: First, all basic units and all available prefixes are collected in two separate lists. Then, for each unit and each prefix, a combined unit is created, e.g., with the prefix *kilo* and the unit *gram*, a new unit *kg* is initialized. Each created unit is represented by an AstroPy unit object. For equality checks and printing operations, the *UnitType* wrapper class is used around each AstroPy unit object.

modules as illustrated in Figure 3.11 are used. All four types of the further on introduced symbol collections ensure that a basic set of components is always available in processed models. In the case of physical units, the units as provided by PyNestML represent a functionally complete set, i.e., it is possible to derive arbitrary units by combining the provided ones.

The *PredefinedUnits* class subsumes a routine used to initialize all basic physical units. Figure 3.12 exemplifies how for each base unit, e.g., *volt* or *newton*, and each available *prefix*, e.g., *milli* or *deci*, a combined *AstroPy* unit is created and wrapped in an object of the previously presented *UnitType* class. As opposed to variables which are only valid in their corresponding models, units and types are not specific to a certain neuron context, but valid for all possible models. Consequently, PyNestML stores all types globally for all processed models. The *PredefinedUnits* class features operations to check whether a given string represents a valid unit definition, e.g., *ms*, while the *getUnit* method is used to retrieve the object representing a unit defined by the string. At runtime, often new combinations of existing bases are derived. For instance, in the case of a multiplication of two variables of type *ms*, it is necessary to derive and register a new unit ms^2. While the derivation of new units is delegated to the further on introduced visitors, the *registerUnit* method can be used to insert a new unit into the type system. An encapsulation of units in the *UnitType* instances and the storage in the *PredefinedUnits* collection makes maintenance and extensions easy to achieve: In the case that the given type system is no longer applicable or a new alternative has been found, the corresponding *UnitType* wrapper can be simply wrapped around a different library without affecting the remaining framework.

Beside physical units, PyNestML is also able to store other types. As previously introduced, primitive types are the second type of objects which have to be managed. For this purpose, PyNestML subsumes physical units and primitive types in a single class, namely the *PredefinedTypes*. In consequence, predefined types consist of type symbols for the primitive types as well as all units stored in the *PredefinedUnits* class, cf. Figure 3.11. This separation has been employed in order to provide a central component for the handling of predefined as well as collected types, while the unit system in the background remains an exchangeable component. For each unit stored in the *PredefinedUnits*, PyNestML creates a new type symbol and stores it in the *PredefinedTypes*. Moreover, all types are treated as *singletons* [VHJG95], i.e., the system detects and prevents redundant registration of a given type. Consequently, whenever the *getType* operation is called, only

a reference is returned. Only buffer and non-buffer type symbols are treated as individual instances due to their different handling in the generating backend. The handling of types as singletons makes equality checks easy to achieve and reduces the overall memory consumption during the model processing[2]. The *PredefinedTypes* class features a set of operations used to get a type symbol or register a new one. The *getType* function includes more elaborated processing. Physical unit objects which do not represent real units, e.g., in the case of $ms/ms == 1$, are detected and treated as being *real* typed. Each unit is simplified before being registered in order to avoid a redundant storage of equal units, e.g., $ms == ms * ms/ms$. In conclusion, this method represents the overall interface to type systems and makes extensions by new primitive as well as unit types easy to achieve, while the architecture remains modular. With the *PredefinedTypes* class all components required to derive new types are already available in PyNestML, i.e., by combining basic physical units the type system is able to deal with compound units.

Types are subsequently used in the *PredefinedVariables* and *PredefinedFunctions* classes to denote the types of the elements. The *PredefinedVariables* class stores all predefined variables available in PyNestML. In its current state, PyNestML provides a set of predefined variables often required in neuroscientific models, including the global time constant t for the time past the start of the simulation, and Euler's number e. Moreover, PyNestML features a concept for *unit variables*. Consequently, it is also possible to utilize the name of a physical unit as a variable. By utilizing such a concept it is easily possible to state expressions representing new, compounded units as part of a computation. For instance, a given expression $55 * mV/nS$ is treated as semantically as well as syntactically correct. By handling units as predefined variables, the framework is able to apply the same set of arithmetic rules as for all other types of expressions, cf. Appendix A. Compound physical units are therefore created by stating defining arithmetic expressions with basic units. All units as defined in the *PredefinedTypes* class are therefore also registered as predefined variables. However, in contrast to derived physical units which are automatically stored in the set of predefined types, PyNestML does not add new unit variables to the predefined variables. Such a handling is not required since complex arithmetic combinations of units are treated as an aggregation of basic units, consequently, only variables for basic units are required. The *PredefinedVariables* class features methods for the retrieval of symbols for predefined variables as well as a *getVariable* method which can be used to detect if a variable is predefined. In the case that a handed over name does not correspond to a variable, *none* is returned. In this case, the client method has to take care of correct steps. In contrast to types, variable symbols located in concrete models are never added to the set of predefined ones given the fact, that these properties are local to their context and should not be visible to other models. PyNestML reports declarations of variables with the same name as one of the predefined variables as an error, cf. section 3.3.

Analogously to the *PredefinedVariables*, PyNestML uses the *PredefinedFunctions* class to store all predefined functions. In its current state, PyNestML supports 21 different mathematical and neuroscientific functions. As already introduced, each function symbol consist of a *name*, the type of the *return* value as well as a list of *parameter types*. All predefined functions are therefore individually initialized and stored. In order to ensure a correct type, type symbols managed by the *PredefinedTypes* class are retrieved and references stored. The *getFunction* method can then be used to request the function symbol for a specified name.

With a data structure for the representation of types as well as a basic collection of

---

[2]at the beginning there are roughly 600 different basic units in PyNestML
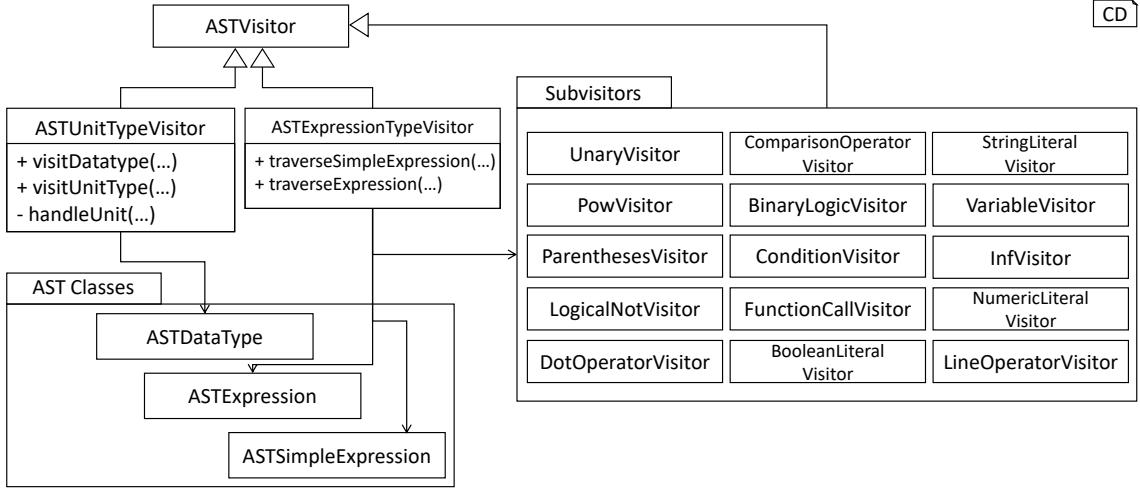
Figure 3.13: Overview of the type-deriving visitor subsystem: The *ASTUnitTypeVisitor* derives correct types for declarations of types as stored in *ASTDataType* nodes, while the *ASTExpressionTypeVisitor* class takes care of correct type derivation in expressions. Here, a set of assisting sub-visitors is used to derive the type symbol based on the concrete type of the expression, e.g., boolean literals or arithmetic expressions, each of which corresponding to one production of the *expression* grammar rule, cf. Appendix A.

fundamental types, PyNestML is now able to enrich the previously constructed AST by a new property, namely the concrete type of all elements. For this purpose, all AST nodes which have to be specified by a type are now, after the AST has been constructed by the lexer and parser, extended by a reference to a *TypeSymbol* object. Based on the type of AST node for which the type has to be derived, this step has been separated into two different phases in order to enforce a clear separation of concerns. Figure 3.13 subsumes the type derivation subsystem.

The simpler case is the handling of data type declarations of constants and variables defined in the model. Given the grammar for the declaration of a type where no plus or minus arithmetic operators are supported, this processing can be completely implemented in a single method. This process is therefore encapsulated in the *ASTUnitTypeVisitor* class which derives the concrete type symbol of a type represented by an *ASTDataType* node. The visitor extends the base visitor class, traverses the tree and invokes further steps whenever an *ASTDataType* node is detected. The *visitASTDataType* method checks whether a primitive or a unit type is represented by the visited node.

In the case that a primitive type has been used, a respective type symbol is simply retrieved from the predefined types collection and the reference stored. Otherwise the handling is handed over to the *visitASTUnitType* subroutine. This method checks how the data type has been constructed. If a simple name is used, e.g., mV, then the corresponding symbol is retrieved from the predefined types and stored. Otherwise, the method proceeds to recursively descend to the leaf nodes of the AST node, cf. Figure 3.14. As defined by Appendix A, leaf nodes are always simple units or an integer typed value. The visitor checks which type of operation has been used to combine the leaf nodes and proceeds accordingly. For power expressions, e.g., ms^2, first the type of the base is derived and consequently extended by means of the power operation. Encapsulated units, e.g., $(ms * nS)$, are updated by setting the outer unit according to the inner one. In the case of

33
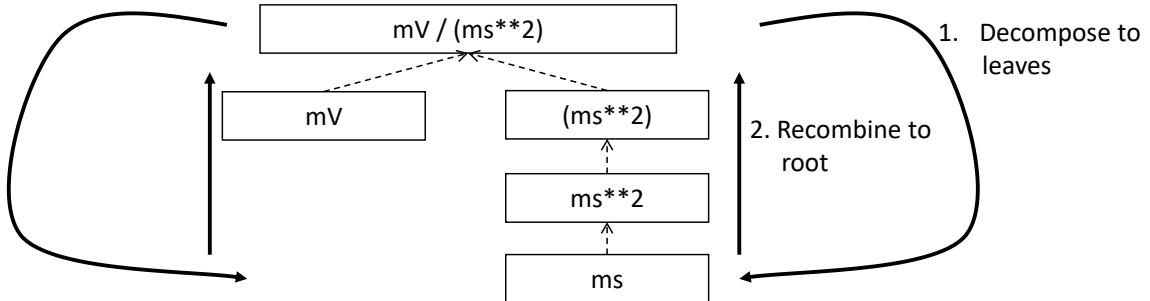
Figure 3.14: Derivation of types in *ASTDataType* nodes: First, the type defining expression is decomposed into its leaves. For each leaf, the corresponding type is retrieved from the *PredefiendTypes* class. Finally, all types are recombined according to the stated operations up to the root and the overall type is stored.

arithmetic point operators, the *visitASTUnitType* method first checks whether a division or multiplication of units is performed. For the former, the left-hand side is first inspected for its type. Given the fact that data types support a numeric value on the left-hand side, e.g., $1/ms$, the *visitASTUnitType* method checks whether it is a numeric type or not. If a numeric value is used, the method retrieves and divides it by the right-hand side. In the case of unit types, the procedure is applied recursively. Multiplication of two units is handled analogously, although here the language does not provide a concept for numeric left-hand side values.

In the case of *expressions*, it is necessary to propagate the types of the leaves to the root of the AST node. This process requires a more sophisticated handling and traversal of the expression. The complex structure of expressions where line-, point- as well other operators can be used makes a modular structure necessary. The derivation of expression types is therefore handled by the *ASTExpressionTypeVisitor*, cf. Figure 3.13. Extending the base visitor, this class represents a traversal routine which, depending on the type of the currently processed expression, invokes an appropriate sub-visitor. The currently active sub-visitor is referenced in the *real self* attribute and indicates how parts of the expressions have to be handled. It consequently checks the type of an element in the expression, e.g., whether it is a boolean literal or an arithmetic combination of two subexpressions, and sets the *real self* visitor according to this element. In its current state, PyNestML supports 15 different sub-visitors, amongst others the *unary visitor* used to update the expression prefixed with a unary plus, minus or tilde, the *power visitor* for the calculation of the type of an exponent expression, the *parentheses visitor* for the type derivation of encapsulated expressions, the *logical not* visitor for the handling of negated logical expressions, the *dot* and *line operators* for handling of arithmetical expressions, the *comparison visitor* for handling of comparisons and the *binary logic* visitor for the handling of logical *and* and *or*.

The use case demonstrated in Figure 3.15 exemplifies the overall process: Given the expression *10mV + V_m + (true and false)* with the variable *V_m* of unit type *millivolt*, first, the *ASTExpressionTypeVisitor* descends to the leaf level, namely the nodes $10mV$, *V_m*, *true* and *false*. For $10mV$, the *numeric literal visitor* is activated which checks whether the expression utilizes a physical unit or not. In the case that a unit is used, the visitor resolves the name of the unit and sets the retrieved type symbol to the type of the node. If no unit is used, the visitor checks whether a *real* or *integer* literal is present and retrieves the corresponding type symbol from the predefined types collection.

```
  ┌──────────────────────────────────┐
  │  10mV + V_m + (true and false)   │  Error: Addition of boolean and mV.
  └──────────────────────────────────┘

  mV  ┌───────────────┐        ┌──────────────────┐
      │  10mV + V_m   │        │  (true and false) │  boolean
      └───────────────┘        └──────────────────┘

  mV  ┌────────┐   ┌──────┐         ┌──────────────┐
      │ 10mV   │   │ V_m  │  mV     │ true and false │  boolean
      └────────┘   └──────┘         └──────────────┘

                          boolean ┌──────┐    ┌──────┐ boolean
                                  │ true │    │ false│
                                  └──────┘    └──────┘
```
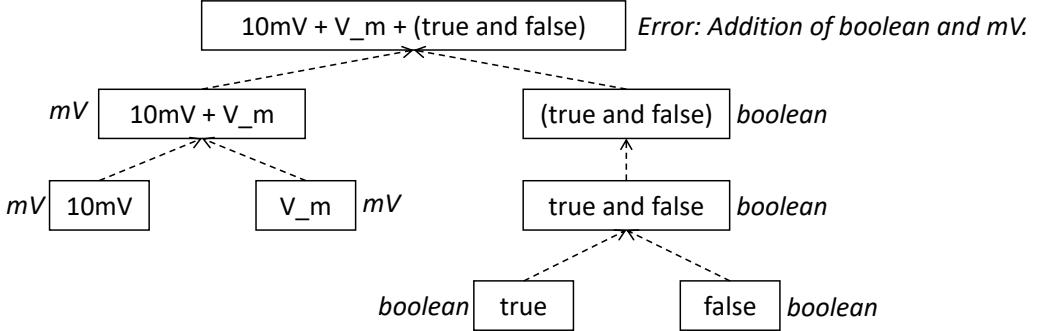
Figure 3.15: Derivation of types in *ASTExpression* nodes: Analogously to *ASTDataTypes* nodes, an expression is first decomposed into its leaf nodes. Subsequently, the corresponding variable symbol is resolved, and its type symbol retrieved. Type symbols are combined according to the operations used to construct the expressions. In the case of errors, e.g., a combination of boolean and numeric types, an error message is propagated to the root.

Analogously, the V_m variable is inspected by the *variable visitor*, and the variable name is resolved to the corresponding variable symbol. Each variable symbol stores a reference to its type symbol. Consequently, this type symbol is retrieved and used as the type of the literal in the expression, e.g., here the type mV. For the boolean *true* and *false*, the *boolean visitor* is used. It simply inspects whether a boolean literal has been used and sets the type of the corresponding expression to the boolean type symbol as stored in the predefined types collection. Having the types of all leaf nodes, the visitor starts to ascend. The expression $10mV + V\_m$ is a line operator combination of two values, thus the *line operator visitor* is activated. The arithmetic plus operator should only be applicable for numeric values and variables representing such. The left- as well as the right-hand side of the plus operator refer to unit values and have the same type, hence the overall type of the expression is set to mV. In the case of *true and false*, the *and* operator can only be used to combine boolean values, which applies in the given case, thus the *binary logic visitor* is used which updates the type of the combined expression to *boolean*. The boolean expression has been encapsulated in parentheses which makes an invocation of the *parentheses visitor* necessary. This visitor simply retrieves the type of the inner part of the encapsulated expression and updates the type of the overall expression accordingly, e.g., in our case to *boolean*. Finally, the root of the expression is reached, namely the arithmetic combination of the expressions $10mV + V\_m$ of type mV and *(true and false)* of type *boolean*. Obviously, such an expression is not correctly typed. The *line operator visitor* detects that incompatible types have been used and sets the type of the expression to an error value. In order to enable the PyNestML to store either a correct type or an error message, the *Either* class is used. This class stores either a reference to a *type symbol* or a string containing an error message. By storing an object of this type instead of an undefined unit, PyNestML is able to derive and interact with errors and propagate the messages to the root of the expression. All detected errors are hereby reported as being of semantical nature, cf. section 3.3. In the given example, the overall type of the expression is an object of the *Either* class with an error message stating that an arithmetic combination of numeric and non-numeric values is not possible. Together with all remaining visitors, this system is able to derive the type of arbitrary expressions by propagating and combining leaf-node types to the root. Here we see exactly why the physical unit system *AstroPy* with its support for arithmetic operators was used: Given

```
ASTVisitor
```

```
ASTSymbolTableVisitor
─────────────────────
- currentBlockType
─────────────────────
+ updateSymbolTable(...)
+ visit{ASTNode}(...)
```

```
ModelParser
```

```
SymbolTable
─────────────────────
- name2neuronScope
─────────────────────
+ initializeSymbolTable()
+ addNeuronScope(...)
+ deleteNeuronScope(...)
+ printSymbolTable(...)
```

```
CoCosManager
─────────────────────
+ postSymbolTableBuiderChecks(...)
+ postOdeSpecifcationChecks(...)
+ checkExpressionCorrect(...)
...
```

```
CoCo
─────────────────────
+ checkCoCo(...)
```

```
Concrete CoCos
```

```
AST Classes
```

```
Symbols
```

```
Scope
─────────────────────
- enclosingScope
- declaredElements
- scopeType
- sourcePosition
─────────────────────
+ addSymbol(...)
+ addScope(...)
+ resolveToSymbol(...)
```

Figure 3.16: Overview of semantical checks: The orchestrating *ModelParser* class utilizes the *ASTSymbolTableVisitor* to construct a model's hierarchy of *Scope* objects. Each scope is populated by *Symbol* objects corresponding to elements defined in the respective model. In order to manage all processed neurons in a central unit, the *SymbolTable* class is used. Finally, the *ModelParser* calls all model-analyzing routines of the *CoCosManager* class and checks the model for semantical correctness. The *CoCosManager* class utilizes different *CoCos* to check several properties of the given model.

the expression $10mV * 2ms$, PyNestML should be able to combine the underlying units to a new one, and the overall type of the expression should be set to $mV * ms$. Such a processing is vehemently simplified if the framework's underlying physical units library supports arithmetic operations on units for the creation of new ones.

This section introduced the type system and showed how PyNestML stores and processes declarations and their respective types. Here, we first implemented data structures to store details of defined elements in the model. Subsequently, we demonstrated how a set of predefined elements is initialized by the *predefined* subsystem. Finally, these elements were used to derive the type of all expressions located in the model by means of the *ASTDataTypeVisitor* and *ASTExpressionTypeVisitor* classes. We will come back to types in the next section where correct typing of expressions as well as other semantical properties are introduced.

3.3 Semantical Checks

After the AST of a given model has been constructed, comments have been collected and the type of all elements derived, the model-processing frontend proceeds to the last step, namely the checking of the semantical correctness of a handed over textual model. For this purpose, we first implement data structures for the storage of a neuron's concrete context, namely the *SymbolTable* and *Scopes* classes. In order to fill these components with context information, a collecting process implemented in the *ASTSymbolTableVisitor* is used. After the context of a model has been established, it remains to check for correct semantics. This task is delegated to the *CoCosManager*, a component which manages a

collection of *context conditions*, cf. section 2.1. Figure 3.16 illustrates which components have been implemented to store, collect and check semantical details of a model.

The *SymbolTable* class has been implemented analogously to the concept introduced in section 2.1. This component represents a container which maps neuron names to their respective global scope. The scope of an AST object is hereby an element of the *Scope* class which stores a reference to its parent scope, leading to a tree-like structure of the scope layering. Utilizing such a structure accelerates the resolving of symbols and eases the working with the context of a model. All elements contained in a scope are hereby stored in a list. Each element is either a *Symbol* or a sub-*Scope*. The final two attributes of the *Scope* class store details regarding the type of the scope and the source location. The former is used to enable an easy to conduct filtering of scopes. For this purpose the enumeration type *ScopeType* is implemented. Each scope is marked as being *global*, *update* or *function*. All elements defined outside the *update* and *function* block are stored in a neuron's top-level scope, while the *update* and *function* block can be used to open new sub-scopes. The *source location* attribute contains the position enclosed by the scope. Storing this detail is beneficial especially in the case of error reports and troubleshooting of textual models.

Besides data retrieval and manipulation operations, the *Scope* class features several aiding methods: The *getSymbolsInThisScope* method can be used to retrieve all symbols in the current scope, while *getSymbolsInCompleteScope* also takes all shadowed symbols in ancestor scopes into account. The *getScopes* operation can be used to return all sub-scope objects of the current scope. In order to retrieve the top scope of a neuron, the *getGlobalScope* method can be used. Finally, the *resolve* methods are provided. The *Scope* class implements two different operations and supports a more precise retrieval of information. The *resolveToAllScopes* method can be used to retrieve all scopes in which a symbol with the handed over *name* and *symbol kind* has been declared. The *resolveToAllSymbols* returns the corresponding symbols. These methods can be used whenever shadowing of variables should be handled and all specified symbols returned. The respective single instance methods *resolveToScope* and *resolveToSymbol* can be used to return the first defined instance of a symbol specified by the parameters. Starting from the current scope, these methods first check if the specified symbol is contained in the scope. If such a symbol is found, it is simply returned, otherwise, the same operation is performed on the parent scope. In conclusion, this method can be used to check if a used element has been declared in the spanned scope of the current block. Figure 3.17 illustrates the resolution process.

The *SymbolTable* class represents a data structure which has to be instantiated and filled with the context information of concrete models. PyNestML delegates this task to the *ASTSymbolTableVisitor* class, a component which implements all required steps to fill the symbol table with life. The overall interface of this class consists of the static *updateSymbolTable* method which expects the concrete AST whose context shall be analyzed and updated accordingly. Based on the visited node, this operation invokes one of the following processings: In the case that an *ASTNeuron* node is visited, a new neuron wide scope is created. Moreover, in order to fill the scope with predefined properties which are always available in the context, references to elements of the *predefined* subsystem are stored. This step ensures that the resolution process of predefined and model-specific variables becomes transparent and accessible over the neuron's scope. It is therefore not required to access individual collections of the *predefiend* subsystem to get the respective elements. Instead, all symbols required by a model are stored in its respective top-level scope and the *PredefinedTypes* collection. Moreover, given the structure of the visitor, it is not directly possible to indicate certain details to processed child nodes, e.g., the top
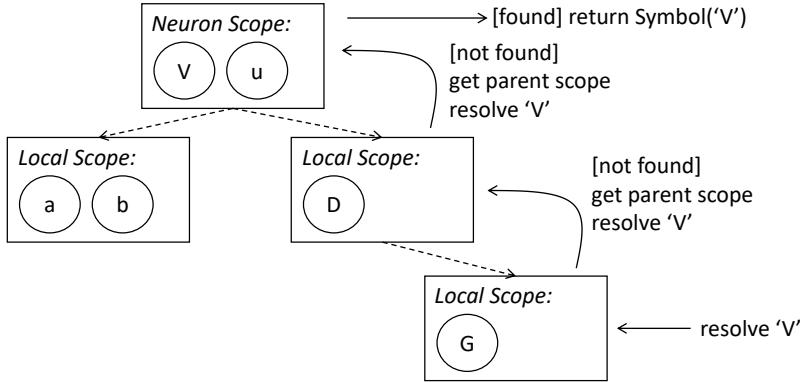
Figure 3.17: The symbol resolution process: The request to return a *Symbol* object corresponding to a given name is received by the nested scope. The scope is checked, and if no symbol with the corresponding name and type is found, a recursive call to the resolution process on the nesting scope is performed. If a symbol has been found, it is returned, otherwise an error is indicated by returning *none*.
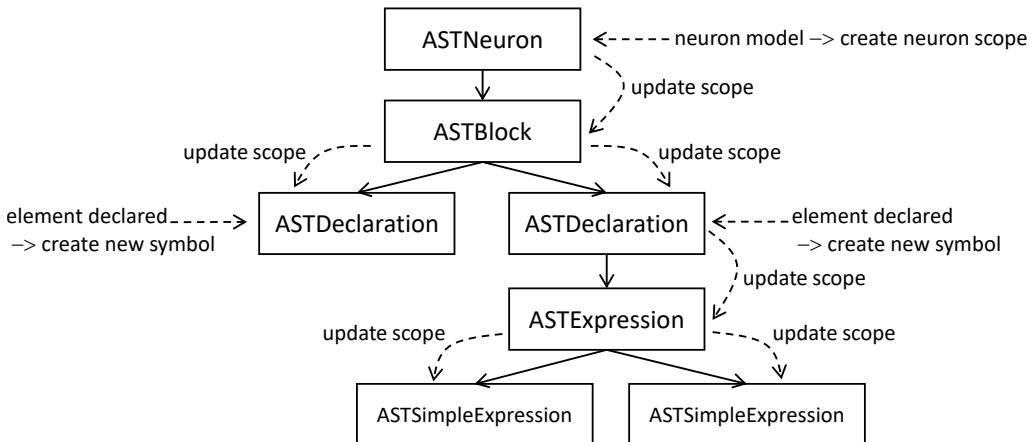
Figure 3.18: AST context-collecting and updating process: Starting at the root, i.e., the *ASTNeuron* object, the *ASTSymbolTableVisitor* creates a neuron-specific scope and descends into the AST. For each node, the routine checks if a child node is stored, and updates its scope according to the current one. Found declarations are used to create new symbols which are consequently stored in the parent's scope.

level scope of the currently handled neuron or which type of block[3] is processed. While the former is solved by a top-down update process as illustrated in Figure 3.18, i.e., before a node is visited, its scope is updated to the parent's scope, the latter requires storage of additional details. Consequently, the type of the currently processed block is stored and represented as a value of the *BlockType* enumeration, cf. section 3.2. Whenever a block of statements is entered, the type of the block is simply stored and removed after the block has been left. Newly created symbols inside the block check this value and derive the information in which type of block they were created. Such a processing is required in order to determine the *ScopeType* of each created (sub-)scope as well as the *BlockType* of created symbols[4].

The creation of new symbols and scopes is only required in a limited set of cases. Most often, only the scope reference of a handled element has to be updated. As shown in Figure 3.18, this step is done in a reversed order: The neuron's root AST node stores a reference to its scope, and subsequently sets the scope of its child nodes to the parent scope. In the case that a block is detected which has to span its own local scope, i.e., an *update* or *function* block, a new *Scope* object is created and stored in the parent scope. This new object is then set as the scope of the nested block and the process is continued recursively. Thus, whenever a scope-spanning block is detected, a new scope is stored in the parent scope, and used in the following as the current scope. The individual *visit* methods of the *ASTSymbolTableVisitor* therefore first update the scopes of their child nodes before a further traversal is invoked. Constants and variables declared in the model require an additional step. Here it is necessary to create a new *Symbol* object representing the declared element. Concrete information regarding the specifications of the symbol is stored in the current AST object, while the *TypeSymbol* can be easily retrieved by inspecting the *ASTDataType* child node. Here we see exactly why a preprocessing by the *ASTDataTypeVisitor*, cf. section 3.2, is required. Having an AST where all nodes have been provided with their respective *TypeSymbols*, the *ASTSymbolTableVisitor* can now easily retrieve this information and use it in *VariableSymbols*. All required details are therefore simply retrieved from the corresponding element, and a new *VariableSymbol* is created and stored in the current scope. In the case of user-defined functions, this process is performed analogously, although here a *FunctionSymbol* is created. The *ASTSymbolTableVisitor* executes this process for the whole AST and populates the symbol table with scope details. As a side effect, the scopes of all AST objects are updated correctly and can now be used for further checks.

After a neuron's scopes have been adjusted, the final step of the model-processing frontend is invoked, namely the checking of semantical correctness. As demonstrated in section 2.1, this steps is performed by means of so-called *context conditions*. Here a modular structure has been employed. PyNestML implements each context condition as an individual class with the prefix *CoCo* and a meaningful name, e.g., *CocoVariableOncePerScope*. In order to subsume the overall checking routine in a single component, the *CoCosManger* class has been implemented, cf. Figure 3.19. Its *postSymbolTableBuilderChecks* method can be used to check all context conditions after the symbol table has been constructed, while the *postOdeSpecificationChecks* method checks if all ODE declarations have been correctly stated in the raw AST.

Given the fact that context conditions have the commonality of checking the context of a neuron model, PyNestML implements the abstract *CoCo* super class. All concrete context conditions therefore have to implement the *checkCoCo* operation which expects a single

---

[3]state, function, equations etc.
[4]a detail required for appropriate code generation, cf. section 3.2
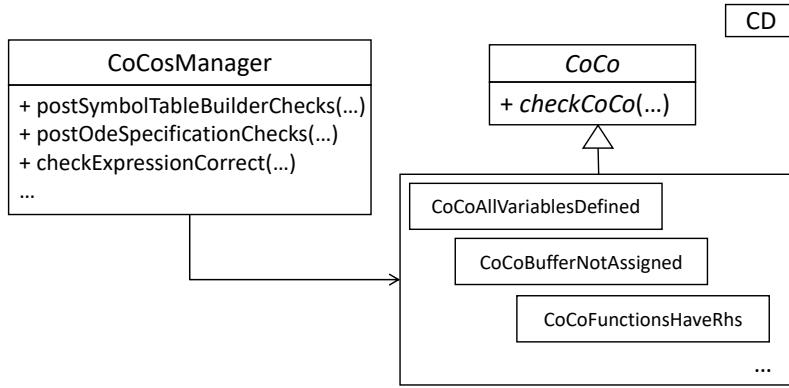
Figure 3.19: The *CoCosManager* and context conditions: The *CoCosManager* class represents a central unit which executes all required checks on the handed over model. Each checked feature of the model is encapsulated by a single class which inherits the abstract *CoCo* class.

AST for checking. Concrete context condition classes describe in a self-contained manner which definitions lead to an erroneous model. Consequently, here a *black list* concept is applied: For models which feature certain characteristics it is not possible to generate correct results. These characteristics should be reported. In its current state, PyNestML features 25 different context conditions which ensure the overall correct structure of a given model. The following composition outlines the implemented conditions.

1. *CoCoAllVariablesDefined*: Checks whether all used variables are previously defined and no recursive declaration is stated.

2. *CoCoBufferNotAssigned*: Checks that no values are assigned to (read-only) buffers.

3. *CoCoConvolveCondCorrectlyBuilt*: Checks that each *convolve* function-call is provided with correct arguments, namely a *shape* and a *buffer*.

4. *CoCoCorrectNumeratorOfUnit*: Checks that the numerator of a unit type is equal to one, e.g., $1/mV$.

5. *CoCoCorrectOrderInEquation*: Checks whether a differential equation has been stated for a non-derivative, e.g., $V_m = V'_m$ instead of $V'_m = V'_m$.

6. *CoCoCurrentBuffersNotSpecified*: Checks that *current* buffers are not specified with the keyword *inhibitory* or *excitatory*. Only *spike* buffers can be further specified.

7. *CoCoEachBlockUniqueAndDefined*: Checks that mandatory *update*, *input* and *output* blocks are defined exactly once, and all remaining types of blocks are defined at most once.

8. *CoCoEquationsOnlyForInitValues*: Checks that equations are only defined for variables stated in the *initial values* block.

9. *CoCoFunctionCallsConsistent*: Checks that all function calls are consistent, i.e., that the called function exists and the arguments are of the correct type and amount.

10. *CoCoFunctionHasRhs*: Checks that all attributes marked by the *function* keyword have a right-hand side expression.

11. *CoCoFunctionMaxOneLhs*: Checks that multi-declarations marked as *functions* do not occur, e.g., *function* V_m, V_n $mV = V_{init} + 42mV$. Several aliases to the same value are redundant.

12. *CoCoFunctionUnique*: Checks that all functions are unique, thus user-defined functions do not redeclare predefined ones.

13. *CoCoIllegalExpression*: Checks that all expressions are typed according to the left-hand side variable, or are at least castable to each other.

14. *CoCoInitVarsWithOdesProvided*: Checks that all variables declared in the *initial values* block are provided with the corresponding ODEs.

15. *CoCoInvariantIsBoolean*: Checks that the type of all given invariants is *boolean*.

16. *CoCoNeuronNameUnique*: Checks that no name collisions of neurons occur. Here, only the names in the same artifact are checked.

17. *CoCoNoNestNameSpaceCollision*: Checks that user-defined functions and attributes do not collide with the namespace of the target simulator platform NEST.

17. *CoCoNoShapesExceptInConvolve*: Checks that variables marked as *shapes* are only used in the *convolve* function call.

18. *CoCoNoTwoNeuronsInSetOfCompilationUnits*: Checks across several compilation units (and therefore artifacts) whether neurons are redeclared. Only invoked when several artifacts are given.

19. *CoCoOnlySpikeBufferWithDatatypes*: Checks that only *spike* buffers have been provided with a data type. *Current* buffers are always of type *pA*.

20. *CoCoParametersAssignedOnlyInParameterBlock*: Checks that values are assigned to parameters only in the *parameter* block.

21. *CoCoSumHasCorrectParameter*: Checks that *convolve* calls are not provided with complex expressions, but only variables.

22. *CoCoTypeOfBufferUnique*: Checks that no keyword is stated twice in an input buffer declaration, e.g., *inhibitory inhibitory spike*.

23. *CoCoUserDeclaredFunctionCorrectlyDefined*: Checks that user-defined functions are correctly defined, i.e., only parameters of the function are used, and the return type is correctly stated.

24. *CoCoVariableOncePerScope*: Checks that each variable is defined at most once per scope, i.e., no variable is redefined.

25. *CoCoVectorVariableInNonVectorDeclaration*: Checks that vector and scalar variables are not combined, e.g. $V + V\_vec$ where V is scalar and V_vec a vector.

In the following we exemplify the underlying process on two concrete *context conditions*, namely *CoCoFunctionUnique* and *CoCoIllegalExpression*. The former is used to check whether an existing function has been redefined in a given model. With the previously done work, this property can be easily implemented: Given the fact that in the basic context of the language no functions are defined twice, the *checkCoco* method of the

CoCoFunctionUnique class simply retrieves all user-defined functions, resolves them to the corresponding *FunctionSymbols* as constructed by the *ASTSymbolTableVisitor* and checks pairwise whether two functions with the same name exist. In order to preserve a simple structure of PyNestML, function overloading is not included as an applicable concept. Thus, only collisions of function names have to be detected. If a collision has been detected, an error message is printed and stored by means of the further on introduced *Logger* class, cf. section 3.4. With the names of all defined *FunctionSymbols* (and analogously *VariableSymbols*) it is easily possible to check whether a redeclaration occurred. Moreover, the stored reference to the corresponding AST node can be used to print the position at which the model is not correct, making troubleshooting possible. Figure 3.20 illustrates the *CoCoFunctionUnique* class.

CD

CoCoFunctionUnique
+ checkCoCo(...)

CoCoIllegalExpression
+ checkCoCo(...)

ASTVisitor

AST Classes

ASTNeuron

ASTDeclaration
...

CorrectExpressionVisitor
+ visitASTDeclaration(...)
+ visitASTAssignment(...)
+ visitASTIfClause(...)
...
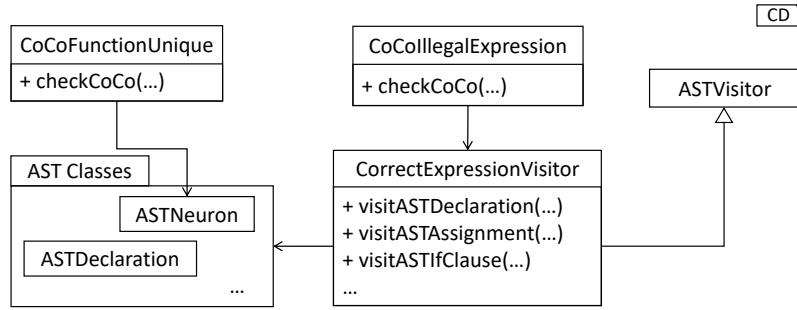
Figure 3.20: Simple and complex context conditions: Simple context conditions such as *CoCoFunctionUnique* can be implemented in a single function, while more complex conditions such as *CoCoIllegalExpression* also utilize additional classes and visitors. Both types of context conditions work on the handed over AST.

The second exemplified context condition *CoCoIllegalExpression* checks whether the expected data type of elements and their corresponding expressions have the same value. With the previously derived *TypeSymbols* of all AST nodes and the instantiated symbol table, here a simple process becomes sufficient for an in-depth checking of correctly typed models. To check correct typing of all required components, the assisting *CorrectExpressionVisitor* is implemented, cf. Figure 3.20. This visitor implements the basic *ASTVisitor* and overrides the *visit* method for nodes whose types have to be checked. In the case of *declarations* and *assignments*, it resolves the variable symbol of the left-hand side variable and retrieves the corresponding type symbol. For the right-hand side expression, the *getTypeEither* of the (simple) expression object is called. Finally, the *equal* method is used to check whether both types are equivalent. Here, an additional check has been implemented: Given the fact that most simulators disregard physical units, but work in terms of integers and doubles, it can be beneficial to allow certain implicit castings. For this purpose the *isCastableTo* method of the further on introduced *ASTUtils* class is used. This function can be invoked to check whether one given type can be converted to a different one. For instance, this method returns *true* whenever a physical unit *TypeSymbol* and a *real TypeSymbol* are handed over, since each unit typed value is implicitly regard as being of type real. Analogously, *real* and *integer* can be casted to each other, although here the fraction of a value might be lost. An implicit cast is always reported with a warning to inform the user of potential errors in the simulation. If an implicit cast is not possible, e.g., casting of a *string* to an *integer*, an error message is printed informing the user of a broken context. Warnings, therefore, state that a given model could possibly contain unintended behavior, while errors indicate semantical incorrectness.

The second type of checks as implemented in the *CoCoIllegalExpression* is a comparison of magnitudes: Values which utilize the same physical unit but differ in magnitude have to be regarded as being combinable. It should, therefore, be possible to add up $1mV$ and $1V$, although the underlying combination of a prefix and unit is not equal. This task is handed over to the *differsInMagnitude* method of the *ASTUtils* class, cf. section 3.4. This method simply checks whether the physical units without the prefixes are equal and returns the corresponding truth value. The remaining *context conditions* are implemented in an analogous manner: If complex checks on all nodes of the AST are required, a new visitor is implemented. In more simple cases a single function is sufficient. Errors and warnings are reported by means of the *Logger* class, cf. section 3.4.

In this section, we introduced how context related details of a model can be stored and checked. For this purpose, we first implemented the *SymbolTable* class which stores references to all processed neuron scopes. The *Scope* class has hereby been used to represent scope spanning blocks which are then populated by sub-scopes and symbols. In order to instantiate a model's scope hierarchy, the *ASTSymbolTableVisitor* was introduced. Finally, the constructed symbol table was used to check the context of the handed over model for correctness. Here, the orchestrating *CoCosManager* class delegated all required checks to individual *context condition* classes, with the result being an AST which has been tested for semantical correctness.

3.4   Assisting Classes

As opposed to the introduction of a typical DSL architecture in chapter 2, where semantical checks, as well as model transformations, were seen as a part of the *function library*, we decided to follow a different approach during the reengineering of NestML. In the previous section, checks for semantical correctness of a given model were already included in the model-processing frontend instead of characterizing this component as an element of the subsystem sitting between the frontend and the code generator. However, the architecture as introduced in Figure 2.2 represents a recommendation and can be adjusted to individual use cases. We, therefore, decided to factor out the functionality normally contained in the function library and instead delegate these components to the model-processing frontend and the generating backend. The result of the frontend should, therefore, be an AST representation of the model which has been checked for semantical and syntactical correctness. Moreover, model transformations are most often of target-platform specific nature, i.e., whenever several target platforms are implemented, it may be necessary to implement several model transformations. As illustrated in Figure 3.21, it is beneficial to regard model transformations as a part of the target-format generating backend and encapsulate all components required for a specific target in a single subsystem. Following these principles, the overall PyNestML architecture has been implemented slightly different as presented in chapter 2: A rich and powerful frontend is followed by a small collection of workflow governing and assisting components, which are in turn concluded by several, independent code generators. In this section we will introduce components sitting in between and governing the overall model-processing control flow and providing assisting functionality. Although not crucial, these elements are often required to provide a straightforward tooling as well as certain quality standards.

As introduced in the previous section, the *ModelParser* class reads in and checks a textual model for syntactical and semantical correctness. However, transforming the model to an equivalent AST is only the first step in the overall processing. Figure 2.2 showed which other steps have to follow and therefore to be chained and governed by an orchestrating
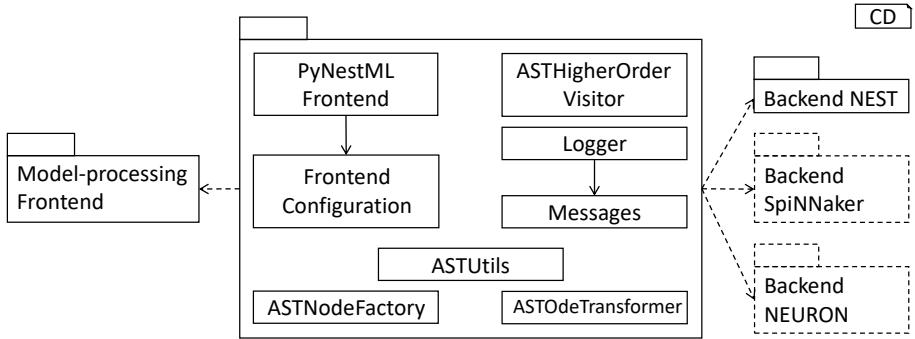
Figure 3.21: Overview of assisting components: The *PyNestMLFrontend* represents an orchestrating component, governing PyNestML's workflow. Parameters handed over by the user are stored in the *FrontendConfiguration*, while the *Logger* and *Messages* classes take care of logging. For the modification and creation of ASTs, the *ASTUtils*, *ASTNode-Factory* and *ASTOdeTransformer* are employed. The *ASTHigherOrderVisitor* represents an assisting component making traversal and modification of ASTs easier.
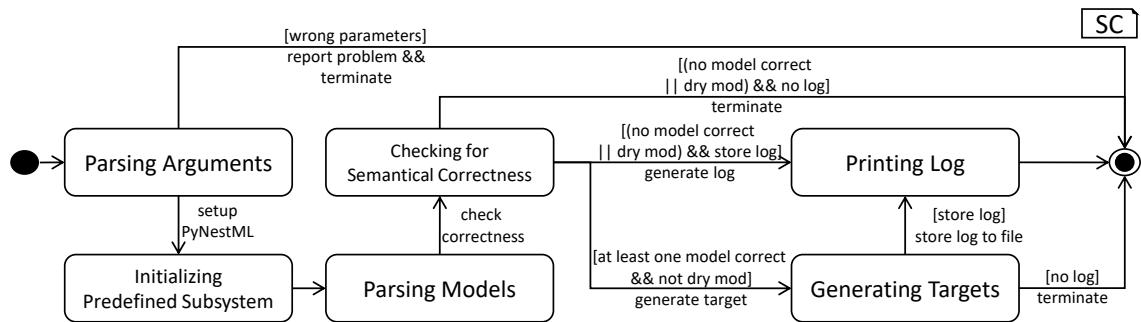
Figure 3.22: The model-processing routine as orchestrated by the *PyNestMLFrontend*.

component. This task is handled by the *PyNestMLFrontend* class, a component which represents the *workflow execution unit* and hides the model transforming process behind a clearly defined interface.

Before the actual processing of the model can be started, it is necessary to handle all parameters as handed over by the user, e.g., the path to the models. These parameters tend to change frequently whenever new concepts and specifications are added. PyNestML therefore delegates the task of arguments handling to the *FrontendConfiguration* class. By utilizing the standard functionality of Python's *argparse*[5] module, the frontend configuration is able to declare which arguments the overall system accepts, cf. chapter 7. The handed over parameters are stored in respective attributes and can be retrieved by the corresponding data access operations. All attributes and operations are hereby static (class properties) and can be accessed from the overall framework by simply interacting with the class. Whenever new parameters have to be implemented, it is only necessary to extend the existing *FrontendConfiguration* class with a new attribute and access operation, while the remaining framework remains untouched.

All arguments as handed over to the *PyNestMLFrontend* class are therefore first delegated to the *FrontendConfiguration* class where all settings are parsed. Only if a valid set of arguments is available, the system proceeds. First, the *predefined* subsystem of the

---

[5]https://docs.python.org/3/library/argparse.html

previous section is initialized. Subsequently, the *ModelParser* class and its *parseModel* method are used to parse the model. This process is executed for all handed over artifacts, with the result being a collection of neuron models represented by ASTs. After all models have been parsed, it remains to check a context condition which is only available whenever several artifacts are processed, namely *CoCoNoTwoNeuronsInSetOfCompilationUnits*. PyNestML checks in the list of all processed artifacts whether two neurons with equal names are present. Although not directly semantically incorrect, this property still has to be ensured. Otherwise, a generated C++ implementation of the respective neuron would overwrite a different one, leading to unexpected results. The corresponding context condition is hereby directly invoked on the *CoCosManager*, cf. section 3.3. All errors are reported and logged by means of the *Logger* class. If the developer mode is off, PyNestML inspects the log and removes all neurons from the current collection which have at least one found error. The adjusted collection is then handed over to the code generating backend. After all models have been processed, the overall log is inspected and stored in a file if required. In conclusion, the *PyNestMLFrontend* class represents the overall *workflow execution unit*, cf. section 2.1, and combines the model-processing frontend and the code-generating backend. Figure 3.22 subsumes the presented procedure.

The *Logger* represents an assisting class which is used in almost all parts of the PyNestML framework. Errors during the parsing and semantical checks as well as all complications arising in the code generators are reported by means of this component. Often identical errors can occur in several parts of the toolchain, e.g., an underivable type in the expression and data-type processing visitors. Whenever these messages have to be adjusted, it is necessary to locate all occurrences and adjust equally in order to preserve consistency. The implementation tackles this problem by storing all messages in a single unit, namely the *Messages* class as shown in Figure 3.23. Each message is encapsulated in a private field and can not be directly accessed. Instead, a corresponding *getter* is used. Consequently, all messages can be changed while the interface remains unaffected. Moreover, the *Messages* class implements an additional feature which makes specific filtering of messages easier to achieve. In order to avoid direct interactions with message strings, each message is returned as a tuple consisting of a string and the corresponding *message code*. The message code is hereby an element of the *MessageCode* enumeration type which provides a wide range of message and error codes. Whenever a getter method of the *Message* class is invoked, a tuple of a message and the corresponding code is returned. Each reported issue can, therefore, be identified by its error code, making filtering of messages by their type or logging level possible.

The *Message* class makes reporting of errors easy to achieve and maintain. The actual printing and storing of reported issues is delegated to the *Logger* class, where all messages are stored together with several qualifying characteristics. In order to filter out messages which are not relevant according to the user, a *logging level* can be set. Messages whose logging level is beneath the stored one are not printed to the screen but may be stored in the optionally generated log file. In order to associate a message with its origin, i.e., the neuron model where the corresponding error occurred, a reference to the currently processed neuron is stored. All messages can therefore also be filtered by their origin.

The corresponding set of operations on the logger represents a complete interface for the storing, printing and filtering of messages. The *logMessage* method inserts a new message into the log and expects the above-mentioned tuple. The *getAllMessagesOfLevel* method returns all messages of a specified logging level, while *getAllMessagesOfNeuron* returns all issues reported for a specific neuron model. The *hasErrors* method checks whether a neuron does or does not contain errors. The final operation of this class is

```
          Logger                                                      CD

 - log
 - currentMessage
 - loggingLevel
 - currentNeuron

 + logMessage(...)                              Messages
 + getAllMessagesOfLevel(...)
 + getAllMessagesOfNeuron(...)       + getStartedProcessingFile(...)
 + printToJSON()                     + getNewTypeRegistered(...)
 ...                                 + getStartBuildingSymbolTable(...)
                                     ...

      <<enumeration>>                       <<enumeration>>
       LoggingLevel                          MessageCode

 INFO                                 START_PROCESSING_FILE
 WARNING                              TYPE_REGISTERED
 ERROR                                START_SYMBOL_TABLE_BUILDING
 NO                                   ...
```
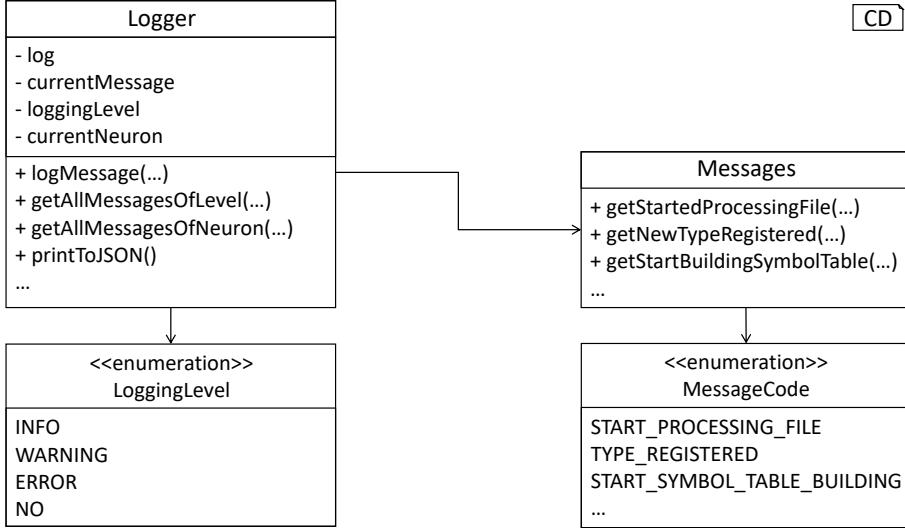
Figure 3.23: The logger and messages components: The *Logger* provides methods for reporting issues (*logMessage*) and precise retrieval of messages (e.g., *getAllMessagesOfLevel*). For a log in file format, the *printToJson* method can be used. In order to make maintenance more focused, all message strings are encapsulated in the *Messages* class. The currently set logging level, as well as individual message codes, are hereby of an enumeration type.

the *printToJSON* method. As introduced in the *PyNestMLFrontend* class, it is possible to store the overall log in a single file. For this purpose, first, it is necessary to create a sufficient representation of the log in JSON format. This task is handed over to the aforementioned method, which inspects the log and returns a corresponding JSON string representation. In conclusion, all methods of this class represent an ideal interface for a troubleshooting and monitoring of textual models.

The *ASTNodeFactory* class implements the *factory* pattern [Gam95] and provides a set of methods used to initialize new AST nodes, while the *ASTUtils* class represents a rather broad collection of operations required across the overall framework. In the case of the latter, especially two methods are of interest: The *isCastableTo* method returns whether a type *X* can be casted to a type *Y*, ensuring that the types of both sides of a given declaration or assignment in the model are equal or at least castable into each other. The *differsInMagnitude* method, on the other hand, returns whether two types represent the same physical unit and only differ in the magnitude. As introduced in section 3.3, both operations are required to ensure that models are regarded as being correct although containing minor typing differences.

Transformations which are especially focused on the *equations* block and its definition of differential equations are contained in the *ASTOdeTransformer* class. Although solely used by transformations contained in the code-generating backend, this class has been decoupled and represents a self-contained unit. Independently of the concrete target platform for code generation, it is often necessary to modify all ODEs in a given model. This class provides a collection of operations for the data retrieval from and manipulation of ODEs. The *getter* functions collect function calls contained in all declared ODEs. The corresponding manipulation operations are marked by the prefix *replace* and can be used to replace certain parts of an ODE by other specifications. Although these operations could also be included in the *ASTUtils* class given their nature of manipulating an AST,

ASTOdeTransformer	ASTNodeFactory	ASTUtils
- functions - sumFunctions	+ createASTInternalBlock(…) + createASTStateBlock(…) + createASTInitialValuesBlock(…)	+ getAllNeurons(…) + isSmallStmt(…) + isCompoundStmt(…)
+ replaceFunctions(…) + replaceSumCalls(…) + getFunctionCalls(…) + getSumFunctionCall(…) …	+ createASTStatement(…) + createASTDeclaration(…) + createASTAssignment(…) …	+ isSpikeInput(…) + isCurrentInput(…) + isCastableTo(…) + differsInMagnitude(…) …

Figure 3.24: AST-manipulating modules: The *ASTOdeTransformer* implements a set of operations focused on the retrieval of information from and modifications of the ODE block. The *ASTNodeFactory* offers operations for the creation of AST nodes, while *AS-TUtils* contains a vast collection of operations on the AST.

for a clearer separation of concerns all operations on the ODE block have been delegated to a single unit. As presented in chapter 4, it is often necessary to adjust a given *equations* block and transform a set of expressions. By encapsulating all operations in a unit, a clear single responsibility and therefore maintainability is achieved. Figure 3.24 summarizes the provided functionality of the *ASTOdeTransformer*.

We conclude this section by an introduction of the *higher-order visitor*, a concept which has been implemented to reduce the amount of code and effort required to interact and modify a given AST. Although highly applicable, this approach can only be employed in programming languages where functions and operations are regarded as objects and can therefore be handed over as parameters to other functions. Luckily, this applies to Python and its concept of duck-typing.

Section 2.1 illustrated that it is often necessary to perform a set of operations on certain types of nodes in a given AST, e.g., whenever all function calls with a specific name and arguments have to be collected. The *visitor* pattern [RH17] provides a possible approach for an implementation of such procedures, where concrete operations and the visiting order are decoupled, cf. Figure 3.25. If one or the other routine has to be modified, the user can simply override the corresponding operation. However, visitors which implement simple operations still require an extension of the base class, making the hierarchy of classes less comprehensible and cluttered. Moreover, in the case that two visitors have to be combined to a single one, it is not directly possible to mix them, but required to implement a new visitor containing both. All this leads to a situation, where maintenance of components is not focused, but distributed over a hierarchy of visitors and their assisting operations, blowing up the code base with unnecessary code and repetitive definitions of new classes.

Especially in the case of PyNestML and its semantics-checking subsystem many visitors had to be written. In order to avoid the above-mentioned problems, the concept of the *higher-order visitor* was developed. Analogously to the (generated) base visitor, this class implements a traversal routine on the AST. However, instead of overriding the base visitor and providing all operations on the AST in a new class, it is only required to hand over a reference to the operation which should be performed on the AST. Coming back to the introductory example: Here, it is only necessary to check whether a node represents a function call, and which arguments it has. Both operations can be stored in a single function definition. The *higher-order visitor* therefore expects such a function reference, traverses the AST and invokes the operation on each node. Other modifications, e.g., which visit a node twice or simply skip it, are directly encapsulated in the corresponding
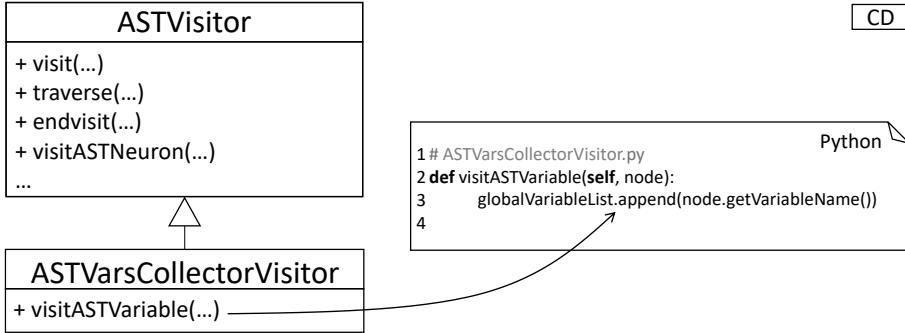
Figure 3.25: The *visitor* pattern in practice [KRV10]: Even small operations, e.g., the collection of certain types of variables, require the usage of sub-classing, where only a single operation is redefined.

function. Utilizing this concept, many obstacles can be eliminated. Simple visitors, e.g., those collecting all variables in a certain block, can be implemented with a single line of code as illustrated in Figure 3.26. The overall code base becomes smaller, while visitors are defined in-place together with their caller, making maintenance easy to achieve and data encapsulation a built-in property.
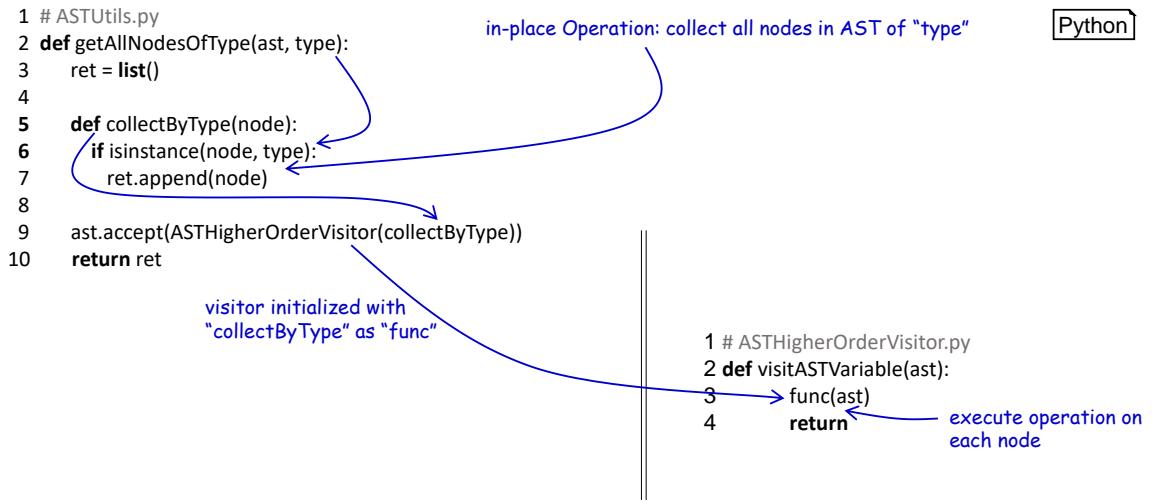
Figure 3.26: The *Higher-Order Visitor*: The *visit* operation is provided by the AST whose subtree shall be visited and the actual operation. This operation can be either declared in-place by *lambda expressions* or as a reference to a different function. The higher-order visitor traverses the tree and invokes the function on each node.

In this section, we presented all assisting classes as contained in the framework:

- *FrontendConfiguration*: A configuration class used to store handed over parameters.

- *PyNestMLFrontend*: A class providing a simple interface to PyNestML.

- *Logger* and *Messages*: A logger with a set of corresponding messages for precise and easy to filter logs.

- *ASTNodeFactory* and *ASTUtils*: Collections of assisting operations as used to create and modify ASTs.

- *ASTOdeTransformer*: A component specialized on manipulating ODE blocks.

- *ASTHigherOrderVisitor*: A visitor which expects a function, which is then executed on each node in the AST. Makes inheritance for simple visitors no longer necessary.

All these components make PyNestML easier to maintain and ensure basic qualities of a software, namely data abstraction, separation of concerns and single responsibility. As we will see in chapter 5, all these characteristics are highly anticipated and make integration of extensions an easy to achieve goal.

3.5   Summary: Model-processing Frontend

In this section we demonstrated how the model-processing frontend of NestML was reengineered and migrated to a new platform. We demonstrated how individual components were implemented and which intentions directed individual concepts. Here, especially the *separation of concerns* and *single responsible* of components had priority: Each subsystem is implemented with the smallest possible interface. Changes on components are focused and *continuity* is given. All introduced components have been developed based on the *Continuous Integration* (CI, [FF06]) and *Test Driven Development* (TDD, [Bec03]) approaches, thus all subsystems, from the lexer and parser to the *ASTSymbolTableVisitor*, are provided with a rich set of tests, automatically executed with each released update. The result of the processes as involved in the frontend is hereby the representation of a textual model by means of an AST, where the semantical correctness of the represented model has been ensured by the *SymbolTable* and a set of *context conditions*. This AST will be used in chapter 4 to create a transformed, target simulator-specific model.

Chapter 4

The Generating Backend

The generation of executable code is one of the most important aspects of a DSL-processing framework and enables the validation of the modeled concepts. The transformation of a textual model to an executable representation by means of a DSL framework prevents a manual, error-prone mapping of models to target platforms. In the case of (Py)NestML, the NEST simulator [GD07] was selected as the first major platform for code generation. NEST represents a powerful simulation environment for biological neural networks and is implemented in C++. In this chapter, we will demonstrate how the code-generating backend was reengineered to generate NEST specific C++ code. For this purpose, section 4.1 will first introduce the orchestrating *NestCodeGenerator* class and subsequently demonstrate how models are adjusted to be more NEST affine. An overview of the components used to generate NEST-specific code concludes this chapter. Figure 4.1 illustrates the subsystems introduced in this chapter and their relations.

4.1   AST Transformations and Code Generation

In order to demonstrate the code-generating backend, this section will first introduce the coordinating *NestCodeGenerator* class and show how the code generation is prepared by transforming the handed over AST to a more efficient form. Subsequently, we highlight a set of templates used for the generation of NEST-specific C++ code. Concluding, an introduction to the special case of expression handling as implemented in the *ExpressionPrettyPrinter* class is given. Figure 4.2 illustrates all components of the reengineered backend.

The *NestCodeGenerator* class orchestrates all steps required to generate NEST-specific artifacts. The overall interface of this class consists of the *analyseAndGenerateNeuron* and *generateModuleCode* methods. By separating the code generation into two different operations, a clear single responsibility is achieved. While all steps necessary to generate the C++ implementation of a neuron model are executed in the *analyseAndGenerateNeuron* method, the task of generating a set of setup artifacts is delegated to the *generateModuleCode* method. The *analyseAndGenerateNeuron* function hereby implements the following steps: First, the assisting *solveOdesAndShapes* function is executed which indicates whether a transformation of the model to a more efficient structure is possible. If so, the AST is handed over to the further-on presented *EquationsBlockProcessor* class, cf. Figure 4.4, and a restructured AST is computed. Back to the orchestrating *analyseAndGenerateNeuron* method, an update of the symbol table is invoked by means of the *ASTSymbolTableVisitor*, cf. section 3.3. This step is required in order to update
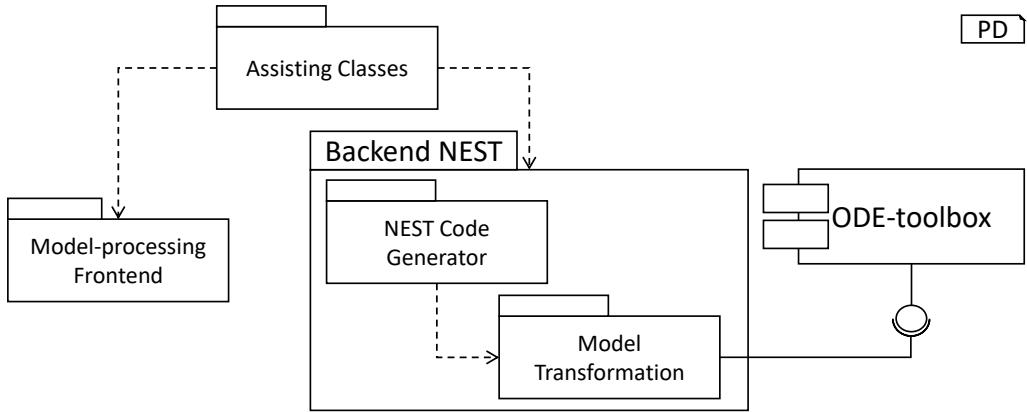
Figure 4.1: Overview of the code-generating backend: The model-processing frontend provides an input AST for the code generation. The NEST-specific backend first transforms the AST by means of the *model transformation subsystem*, before the *NEST code generator* is used to generate the respective C++ code. The instructions how the AST has to be adapted are computed by an external ODE-toolbox.
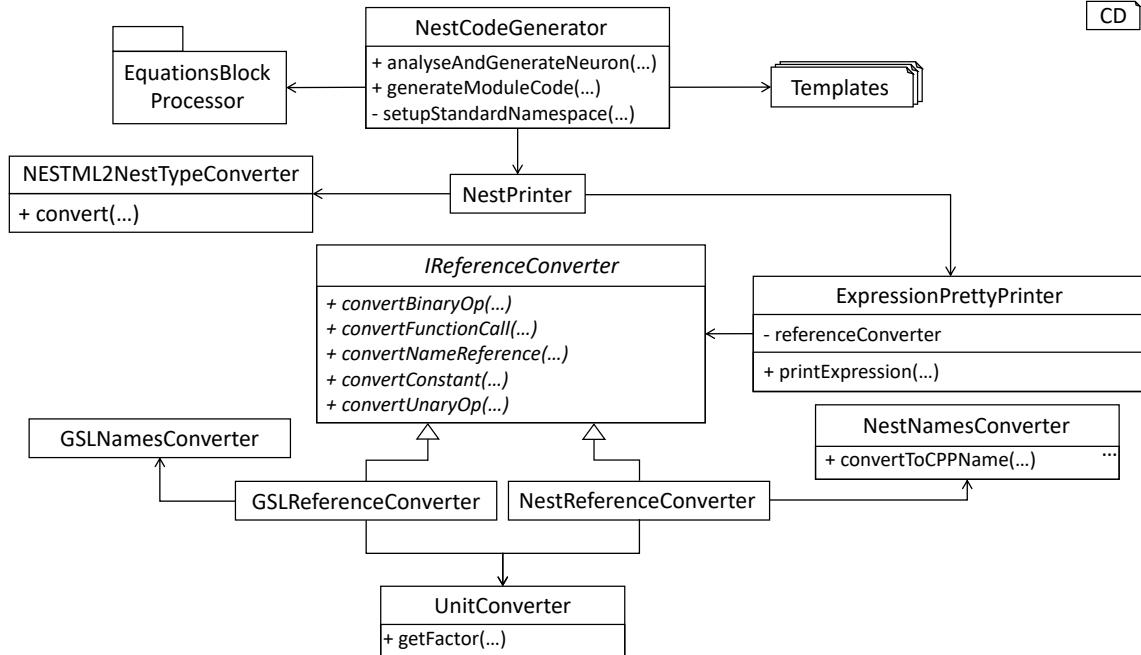

Figure 4.2: Overview of the NEST code generator.
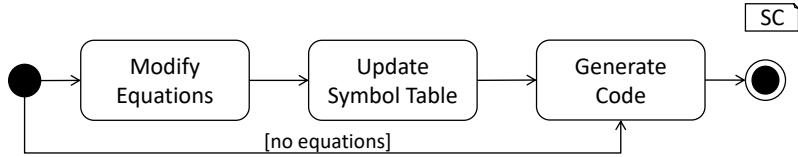
Figure 4.3: Processing of a model in the NEST backend.

the model's symbols according to the restructured AST where new declarations have been added. Finally, the generation of C++ code is started by means of the *generateModelCode* method. Responsible for the generation of a header as well as an implementation file of a concrete neuron model, this operation delegates the work to the *generateModelHeader* and *generateModelImplementation* subroutines. Figure 4.3 summarizes the above introduced workflow.

Depending on the selected simulator or environment different concepts are supported. This circumstance has to be regarded whenever code is generated. While simulation environments such as *LEMS* [CGC+14] support physical units as an integral part of the simulation, others such as NEST do not. In order to avoid unsupported declarations of models, an AST-to-AST transformation is implemented which restructures the source model to a target platform supported format. Besides missing support for certain concepts, also an optimization of declared models is often of interest. Transformations, therefore, enable a DSL framework to adjust models to specific targets and generate efficient code.

In the case of PyNestML, all transformations of neuron models are focused on the *equations* block, where, depending on the stated declarations, models are restructured and definitions transformed to a more efficient and easy to generate form. The target simulation environment NEST utilizes the *GNU Scientific Library* (GSL, [Gou09]) for the evaluation and integration of differential equations. GLS expects a special form of the *equations* block where only *ordinary differential equations* (ODEs) with their respective starting values have been declared. Such a form enables an efficient computation and handling of neuron spikes. For models which contain declared *shapes* it is therefore necessary to compute an exact solution where the *equations* block evolution is replaced by direct computation steps. In cases where such an optimization is not possible, at least a transformation of all *shapes* to equivalent representations by means of ODEs and initial values shall be computed. Such a form of the neuron model avoids time-consuming evaluation of *shapes* for each time-step t. To summarize, the first major task of the code generator is to perform an AST-to-AST transformation where the *equations* block is replaced by an exact solution or all *shapes* have been converted to ODEs and initial values. All this helps to *normalize* the generated code and therefore to ease its evaluation.

In order to compute these optimizations, the ODE-toolbox as introduced by Blundell et al. [BPEM18] is integrated. Written in Python, the ODE-toolbox can be used in a black box manner to restructure a stated *equations* block to a less computationally expensive form. Amongst others, it features concepts for a derivation of exact solutions, elimination of computationally expensive *shapes*, and constants folding. For an interaction with this tool, it is first necessary to convert the *equations* block to a representation processable by the ODE-toolbox, and subsequently, integrate the computed new formulation of the ODE or an exact solution into the source AST. The referenced ODE-toolbox is implemented in an environment-agnostic manner, where an exchange of data with the toolbox is performed over the platform-independent *JSON* format [NPRI09]. Before the ODE-toolbox can be used, it is therefore first necessary to create a representation of a model's properties in
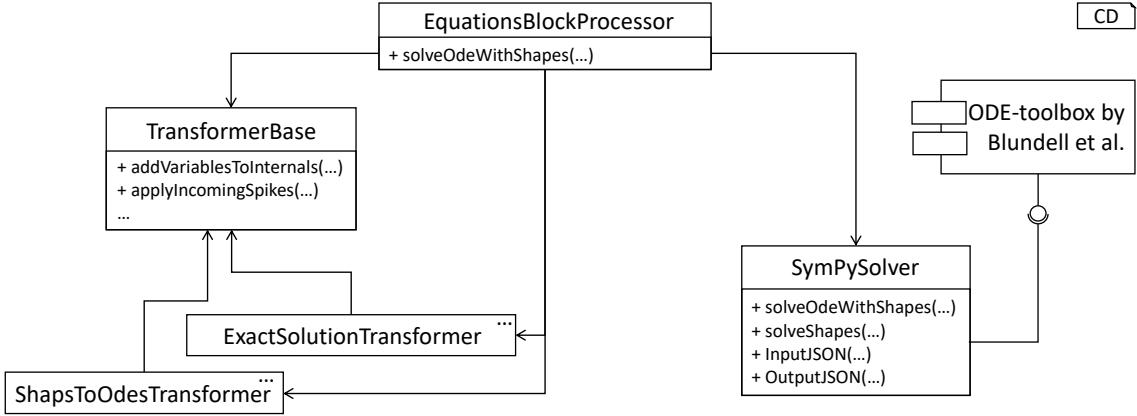
Figure 4.4: The model transformation subsystem: The *EquationsBlockProcessor* receives a neuron model. The *equations* block is extracted and handed over to the ODE-toolbox by means of the *SymPySolver* wrapper class. The returned result is finally processed by the *transformers* and integrated into the AST.
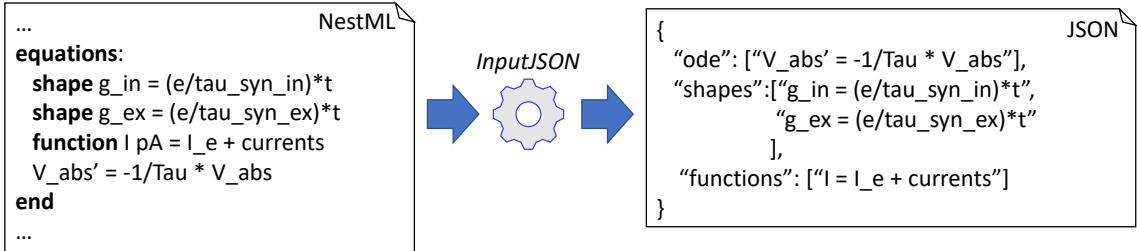
Figure 4.5: From NestML to JSON: In order to interact with the ODE-toolbox, all declarations contained in the *equations* block are converted to JSON format.

JSON format. Such a handling makes the used ODE-toolbox an exchangeable component, where only the wrapper converting and exchanging data has to be adjusted whenever a different toolbox is used. PyNestML delegates the interaction with the toolbox to the *SymPySolver* class. Summarizing, the overall process as employed in this component can be described as follows: Given an *equations* block, print its specifications to an equivalent JSON string. Hand over the generated JSON object to the ODE-toolbox and finally invoke the optimizing routine. Afterwards, the computed results are integrated into the AST by the *EquationsBlockProcessor* class and its assisting components. Figure 4.4 illustrates the AST-transforming part of the NEST code generator.

The task of creating a JSON representation of a given *equations* block is handled by the *InputJSON* method. The purpose of this operation is to analyze the *equations* block, print all components to a processable format and finally restructure it into a correct JSON string. This function retrieves three different types of equation specifications as definable in the *equations* block, namely all *shapes*, *functions* and *equations*. Instead of handing over an AST to the ODE-toolbox, all expressions are first printed by means of the *ExpressionPrettyPrinter* class, cf. Figure 4.2, to a Python-processable format. By exchanging strings instead of objects, a better control and comprehension of all side effects is achieved. For all three types of declarations in the *equations* block, PyNestML implements an additional printing routine: The *printEquation* function retrieves the name

54
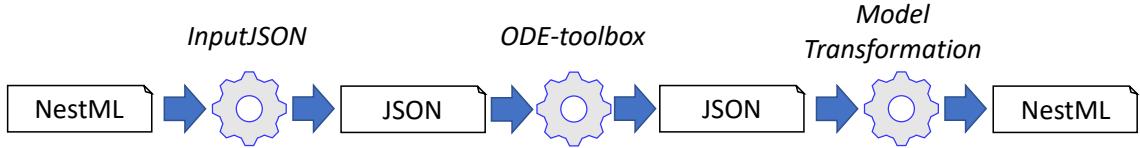
Figure 4.6: Interaction with the ODE-toolbox: Stated declarations in the source model are transformed to an equivalent representation in JSON format and handed over to the ODE-toolbox. The computed modifications are de-constructed from JSON format to a collection of individual definitions and integrated into the model.

of the left-hand side variable together with the differential order and combines it with the right-hand side expression printed by the *ExpressionPrettyPrinter*. This procedure is executed analogously for *shapes* and *functions*. Finally, it remains to combine the stored strings to a valid JSON format. The *InputJSON* function therefore iterates over the stored strings and combines them by means of a correct syntax as illustrated in Figure 4.5. The result of the process as implemented in this function is a JSON string encapsulating all *equations* block specifications in a format processable by the ODE-toolbox.

Having a representation of the equations block in an appropriate string format, PyNestML starts to interact with the ODE-toolbox. The concrete communication is hereby delegated to the orchestrating *SymPySolver* class. This component represents a wrapper for the ODE-toolbox and executes all steps as required to communicate with the toolbox and convert the input and output to appropriate formats, cf. Figure 4.6. The input format is hereby encapsulated in a JSON string as constructed by the *InputJSON* function, which is subsequently handed over to the *compute-solution* operation of the ODE-toolbox. The result of this operation is a set of modified declarations where certain parts have been replaced or simplified, e.g., *shapes* represented by ODEs and initial values. Analogously to the input, the output as returned by the toolbox is also represented by means of a string in JSON format. It is, therefore, necessary to parse the modified declarations and inject them into the currently processed AST. In order to make the overall processing modular and easy to maintain, PyNestML implements the *OutputJSON* function which is solely used to de-construct a JSON string to a collection of individual elements. The actual processing and injection of computed ODE declarations into ASTs is delegated to the *TransformerBase* and its assisting classes.

The *OutputJSON* function returns a dictionary of fields for different declarations as computed by the ODE-toolbox. All fields store the modified ODE declarations as a string, while the actual parsing is executed by subsequent components. The *status* field, for instance, indicates whether any problems occurred during the *equations* block processing. The remaining fields analogously define other properties which can be added by the ODE-toolbox, e.g., new state variables and differential equations. The decomposed output as stored in the dictionary can now be used to perform an AST-to-AST transformation.

Having an optimized structure of the *equations* block, PyNestML starts to transform the AST. Here, depending on the type of the returned solution, a different handling is required. However, which handling is concretely executed should not be a concern of PyNestML, but rather selected according to the toolbox output. This routine is therefore implemented in the *EquationsBlockProcessor* class which encapsulates all steps of the transformation in a single method. Consequently, whenever it is required to analyze a given model and transform it according to the computed modifications, the functionality as contained in this class is used. The underlying processing is hidden and therefore easy to exchange and
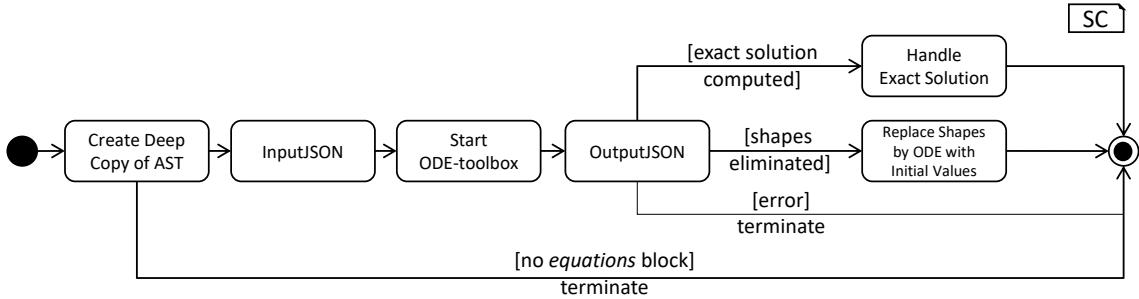
Figure 4.7: The model-transforming process.

maintain.

The transformation of a neuron model can be invoked by means of the *solveOdeWithShapes* method of the *EquationsBlockProcessor*. This operation expects a single neuron model and performs a series of steps as illustrated in Figure 4.7. First, a new deep copy of the processed AST is created. Potentially having several targets for code generation with individual transformations, each backend transformation should work on a local copy instead of modifying a global one. Without creating a local working copy, each modification would be visible to all implemented backends, possibly preventing correct processing whenever a transformation is not appropriate for a given target. Subsequently, the routine checks whether an *equations* block is present. Obviously, no modifications are required if no equations are given, thus the operation terminates and returns the current working copy. Otherwise, the content of the neuron's *equations* block is delegated to the previously introduced *SymPySolver* class. Depending on the results as returned by the ODE-toolbox, a different handling is employed: In the case that at least one *shape* and exactly one equation are contained in the textual model, the toolbox is most often able to compute an exact solution. Computed modifications of this type contain new variables and assignments, thus the task to transform the processed working copy is delegated to the *ExactSolution-Transformer* class, cf. Figure 4.4. Expecting a JSON string, this class parses and injects all returned modifications into the processed AST. In cases where a given *equations* block contains only *shapes*, the ODE-toolbox tries to derive a solution where *shapes* are replaced by *equations* and *initial values*, making the computation less time and resources consuming. The corresponding adaption of the AST is delegated to the *ShapesToOdesTransformer* class which replaces *shapes* by their computed ODE counterpieces. The *ExactSolutionTransformer* and *ShapesToOdesTransformer* classes hereby import the assisting *TransformerBase* class. This component contains general functionality as required to process both types of returned solutions, e.g., the *applyIncomingSpikes* method which replaces all *convolve* function calls in the *equations* block by concrete update instructions, e.g., assignments of values stored in buffers to state-variables. For certain types of declarations, the ODE-toolbox by Blundell et al. is not able to derive a more efficient solution [BPEM18]. In these cases, the NEST simulator performs a time consuming numeric integration of the unmodified *equations* block. Not supported declarations as well as errors during the *equations* block processing are hereby indicated by the *status* field of the JSON object as returned by the toolbox. In this case, the local working copy of the AST is not further modified but simply returned to the code-generating subsystem. As previously stated, the overall processing implements a transformation which is specific to the NEST simulator. However, other backends may also reuse parts of the presented classes. Consequently, all concrete transformations as implemented in the *ExactSolutionTransformer*,

56
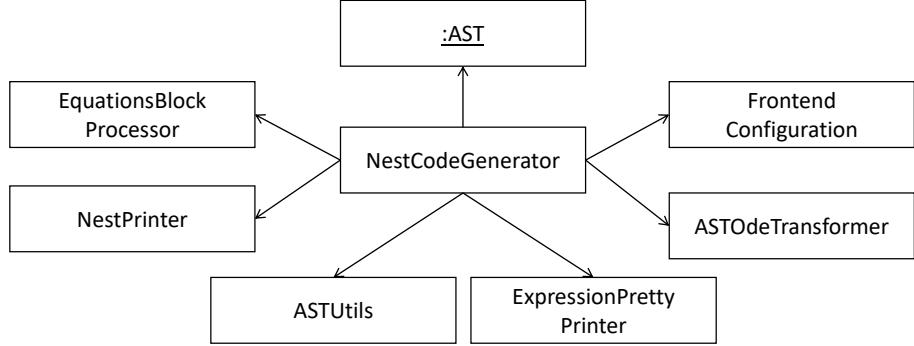
Figure 4.8: The *NESTCodeGenerator* class and assisting components.

ShapesToOdesTransformer as well as the *TransformationBase* class have been summarized in a dedicated module.

The optimized representation of the source model is returned to the orchestrating *analyseAndGenerateNeuron* method of the *NestCodeGenerator* class. Here, it is first prepared for the code generation by retrieving general characteristics and setting up a generation context which states, e.g., whether a *spike* buffer is contained in the model. Subsequently, a template engine and a set of templates are used to generate model-specific C++ code. The result of this step is an executable representation of a source model as well as a set of additional artifacts which can now be used to integrate the neuron model into the NEST simulator.

Jinja2 as well as many other template engines often do not directly interact with the AST, but follow a more general concept by operating on a *generation context*. Such a context consists of a map from identifiers to objects, methods and other properties. For instance, if the generating routine has to be able to interact with the *ASTUtils* class, it is required to create a dictionary mapping a unique identifier to an *ASTUtils* class reference. This identifier can then be used in the context of the template to interact with the corresponding object. Before the code generation is invoked, it is therefore first necessary to set up a generation context. In the case of PyNestML, this context consists of several processed objects as well as assisting classes, cf. Figure 4.8. For the sake of modularity, the creation of an appropriate context is delegated to the *setupStandardNamespace* function which instantiates a generation context according to the handed over AST.

Having a set up context, the *NestCodeGenerator* initiates the actual code generation by invoking the *render* operation on the further on introduced templates, with the result being a set of generated C++ artifacts as illustrated in Figure 4.9. In order to enable an easy to achieve integration of the generated C++ code into the NEST infrastructure, PyNestML implements a concept for the generation of setup files. By utilizing predefined extension points of NEST, new neuron models can be integrated into the simulation environment by means of a corresponding module file. The task of generating these artifacts is delegated to the *generateModuleCode* procedure. Except for a different set of templates, this method behaves analogously to the above-introduced *generateModelCode* procedure. After all model-specific as well as setup artifacts have been generated, the control is returned to the PyNestML workflow unit.

As demonstrated in section 2.1, often target implementations can be described in a schematic way by means of a template, where placeholders are replaced by model-specific details in order to get executable code. These templates represent a major component of a code generator and are used by the above-introduced routines, e.g., the *generate-*
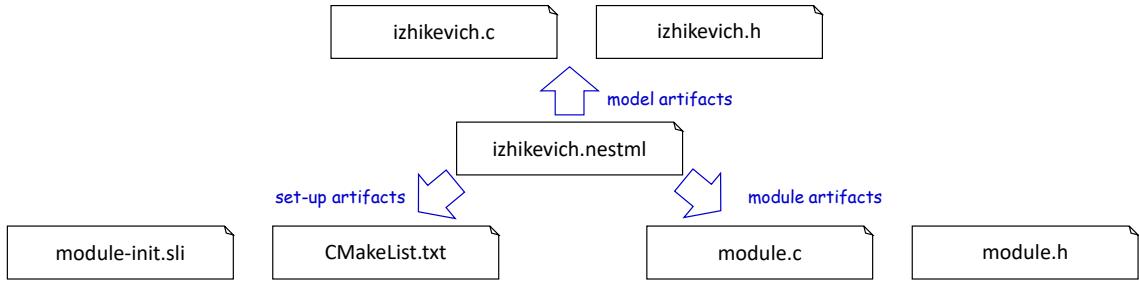
Figure 4.9: Generated artifacts of the *Izhikevich* neuron model.
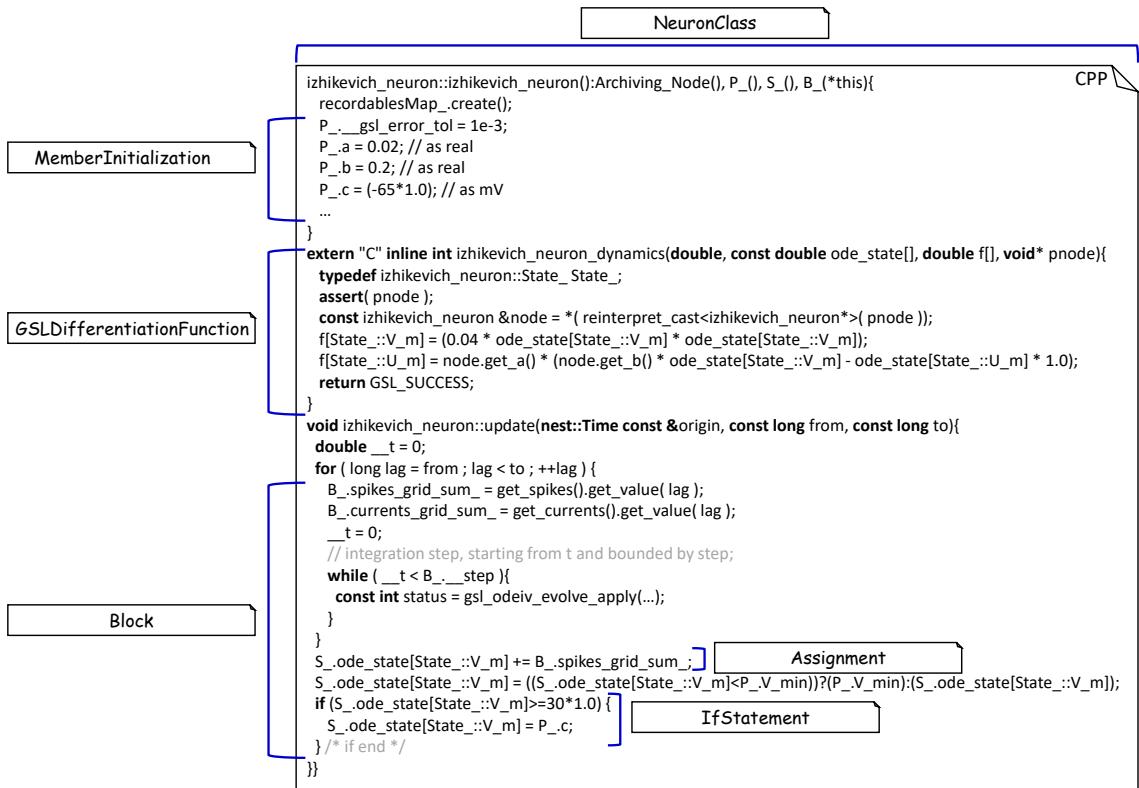
Figure 4.10: Templates and the generated code of the *Izhikevich* neuron model.
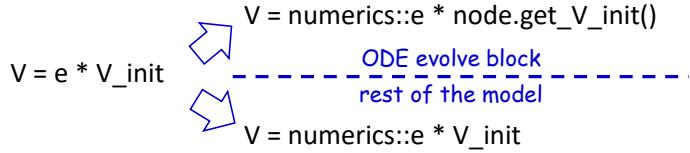
Figure 4.11: Context sensitive target syntax.

*ModelHeader* method. The implemented NEST backend employs six governing templates and a set of assisting sub-templates. Models of neurons are generated by means of the *NeuronHeader* and *NeuronClass* template, while the generation of a model integration file is delegated to the *ModuleHeader* and *ModuleClass* templates. The generation of setup files is delegated to the *SLI_Init* and *CMakeList* templates. Figure 4.10 exemplifies how templates are used by means of generated C++ code. The processing as executed by the generator engine involves a retrieval of data from the model's AST and the symbol table, and a replacement of placeholders in the evaluated template. All required declarations are hereby extracted from the AST by the corresponding *getter* operation, e.g., *getStateSymbols*, and stored in C++ syntax.

While templates, in general, are able to depict an arbitrary syntax, their usage can become inconvenient whenever many cases have to be regarded and conditional branching occurs. This problem becomes more apparent when dealing with expressions: While the overall form of the AST is restructured to be more NEST affine, individual elements remain untouched and are still represented in PyNestML syntax. However, certain details such as the used physical units are not supported by NEST. It is therefore required to transform atomic elements such as variables and constants to an appropriate representation in NEST. Moreover, in a single model it may be necessary to represent a certain element in different ways, e.g., Figure 4.11. Consequently, it is not possible to simply modify the AST to use appropriate references and definitions. PyNestML solves this problem by using an ad-hoc solution as implemented in the *ExpressionPrettyPrinter* class. Mostly used whenever expressions have to be printed, this class is able to generate a handed over AST object in a specified syntax. Similar to the type deriving routine, cf. section 3.2, the *ExpressionPrettyPrinter* class first descends to the leaves of a handed over expression node. Subsequently, all leaf nodes are printed to a target-specific format, before being combined by counterpieces of the stated operators. This process is executed until the root node has been reached. The returned result is then used to replace a placeholder in the template by a string representation of the expression.

The key principle of the *ExpressionPrettyPrinter* class is its composable nature: While the *ExpressionPrettyPrinter* only dictates how subexpressions and elements have to be printed and combined, the task to derive the actual syntax of elements and operators is delegated to so-called *reference converters*. Implementing the *template and hook* pattern [VHJG95], here it is possible to utilize different reference converters to print elements and operators into a different syntax. Figure 4.12 demonstrates how expressions are transformed to a string representation by utilizing the above-introduced routine.

The abstract *IReferenceConverter* class declares which operations concrete reference converter classes have to implement. Besides converting functions for binary as well as unary operators, it is also necessary to map variables, constants and function calls. All these elements are therefore provided with their respective *convert* functions expecting an AST node of a corresponding type. The *ExpressionPrettyPrinter* class hereby stores a reference to the currently used reference converter, which is then used to convert the above-
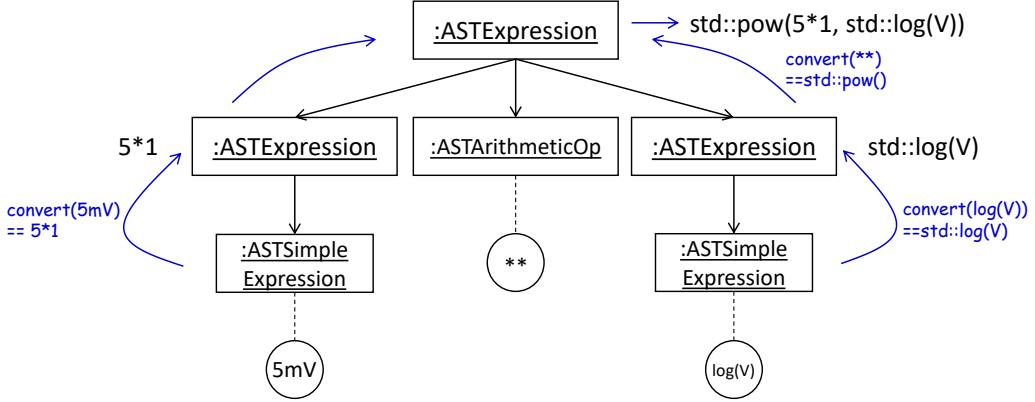
Figure 4.12: From *ASTExpression* to a string.

mentioned elements. The separation of a reference converter and the pretty printer leads to an easily maintainable and extensible system: Similar to the visitor pattern, cf. section 3.1, where only the *visit* method has to be adjusted, here the user can simply replace or extend the reference converter without the need to modify the overall printing routine. Moreover, the code-generating routine becomes composable, where the implemented pretty printer can be independently combined with different reference converters.

The *NESTReferenceConverter* is the first concrete implementation of the *IReferenceConverter* class and is used whenever concepts of NestML have to be converted to those in NEST. Being used in almost all parts of the provided templates, this class features a conversion of operators and constants to their equivalents of the NEST library. As illustrated in Figure 4.12, each element of a given expression is inspected individually and a counterpiece in NEST is returned, making the generated code semantically correct and references valid. The *GSLReferenceConverter* class implements the handling of references which is only required in the context of *equation* blocks. NEST utilizes GSL for the evolvement of equations. Consequently, references as stated in the *equations* block have to resolve to elements of GSL. The *GSLReferenceConverter* hereby inspects the handed over element and returns the respective counterpiece. If a mapping is not defined, the element is simply returned without any modifications.
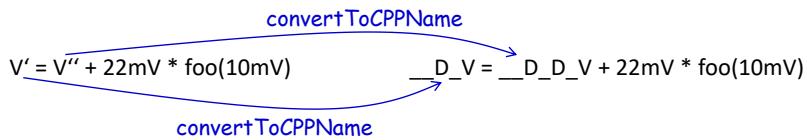
Figure 4.13: Adaption of syntax by the *convertToCPPName* method.

C++ as well as many other languages do not support the apostrophe as a valid part of an identifier. Consequently, variables stated together with their differential order can not be directly generated as C++ code. PyNestML solves this problem by implementing an on-demand transformation of names, executed whenever a variable is processed during code generation. In the case that the name of a generated element contains an invalid literal, PyNestML employs the *convertToCPPName* operation which prefixes a variable for each stated order by the letter *D*, cf. Figure 4.13, resulting in a valid C++ syntax. Moreover, as illustrated in Figure 4.10, generated code features information hiding where attributes of objects and classes can only be accessed by the corresponding data access operations. Together with the *convertToCPPName* function, a conversion of names and references

to their respective data access operation is implemented in the *NestNamesConverter*, respectively *GSLNamesConverter* class for the processing of equations. Both elements are accessed during code generation and the usage of the *ExpressionPrettyPrinter* class.

NestML	NEST
integer	int
real	double
string	string
void	void
boolean	bool
buffers	RingBuffer
SI units	double

Figure 4.14: Mapping of NestML types to NEST.

The second type of assisting component, namely the *NestPrinter* class, is used across the overall backend and implements several methods as often required. The *printOrigin* method, for instance, states from which type of block the corresponding variable or constant originates. Depending on the origin, a different prefix is attached, e.g., S_- for state or P_- for parameters. Such a handling is required given the fact, that all attributes in the generated code are stored in *structs* [Sch98] of their respective types. By prefixing an element's name by a reference to its structure, the correctness of generated code is preserved.

The *NESTML2NestTypeConverter* class provides a mapping from NestML types to appropriate types in C++, cf. Figure 4.14. It should be noted that NestML buffers represent variables and consequently have to be declared with a respective type. For this purpose, NEST's implementation of the *RingBuffer* is used as the corresponding counterpiece. Whenever an element is generated, the functionality contained in the *NESTML2NestTypeConverter* class is used and an appropriate NEST type is returned.

Symbol	Name
ms	milliseconds
pF	picofarad
mV	millivolt
pA	picoampere
nS	nano siemens
MOhm	mega ohm

Figure 4.15: Common neuroscientific physical units.

In the case of physical units additional handling is required. NEST assumes that only a restricted set of physical units, the so-called common neuroscientific units as illustrated in Figure 4.15, are used. In the case that a given constant or variable utilizes a physical unit, the corresponding C++ code is generated without any units and only the numeric part is regarded. Nonetheless, to preserve semantical equivalence of the generated code and the source model, the scalar of a unit is derived in the following manner: In the case that an atomic unit is given, e.g., mV, PyNestML checks whether it is a common neuroscientific unit or not. If so, the neutral scalar *1* is returned. Otherwise, the value is scaled in relation to its common neuroscientific unit, e.g., V is converted to mV and the scalar 1000 is returned. In the case that a compound unit is used, e.g., $mV * s$, the evaluation is executed recursively and all scalars combined. Figure 4.16 illustrates this procedure. The

UnitConverter class implements a routine which is able to perform these steps and scale values according to their physical units. This component is invoked during the generation of expressions and declarations to C++ code, and preserves semantic equivalence of the initial model and the generated code.
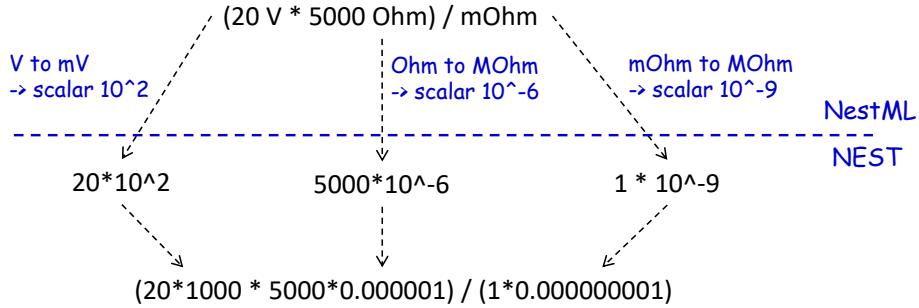
Figure 4.16:   The conversion of physical units from PyNestML to NEST.

However, a mapping of physical units to their respective scalars is not bijective. For instance, the scalar *1000* in a generated expression could originate from the unit *volt* or *second*, or be a simple scalar stated in the source model. Such a handling makes troubleshooting of generated code complex where the origin of an element is not directly clear. This problem is solved by the *IdempotentReferenceConverter* class, a component which implements a simple *identity mapping*, i.e., all elements are converted to themselves. This class is used during the generation of a model's documentation where all variables, types, as well as references, are generated in plain NestML syntax.

Together with the above-presented set of assisting classes, the functionality as implemented in the *ExpressionPrettyPrinter* class enables PyNestML to print complex expressions and other declarations without utilizing templates with cascaded branching and sub-templates for the generation of atomic parts, e.g., function calls. The result is an easy to maintain set of components, where complexity is distributed across several subsystems and no *god* classes or templates [Rie96] are used.

4.2   Summary: The code-generating Backend

We conclude this chapter by a brief overview of the implemented routines as well as the performed refactoring steps. Section 4.1 demonstrated how NEST-specific C++ code can be generated from an optimized AST. Here, we first introduced the coordinating *NestCodeGenerator* class and showed how code generation is prepared. To this end, we outlined how declarations of models can be optimized by restructuring the *equations* block to a more efficient form. The computation of the optimizations is hereby delegated to the ODE-toolbox by Blundell et al. In order to integrate the results as returned by the toolbox, we implemented the *EquationsBlockProcessor* and its assisting classes. Together, these two components yield a more efficient definition of a model. Subsequently, we highlighted a set of templates used to depict the general structure of generated C++ code. In order to reduce the complexity in the used templates, PyNestML delegated the task of generating expressions to the *ExpressionPrettyPrinter* class. Together, these components implement a process which achieves a *model to text* transformation on textual models.

PyNestML has been developed with the intent to provide a base for future development and extensions. As we demonstrated in section 4.1, the transformation used to construct NEST-affine and efficient code has been called from within the NEST code generator as

a preprocessing step. Future backends for target platform-specific code generation can, therefore, implement their individual and self-contained transformations, while all backends receive the same, unmodified input from the frontend. Individual modifications of the AST can be easily implemented as composable filters in the AST processing pipeline. Nonetheless, some of the model optimization steps are of target platform-agnostic nature, e.g., simplification of physical units, and are therefore implemented as a target-unspecific component in the workflow. Moreover, the key principle of the *ExpressionPrettyPrinter*, namely its composability by means of reference converters, represents a reusable component which can be used for code generation to arbitrary target platforms. All this leads to a situation where extensions can be implemented by composing existing components.

Chapter 5

Extending PyNestML

As typical for all types of software, requirements of the implementation often change. PyNestML was implemented with the aim to provide a modular and easy to extend framework which can be adjusted and reconfigured by exchanging components, e.g., context conditions and reference converters. In this chapter, we will briefly demonstrate how extensions to PyNestML can be implemented. Representing components which are often adapted, the following use cases are introduced:

- Grammar: How can the grammar artifacts be extended and in consequence which components have to be adapted?

- Context Conditions: How can new semantical rules be introduced?

- Code Generation: How can the code generator be extended?

All three scenarios represent use cases which often occur when new types of supported concepts are introduced.

5.1 Modifying the Grammar

The following (hypothetical) use case illustrates the extension of the grammar: A new type of block shall be introduced. Declaring constraints which have to hold in each simulation step, this block contains boolean expressions representing invariants of the neuron model. It is therefore first necessary to extend PyNestML's grammar to support a new type of blocks. Figure 5.1 illustrates how a new grammar rule is introduced to support this use case.

The grammar artifacts represent the starting point of each DSL. Consequently, all modification to the grammar have to be propagated to components which depend on its structure, namely:

- The lexer and parser used to parse a model to a parse tree.

- The AST classes storing details retrieved from the parse tree.

- The base visitor as well as the *ASTBuilderVisitor* classes.

- The symbol table building visitor as encapsulated in the *ASTSymbolTableVisitor*.

In section 3.1 we introduced how a manual implementation process of the lexer and parser can be avoided by utilizing Antlr. By executing Antlr on the modified grammar artifact, an

```
…                                                                                    PyNestML.g4
body: BLOCK_OPEN
      (NEWLINE | blockWithVariables | equationsBlock | inputBlock | outputBlock | updateBlock | function | invariantBlock)*
      BLOCK_CLOSE;
…                                                              include new block type
invariantBlock: 'invariant' BLOCK_OPEN
                (expression NEWLINE)*
                BLOCK_CLOSE;
…
```

Figure 5.1: New grammar rules: In order to include a new grammar rule, the existing *body* production is extended by a reference to the extension. The *invariantBlock* production encapsulates the added concept.

```python
1  # ASTBuilderVisitor.py                                              Python
2  …
3  def visitASTBody(self, ctx)
4      body_elements = list()
5      if ctx.invariantBlock() is not None:
6          body_elements.append(self.visit(ctx.invariantBlock() ))
7          …
8      sourcePos = …
9      body = ASTNodeFactory.makeASTBody(body_elements, sourcePos)
10     return body                                          refer to new rule
11
12
13 def visitASTInvariantBlock(self, ctx):
14     expressions = list()
15     for expr in ctx.expression():                        collect all invariants
16         expressions.append(self.visit(expr))
17     sourcePos = …
18     invariantBlock = ASTNodeFactory.makeASTInvariantBlock(expressions, sourcePos)   return
                                                                                       new node
20     return invariantBlock
```

Figure 5.2: Modifying the AST builder: In order to initialize an AST according to the new grammar, the *ASTBuilderVisitor* is extended by an *ASTInvariantBlock* node building method. An adaptation of the existing *visitASTBody* method includes the new rule.

implementation of the lexer and parser adapted to the extensions is generated. Together, these components are used to create the parse tree representation of a model. Proceeding, it is now necessary to provide a mutable data structure which is able to hold details retrieved from the parse tree. A new *ASTInvariantBlock* class is therefore implemented which holds all details of the new rule. As shown in Figure 5.1, each invariant block consists of a set of expressions. Consequently, the *ASTInvariantBlock* class features an attribute which stores lists of *ASTExpression* objects. Together with a set of data retrieval and modification operations, this class represents a data structure which is able to hold all invariants of a neuron model.

Having a modified meta model, it remains to adapt PyNestML to retrieve invariants from the parse tree. PyNestML delegates the initialization of an AST to the *ASTBuilderVisitor* class, cf. section 3.1. Figure 5.2 illustrates how the AST-building routine has to be adapted to regard the new *invariant* block. Here, it is also necessary to extend the existing *visitASTBody* rule to include the instantiation of *ASTInvariantBlock* nodes. With the modified structure of an AST where a new type of node has been added, it is also necessary to adapt the *ASTVistor* class. Implementing a basic traversal routine on the AST, here it is crucial to include an additional traversal method for the new type of AST node as well as the corresponding visit routine. Both methods can then be extended in concrete

```python
1   # ASTVisitor.py                                    Python
2   …
3   def visit(self, node)
4       if isinstance(node, ASTInvariantBlock):
5           self.visitASTInvariantBlock(node)
6       …
7
8   def traverse(self, node):
9       if isinstance(node, ASTInvariantBlock):
10          self.traverseASTInvariantBlock(node)
11      …
12
13  def endvisit(self, node):
14      if isinstance(node, ASTInvariantBlock):
15          self.endvisitASTInvariantBlock(node)
16      …
17
18
19
20
21  …
22  def visitASTInvariantBlock (self, node):
23      pass
24
25  def traverseASTInvariantBlock(self, node):
26      for expr in node.getExpressions():
27          self.visit(expr)
28
29  def endvisitASTInvariantBlock(self, node):
30      pass
31  …
32
33
34      provide monomorph hook methods
```

extend dispatcher methods

Figure 5.3: Modifying the AST visitor: The *ASTVisitor* class is adapted to support the new type of AST node. The dispatcher functions are adapted, while new monomorphic hook methods are added.

visitors in order to interact with the *invariant* block. As illustrated in Figure 5.3, all extensions are focused in a small set of methods. Besides a modification of the dispatcher methods, individual monomorphic functions are added.

An initialized AST represents a base for further checks and modifications. Section 3.3 illustrated how semantical checks are implemented by means of a symbol table and a set of context conditions. With a new type of block, it is therefore necessary to adapt the symbol table building routine. Extending the *ASTVisitor* class, all modifications are focused in the *ASTSymbolTableVisitor*. Figure 5.4 illustrates how the symbol table construction routine has to be adapted. Together, these steps enable PyNestML to parse a model containing the new *invarinat* block, construct the respective AST and populate the symbol table with all required details.

5.2 Adding Context Conditions

Whenever a DSL is extended by new concepts, it also becomes necessary to regard additional semantic rules. In the case of the *invariant* block, it is essential to ensure that only boolean expressions have been stated in this type of block. With an initialized AST, this property can be easily checked by a new context condition. Whenever new semantic rules are established, it is therefore necessary to implement the following adaptation:

- A new context condition implementing all required context checks.

- A modification of the coordinating *CoCosManager* class.

In order to achieve modularity, each context condition is encapsulated in an individual class. The new *CoCoInvariantBlockCorrectlyTyped* class therefore implements all processes as required to check the handed over AST for correctness. Concrete checks are delegated to the *InvariantTypeCheckVisitor* class. Extending the *ASTVisitor*, this class implements a routine which visits the *ASTInvariantBlock* node of a given AST and iterates over all stated expressions. Section 3.2 illustrated a preprocessing of the AST where the types of all expressions have been derived. It therefore only remains to check whether

```python
1  # ASTSymbolTableVisitor.py                                    Python
2  …
3  def traverseASTBody(self, node):
4    for elem in node.getBodyElements():
5      if isinstance(elem, ASTInvariantBlock):
6        self.visitASTInvariantBlock(elem)        ⎤ dispatch new type of node
7      if isinstance(elem, ASTUpdateBlock):         ⎦
8        self.visitASTUpdateBlock(elem)
9        …
10  …
11  def visitASTInvariantBlock(self, node):
12    for expr in node.getExpressions():
13      expr.updateScope(node.getScope())
14      …
15  …
```

Figure 5.4: Adapting the *ASTSymbolTableVisitor*: The *traverseASTBody* method is extended to regard the new type of block, while the actual handling of the block is delegated to the *visitASTInvariantBlock* method.

```python
1  # CoCoInvariantBlockCorrectlyTyped.py                        Python
2
3  class CoCoInvariantBlockCorrectlyTyped(CoCo):
4    @classmethod                              encapsulate context
5    def checkCoCo(cls, neuron):               condition in a class
6      visitor = InvariantTypeCheckVisitor()
7      neuron.accept(visitor)
8                                              new visitor implementing the checks
9  class InvariantTypeCheckVisitor(ASTVisitor):
10   def visitASTInvariantBlock(self, node):
11     for expr in node.getExpressions():
12       if expr.getType() is not PredefinedTypes.getBooleanType():
13         Logger.logMessage(…)
14                                              report issue
```

Figure 5.5: Adding context conditions: Each context condition is implemented in a self-contained class with all required functionality to check the context.

a boolean expression has been stated. Figure 5.5 outlines how these modifications are implemented.

PyNestML delegates the task of checking models for semantical correctness to the orchestrating *CoCosManager* class. Storing references to all implemented context conditions, this class encapsulates all implemented semantical checks. It is therefore necessary to extend this class by a reference to the above-introduced *CoCoInvariantBlockCorrectlyTyped*. Whenever a processed model is checked, all context conditions are consecutively invoked on the AST and errors are reported. Figure 5.6 illustrates how the *CoCosManager* class has to be extended to regard a new context condition.

5.3 Modifying the code-generating Backend

With the introduction of new concepts to the model-processing frontend, it is also often intended to generate new artifacts or additional code. Extensions are hereby focused in the employed templates which depict how code has to be generated. In the case of the *invari-*

```python
1  # CoCosManager.py                                          Python
2  class CoCosManager(object):
3    ...
4    def postSymbolTableChecks(cls, neuron):                include new CoCo
5      ...                                                   in checks
6      CoCoInvariantBlockCorrectlyTyped.checkCoCo(neuron)
7      CoCoAllVariablesDefined.checkCoCo(neuron)
8      ...
```

Figure 5.6: Extending the *CoCosManager*: New context conditions have to be made known to the managing *CoCosManager* class.

```
                                                    NeuronClass.jinja2
...
/* Parameter and state extractions and manipulation functions*/
{{neuronName}}::Buffers_::Buffers_({{neuronName}} &n):  logger_(n)
{%- if useGSL -%} , __s( 0 ), __c( 0 ), __e( 0 )
{%- endif -%}{
 // Initialization of the remaining members is deferred to  init_buffers_().
}
{%- if neuron.hasInvariants() -%}
   {%- include "Invariant.jinja2" -%}
{%- endif -%}
...

                 include template for invariant generation

                                                    Invariant.jinja2
                          {{neuronName}}::Invariant({{neuronName}} &n){
                            {%- for inv in neuron.getInvariants() -%}
                            assert({{{printer.printExpression(inv.getExpression())}}});
                            {%- endfor %}
                          }
```

Figure 5.7: Inclusion of new templates: The existing set of templates is modified to include additional templates. For the sake of modularity, each extension should be implemented in an individual artifact.

ant block as introduced in the previous section, it is necessary to extend the existing set of templates to enable a generation of invariants in C++ code. The modularity of templates enables an easy to extend structure where additional concepts can be included by implementing sub-templates. New templates can be composed of existing ones. Modifications to the code-generating backend are hereby conducted in the following components:

- New templates which embody additional code that has to be generated.

- The governing templates in order to include the extensions.

As illustrated in Figure 5.7, the existing *NeuronClass* template is extended by a new *invariant* function which checks all stated invariants during the execution of the simulation. JinJa2 as the underlying generator engine of PyNestML features concepts for template inclusion and therefore enables an easy extension of PyNestML's code generator. The referenced template is hereby implemented as a new artifact. In conclusion, it is sufficient to implement all extensions in individual templates and include them by the above-demonstrated mechanism.

Chapter 6

The MontiCore Language Workbench

As shown in the previous chapters, the engineering of domain-specific languages is an error-prone process with many pitfalls and hard to find mistakes. While errors in the underlying grammars can be mostly detected and reported by the processing tools, many other components are completely written by hand and require a manual review process in order to detect unintended behavior or side effects. Here, the generation of software components has emerged as a possible solution to avoid these problems and accelerate the overall development process of a DSL. By verifying the component-generating framework by means of formal approaches [Ble05], it can be ensured that the generated results are correct. Several tools and complex workbenches have been developed in recent years supporting a wide range of tasks, cf. [SSV⁺b], from the generation of the underlying structure of the required components, e.g., classes where all methods have to be implemented by hand, through to more complex subsystems with fully functional routines such as visitors [HNRW16]. In this work, we will focus on the state-of-the-art language workbench *MontiCore* [KRV10, RH17] and demonstrate how a correct extension and integration of this framework can significantly reduce the overall implementation effort. The goal of this chapter is, therefore, to illustrate how MontiCore can be extended to support a new platform for the generation of several components as introduced in previous chapters, making a manual adaptation of the code base unnecessary whenever new extensions to PyNestML are implemented. For this purpose, we first briefly introduce MontiCore's workflow in section 6.1, before demonstrating in section 6.2 how it can be extended to support a new target platform.

6.1 The Workflow of MontiCore

The MontiCore Language Workbench is a framework developed with the intent to support and, in part, automate the development of DSLs. MontiCore provides a rich and expressive grammar declaration language for the definition of the abstract and concrete syntax in the same artifact. For this purpose, a syntax similar to Antlr was selected and enriched by several semantical as well as syntactical concepts, e.g., *interfaces* which enable the concept of *inclusion polymorphism* [Rey09] in grammar rules. With more than one decade in practice, many quality of life features were evaluated and integrated to accelerate a grammar definition and reduce the poor legibility as often present in defined grammars. Syntactical adaptations were focused on a clearer declaration of grammars, e.g., where
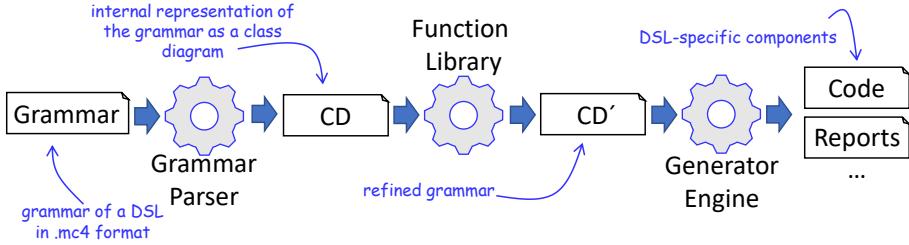
Figure 6.1: MontiCore's workflow: A given grammar in MontiCore format is first read in by a *grammar parser* and an internal representation by means of a class diagram is created. The class diagram is subsequently processed by the *function library* and enriched by additional classes, methods, and attributes. An extended class diagram is finally handed over to a target-specific code *generator engine*. The result is a generated set of DSL-specific components which can now be used in concrete projects to parse and interact with models.

tokens and grammar rules can be directly distinguished by their respective keywords. The core of MontiCore is a powerful grammar-processing engine, enabling the automated transformation and generation of a large set of components as often required in concrete DSLs: Besides AST classes[1] for all instantiable elements of the handed over grammar, MontiCore features a routine for an automated derivation and generation of AST visitors [HNRW16]. For the sake of semantic correctness of read-in models as processed by the handled DSL, MontiCore partially generates the structure of a symbol table and context conditions [HMSNR15]. The generation of a lexer and parser for the language as denoted by the grammar artifact completes the feature set, enabling the user of the workbench to derive a self-contained and easy to adapt collection of tools for the declared DSL. MontiCore, therefore, represents an ideal starting point for the engineering of new modeling languages, where many components are no longer written by hand but generated by the workbench.

Figure 6.1 illustrates the overall grammar-processing routine which consists of three major steps, each of which implemented as a modular component and geared together to generate a DSL-specific toolchain. The workflow is initiated by providing the workbench with one or more DSL-defining grammars, an artifact written in MontiCore's grammar specification language and identifiable by the file ending *mc4*. MontiCore grammars feature a syntax which is structured according to the Extended Backus-Naur Form (EBNF) and is similar to other existing grammar specification languages, cf. [PLW$^+$00]. Analogously to Antlr, cf. section 2.1, a grammar artifact is composed of lexer and parser productions, defining the abstract as well as concrete syntax of a DSL [Kra10]. Here, a modular and concise structure represents the key advantage: MontiCore features a concept for the composition of grammars by means of components [HLP$^+$15]. By reusing existing grammar elements, e.g., token definitions, only the core of a DSL as represented by the grammar rules has to be designed by hand. A definition of tokens is usually not required but can be imported from MontiCore's rich collection of predefined tokens. In conclusion, the grammar is constructed according to the *building block* approach, where a given DSL is designed by composing existing components. Figure 6.2 illustrates the working example grammar as introduced in Figure 3.3 adapted to MontiCore's grammar format. As we will show in section 6.2, the later on generated AST classes can be partially manipulated by providing additional details in the grammar, e.g., stating names of a grammar rule's component.

A read-in grammar is first checked for syntactical and semantical correctness. For this

---

[1] often referred to as the *Meta Model* of a DSL

72

```
 1  package org.pynest;                                               ┌─────┐
 2                                                                    │ MC  │
 3  grammar PyNestML extends de.monticore.literals.Literals {         └─────┘
 4                                                         import MontiCore
 5      //Tokens                                           token collection
 6      token COMMENT = ("#" (~('\n' |'\r' ))*) : { _channel = HIDDEN; };
 7
 8      //Grammar Rules
 9      Neuron = "neuron" StringLiteral (Block)* EOF;
10      Block = StringLiteral ':' (Assignment | Declaration)* "end";
11      Assignment = StringLiteral '=' Expression;
12      Declaration = StringLiteral Type ('=' Expression)?;
13      Type = "integer" | "boolean" | "string";      reuse MontiCore token definitions
14      Expression = simpleExpression                 from de.monticore.literals.Literals
15              | Expression (times:'*'|div:'/') Expression
16              | Expression (plus:'+'  |minus:'-') Expression;
17      SimpleExpression = StringLiteral | BooleanLiteral | NumericLiteral;
18  }
```

Figure 6.2: Grammar definition, cf. Figure 3.3, adapted to MontiCore syntax.

purpose, MontiCore parses the DSL-defining grammar and subsequently checks its semantical correctness by means of a symbol table and a set of context conditions, cf. section 2.1. MontiCore implements a wide range of semantical context rules a grammar has to fulfill to be regarded as being correct: From basic checks, e.g., requiring that the grammar's name has to be introduced by an upper-case letter, to more complex requirements, e.g., that all referenced grammar rules are defined. All context conditions as well as MontiCore's grammar parser are hereby completely target platform-agnostic and therefore decoupled from the artifacts-generating backend of the language workbench.

 If a handed over grammar is well defined and all referenced super-grammars are available, an internal representation is created for further processing. This representation is stored as a textual *class diagram* of the corresponding grammar. A class diagram consists of class definitions for each individual production stated in the source grammar, cf. Figure 6.3. For each part of a grammar rule, e.g., a marker such as the plus symbol, a respective attribute is added to the grammar rule's class. Subsequently, MontiCore further enriches the textual class diagram with additional methods. For each attribute, a data retrieval and manipulation method is added. Other methods as often required in a DSL toolchain, e.g., *equality* and *toString* operations, are constructed according to the grammar rule and added to the class diagram. The result is a class diagram containing all elements required for the further on introduced code generation of DSL components. Due to MontiCore's underlying platform, all references and types stated in the class diagram are denoted by concepts of Java.

 A modified class diagram is handed over to the final step, namely the generation of the corresponding platform-specific code. Here, MontiCore performs a *model to text* transformation [Sch12] and generates the class diagram in a target-specific format. Based on the selected target platform, a different backend subsystem is selected and further used for code generation. Each backend hereby inspects the given class diagram as well as the read-in grammar and generates a set of components, amongst others all required AST classes, a base visitor class, and a partially generated symbol table subsystem. All imports stated in the initial grammar are resolved and added to the internal representation of the grammar before it is used to generate the respective lexer and parser implementation. Figure 6.4

```
Expression = simpleExpression | Expression (times:'*'|div:'/') Expression | Expression (plus:'+'|minus:'-') Expression;    MC
```

each grammar rule
represented by a class

references to other grammar rule classes   CD

```
classdiagram PyNestML{
...
public class ASTExpression{
    protected Optional<de.monticore.PyNestML.ASTSimpleExpression> simpleExpression;
    protected java.util.List<de.monticore.PyNestML.ASTExpression> expression;
    protected Optional<String> times;
    protected Optional<String> div;
    protected Optional<String> plus;
    protected Optional<String> minus;
  }
...
}
```

terminals stored by their
token text

Figure 6.3: From grammar to class diagram: For each rule in the read-in grammar a new class definition is created. Terminals are stored by their text, non-terminals by references to sub-rule classes.

illustrates how a class definition in the class diagram is used to generate a respective AST class. The processing of a source grammar terminates after all DSL components have been generated.

MontiCore's workflow is hereby controlled by a user-supplied *groovy* script [CH06], where individual steps, e.g., the generation of a symbol table component, can be deactivated if not required. Consequently, the set of generated artifacts can be parametrized without changing the framework's implementation. In cases where handwritten code has to be injected, MontiCore features the concept of generated *top* classes [RH17]. Whenever handwritten code has been handed over, the framework checks whether artifacts with equivalent names are part of the generated components. In this case, an abstract top class, e.g., *ASTNeuronTop*, is generated instead, representing a basic set of features which can be extended by handwritten code.

6.2 Extension of MontiCore

MontiCore provides several target platforms for component generation, from imperative languages such as *Java*, through to the scripting language *JavaScript*. As introduced in chapter 1, Python was selected as the new target platform for NestML. Up to now, MontiCore did not support this platform for code generation. However, due to MontiCore's highly customizable behavior and modular structure, a possible extension to new target platforms can be achieved with proportionally tolerable effort. In this section we will, therefore, demonstrate how the existing infrastructure has to be adjusted to support Python as a new target platform for the generation of code and reports. As illustrated in Figure 6.5, we will demonstrate how an adaption can be performed based on a subset of available features, namely the lexer and parser as well as AST classes and visitors. In order to exemplify this process, the working example grammar as illustrated in Figure 6.2 will be used. The routine required to generate all remaining components of the model-processing frontend, e.g., a symbol table infrastructure, is hereby extended analogously.

Due to the target platform-agnostic structure of the input grammar format, no modifications to MontiCore's grammar definition language are required whenever a new target for code generation is added. Consequently, the parsing of the grammar does not have to be modified, but can be used in a black-box manner. Regardless of the selected target

```
classdiagram PyNestML{                                                              CD
  ...
  public class ASTExpression{
    protected Optional<de.monticore.PyNestML.ASTSimpleExpression> simpleExpression;
    protected java.util.List<de.monticore.PyNestML.ASTExpression> expression;
    protected Optional<String> times;
    protected Optional<String> div;
    protected Optional<String> plus;
    protected Optional<String> minus;
  }
  ...
}
```

for each class generate
a Java artifact

collection of methods
added by MontiCore

```
public class ASTExpression{                                                         Java
    protected Optional<de.monticore.PyNestML.ASTSimpleExpression> simpleExpression;
    protected java.util.List<de.monticore.PyNestML.ASTExpression> expression;
    protected Optional<String> times;
    protected Optional<String> div;
    protected Optional<String> plus;
    protected Optional<String> minus;

    public ASTExpression(...){...}
    public java.util.List<de.monticore.PyNestML.ASTSimpleExpression> getSimpleExpressions(...){...}
    public boolean deepEquals(...){...}
    ...
}
```

effective methods

Figure 6.4: From class diagram to code: The class diagram is enriched by methods and classes. Subsequently, each defined class is generated to a target-specific artifact, e.g., Java code.
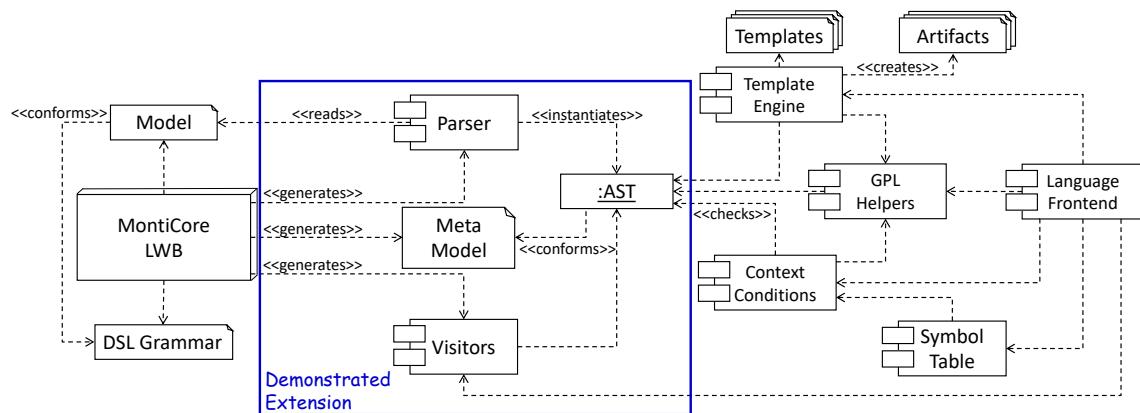

Figure 6.5: Demonstrated extensions to MontiCore: Representing a foundation of the model-processing frontend, the demonstrated extension enables MontiCore to generate all components required to parse and interact with textual models in Python.

platform, a given grammar always has to follow the same set of rules, e.g., that all referenced productions are defined. Thus, the context conditions each processed grammar has to fulfill can be reused without altering their behavior. All this results from MontiCore's strict decoupling of the frontend, transformation and backend subsystems. Changes to the code-generating backend do not affect the frontend. In conclusion no modifications to the grammar-parsing frontend are required to support a different platform for code generation.

As introduced in section 6.1, MontiCore stores a parsed grammar artifact by means of a textual class diagram. This approach to store a grammar features two major advantages: On the one hand, a grammar class diagram is mostly programming language-agnostic. All types and references are of basic nature, i.e., either references to other rules, or primitive types and collections of such. On the other hand, by utilizing such an internal representation of the grammar's structure, it is easily possible to adjust or extend it by new properties, an approach similar to the concept of modifiable ASTs as introduced in section 2.1. Those modifications are implemented by *decorators* which inspect a class diagram and extend ("decorate") them with additional methods, attributes and classes. Although almost all targets for code generation share a kernel of required functionality, e.g., data retrieval operations, often target platform-specific functions such as an overwritten *__str__* operation have to be added to the generated components. For this purpose each target for code generation has to feature a dedicated *Decorator* class which adds all required functionality to the grammar's class diagram. In our use case, it is necessary to create a new Python-specific class diagram decorator, which inspects a class diagram and extends it by additional methods. We therefore implemented the new *PythonCdDecorator* class which encapsulates all required routines on a given textual class diagram. However, given Python's overall paradigm of an interpreted language and the concept of duck typing [van95], it is not possible to provide all concepts as existing in Java in the generated Python components. Therefore, while all data access and modification methods were successfully ported, other specifications, e.g., the declaration of packages to which a module belongs to, could not be migrated, due to Python's different handling of modules. Moreover, while Java provides an approach to specify the access modifier, i.e., which elements are available in which scopes, making it an inherent part of the language itself, Python transfers this responsibility to the client and its disciplined usage of correct access paths.

The process as executed by the *PythonCdDecorator* class can be subsumed as follows: For each available class in the class diagram, inspect all declared attributes. Based on the type of the attribute, add new access and modification operations to the class diagram. Here, MontiCore utilizes the concept of *hook points* [RH17], i.e., specified locations in the textual class diagram where additional declarations can be injected. For this purpose, MontiCore first adds method headers[2] to the classes stored in the class diagram by means of the *addAdditionalMethods* operation. Each added method header introduces a new hook point for a method body. For each created hook point, the *addAdditionalMethods* operation subsequently indicates which template has to be used to generate the content of the respective method body. MontiCore, therefore, replaces method body placeholders as added together with a new method by a concrete instruction which template has to be used to generate the content. Figure 6.6 illustrates the routine executed to add a *getter* method to the processed class diagram. In order to support all required methods, we implemented an adjusted version of the *addAdditionalMethods* operation which adds a set of method headers together with concrete instructions for the corresponding method bodies. By

---

[2]e.g., *public Optional<de.monticore.PyNestML.ASTExpression> getASTExpression();*

```
1  //PythonCdDecorator.java
2  …                                                   1.define header of added method
3  String header = "public " + TypesPrinter.printType(attribute.getType()) + " get"  + attribute.getName() + "();" ;
4
5  HookPoint getMethodBody = new TemplateHookPoint("ast_python.additionalmethods.Get", clazz, attribute.getName());
6                                                      2. attach generating template to placeholder
7  replaceMethodBodyTemplate(clazz, header, getMethodBody);
8  …                                                   3. inject method definition into class diagram
```

Java

Figure 6.6: Method injection in class diagrams: The class diagram decorator first creates a new method header. Subsequently, a new hook point is created and a template as used to generate the method body is attached. Finally, the method header as well as the hook point are injected into the class diagram.

utilizing the new decorator, MontiCore extends a class diagram by data retrieval and modification operations, equality checks as well as operations to clone a given node. New classes, e.g., the *ASTNodeFactory* as used to create new AST nodes, are added to the class diagram by the same principle. Here, hook points are coupled to the class signature instead of a method header. Analogously to the functionality implemented in the existing Java artifact generator, the implemented Python-specific decorator extends a class diagram by a *Factory* class [VHJG95] as a central point for node construction, as well as a *Builder* class [VHJG95] for the creation of complex objects. The result is a class diagram where several methods and classes have been added by MontiCore.

In most cases, different target platforms utilize a different set of keywords to denote certain concepts. For instance, while Java indicates alternatives by the *else-if* directive, Python employs *elif* to instruct such behavior. These differences in possible targets have to be handled in order to avoid namespace collisions of generated code and concepts in the respective target platform. Depending on the selected target platform, it is therefore required to implement a new *NamesHelper* class which provides a routine to avoid namespace collisions. To avoid a collision with keywords reserved in Python, the *PythonNamesHelper* class was implemented. If a given class diagram contains an element whose name collides with a keyword in Python, the routine implemented in the *PythonNamesHelper* prefixes the name with $r\_$. For instance, in the case that a given rule in the grammar has been named *elif*, the corresponding node creating method of the generated *ASTNodeFactory* class references to r_elif. This procedure ensures that the generated components are compilable and semantically correct.

Having an extended class diagram, MontiCore utilizes it to generate all DSL components in a target-specific format. The templates used to determine this format are therefore the second component which has to be adjusted to support Python as a new target platform. Here, one of the key difference between Python and other target platforms has to be regarded, namely the notation employed to separate individual statements. While Java utilizes brackets to mark the beginning and end of a unit and disregards indentation and comments, Python focuses on indentation as a means to specify those properties. In order to adhere to this requirement, we designed all templates to preserve a correct indentation as well as an appropriate usage of keywords and tags, cf. Figure 6.7. For each class in the class diagram, the implemented *AstPythonGenerator* class therefore generates a respective Python artifact. As previously mentioned, the grammar's class diagram is enriched by new properties by means of decorators. Some of these specifications injected into all classes in the diagram, e.g., variables which are used to store comments, require additional dependencies to remain valid. Up to now, all runtime environment dependen-

```
1  # generated by template ast_python.AstClass                          FTL
2  class ${ast.getName()}(${genHelper.getSuperClassesAsString(ast)}):
3      <#if genHelper.isAbstract(ast)>
4      """   This is an abstract class.   """
5      __metaclass__ = ABCMeta
6      </#if>
7
8      <#-- generate all attributes -->
9      <#list ast.getCDAttributes() as attribute>
10     <#if !genHelper.isInherited(attribute)>
11     ${tc.includeArgs("ast_python.Attribute", [attribute, ast])}
12     </#if>
13     ...
```

use Python concepts of object-orientation

utilize Python-specific sub-templates

preserve correct indentation and syntax

```
1  # generated by template ast_python.AstClass                    Python
2  class ASTSimpleExpression(ASTNode):
3      stringLiteral = none
4      booleanLiteral = none
5      numericLiteral = none
6
7      def __init__(self, stringLiteral, booleanLiteral, numericLiteral):
8          super(ASTSimpleExpression, self).__init__()
9          self.stringLiteral = stringLiteral
10         self.booleanLiteral = booleanLiteral
11         self.numericLiteral = numericLiteral
12
13     def getStringLiteral(self):
14         ...
```

Figure 6.7: Template and result: A *freemarker* [fre17] template as used to generate an AST class. Comment tags as well as keywords of Python are integrated. Correct indentation in the template ensures syntactical correctness of the generated code. Adequate counter pieces for object-orientation in Python, e.g., the *meta-class* notation for abstract classes, are utilized.
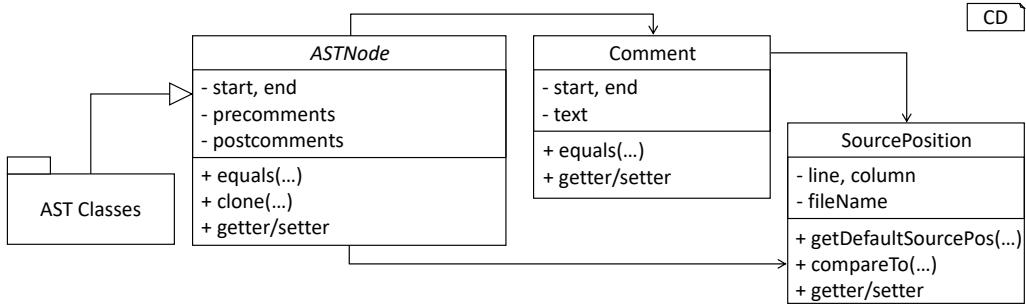
Figure 6.8:   Generated dependencies: In order to generate code with available dependencies, the featured extension generates an *ASTNode* class representing the base for all concrete AST classes, the *Comment* class used to store comments and a *SourcePosition* representing a source position in the textual model in a DSL.

cies have been resolved from within MontiCore, i.e., whenever one of the dependencies was required in the generated code, the corresponding module was simply imported from MontiCore. However, due to the circumstance that MontiCore is not available in Python, it is not possible to import those specifications into the generated project by downloading a specific library or framework. To solve this problem, parts of the runtime environment are generated together with the remaining DSL-specific components. Consequently, we implemented three additional templates to preserve consistency of generated Python code: The *Comment* class template is used to generate a component employed to store comments of a model, while the *SourcePosition* class template is responsible for the generation of a component which summarizes commonly required operations on source positions. The generated *ASTNode* class represents a base class of all AST classes and contains general operations and properties as common in all AST nodes, e.g., a *comment* field. Figure 6.8 illustrates the generated dependencies.

  With a collection of AST classes representing a data structure for a read-in model of the handled DSL, MontiCore proceeds to generate *visitors* as well as a *lexer* and *parser*. Depending on the selected target platform, a different syntax and approach has to be used to generate semantically correct components. The routine for the generation of these components is therefore modified to support Python as a new platform. By means of

the generated lexer and parser, we are able to read in a model and reconstruct it as a parse tree, cf. section 2.1. However, the parse tree misses common utilities, e.g., data retrieval and manipulation methods, thus a set of AST classes is generated and used. The process of retrieving details from the DSL model's parse tree and their storage in an AST requires a complex traversal routine on the parse tree. Luckily, the structure of this routine can be completely derived from the source grammar. In the case of Java as the target platform for code generation, MontiCore integrates the process of initializing an AST from a textual model into the generated parser. In conclusion, the parser can be used to parse a model to the corresponding parse tree or an AST. However, here a clear single responsibility is no longer given, since two different routines are combined in a single component. In cases where context conditions of a DSL have to be checked during the model-parsing process, it is not directly possible to utilize a modified AST-creating routine. The implemented extension to MontiCore therefore separates these concepts into two different components: Lexer and parser are generated without any additional code for the creation of an AST, while the further on introduced *ASTBuilderVisitor* class is used to inspect a given parse tree and instantiate a model's AST. A clear separation of the model-parsing and AST-building processes results in a modular and easy to maintain structure. Moreover, chapter 3 demonstrated that it can be beneficial to have direct access to the parse tree, e.g., when dealing with source model comments.

MontiCore utilizes *Antlr* [PLW+00] by creating an intermediate representation of the source grammar in Antlr specific syntax and subsequently generating the lexer and parser by using Antlr's toolchain. Moreover, in the case of Java as the target for component generation, MontiCore injects AST-creating code into the generated parser. This behavior has been modified and consequently no additional code is added to the generated lexer and parser in Python. Instead, the responsibility to initialize an AST is delegated to the *ASTBuilderVisitor*, which can now be used to create an AST from a given parse tree. This approach to separate concepts enables an implementation of arbitrary operations on the parse tree, while the abstract syntax tree can still be created by the specialized visitor. Moreover, all components are easier to maintain and extend given the strict separation of concerns, where the set of visitors can be easily adjusted without any need to modify the generated lexer and parser. All modifications to MontiCore are hereby limited to the *Grammar2Antlr* module, where, depending on the selected target for component generation, no code is injected into the intermediate representation of the DSL grammar in Antlr syntax. Figure 6.9 illustrates how depending on the selected target platform additional code is injected into the intermediate grammar, and thus the generated lexer and parser. Consequently, all methods in the *Grammar2Antlr* class were modified to check which target for code generation has been selected, and inject AST-building code only if Java has been selected as the target platform.

The generated *ASTBuilderVisitor* class is used to inspect a given parse tree and instantiate a model-specific AST. Having a class diagram of all rules in the initial grammar, MontiCore detects which elements are stored in a parse tree node and how these elements have to be retrieved, cf. Figure 6.10. This information is then used to generate a visitor which visits all available fields of a parse tree node, retrieves stored data and attaches a new node to the current AST. The generated *ASTBuilderVisitor* class executes this process for the whole model before a new AST representation of the processed model is returned.

With a set of AST classes, a lexer and parser as well as the *ASTBuilderVisitor* class, MontiCore is able to generate all DSL components required to parse a model and create the respective AST. However, often it is of interest to parse not a complete model, but only cer-
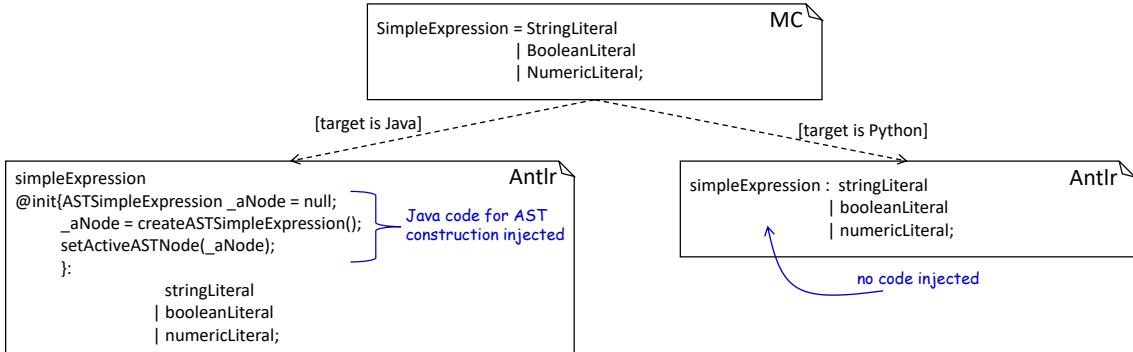
Figure 6.9: From MontiCore to Antlr: Depending on the selected target for the component generation, the source grammar is enriched by additional methods for the construction of an AST, before it is generated to Antlr-specific grammar format.
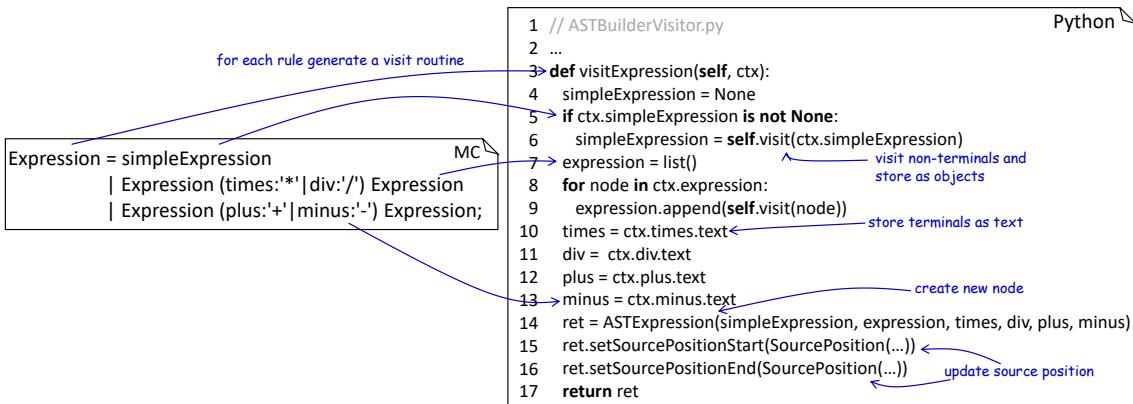
Figure 6.10: From class diagram to *ASTBuilderVisitor*: For each grammar rule, an individual parse tree *visit* method is generated. Terminals are retrieved from the processed node and their text stored, while non-terminals are first visited and the returned objects used. Finally, a new AST node is constructed and returned.

```
1 <#list classes as class>
2 @classmethod
3 def parse${class.getName()}(cls, element):
4   try:
5     input = FileStream(element)
6   except IOError:
7     input = InputStream(element)
8   lexer = AntlrLexer(input)
10  stream = CommonTokenStream(lexer)
11  parser = AntlrParser(stream)
12  astBuilder = AstBuilderVisitor()
13  return astBuilder.visit(parser.${class.getName()}())
14 </#list>
```

FTL   SC

[input is file]   [input is string]

Create File Stream   Create Input stream

Create Lexer

Create Token Stream   [incorrect token]

Parse Token Stream   [no grammar rule]

Construct AST

Figure 6.11: The *Parser* class: For each rule in the grammar, a new *parse* method is generated. This method encapsulates all steps required to parse a model and construct the corresponding AST.

tain parts of it, e.g., a single statement. The DSL-specific process of utilizing a new lexer, parser and AST builder to parse such a statement is hereby always constructed similarly, by first creating a lexer and parser to parse an element, and subsequently instantiating an AST by the *ASTBuilderVisitor*. In order to simplify this model- and statement-parsing process in the handled DSL, an additional template for code generation was designed: Executed during the generation of AST classes, this template generates a new *Parser* class, cf. Figure 6.11. For each available production in the initial grammar, this class contains a method encapsulating all above-mentioned steps. The generated DSL toolchain can then interact with the model-parsing frontend by utilizing methods contained in the interface of the generated *Parser* class.

As demonstrated in section 2.1, the *visitor pattern* enables us to traverse and work on a given AST. MontiCore features all required components to generate a base visitor implementation for a read-in model and its respective AST classes, where only the concrete operations on the AST have to be specified by handwritten code. We, therefore, conclude the set of implemented extensions to MontiCore by the *PythonVisitorGenerator* class whose routines inspect a grammar class diagram and retrieve all classes which are neither static nor abstract, thus can be used to create concrete objects during the model processing in the generated DSL toolchain. These classes are then utilized to generate a base visitor class which implements a traversal strategy with a placeholder for concrete operations on the AST. Utilizing the *template method* pattern [VHJG95], arbitrary operations on the AST can then be implemented. The generation of such a visitor prevents the error-prone writing of a routine whose size grows proportionally to the number of AST classes and attributes. Due to Python's missing concept of declared types and function overloading, it is not possible to use *ad-hoc polymorphism* [Car85] in the generated code to dispatch a *visit* routine invocation according to the handed over type. Consequently, the generated visitor class features only a single *visit* method. This routine checks the type of the handed over AST node and dispatches the call to the correct operation. The respective operation is hereby named uniquely based on the visited node, e.g., *visitASTDeclaration*, in order to avoid clashes with other methods. We therefore avoid the unavailable function overloading in Python by utilizing a *monomorph* approach. Figure 6.12 briefly compares the generated base visitors as used in Java and Python. Together with the model-parsing process encapsulated in the *Parser* class, the generated base visitor enables a DSL-specific toolchain to parse and interact with a textual model.

```
1  // SimpleVisitor.ftl – Java                        FTL
2  …
3  <#list cd.getTypes() as type>
4      <#if type.isClass() && !type.isAbstract()>
5  default public void visit(${type.getName()} node){}
6      </#if>
7  </#list>
9  …
```

dynamic binding based on type

polymorph function by overloading

```
1  // SimpleVisitor.ftl - Python                      FTL
2  …
3  def visit(self, node):
4  <#list cd.getTypes() as type>
5      <#if type.isClass() && !type.isAbstract()>
6      if isinstance(node, ${type}):
7          self.visit${type}(node)
9      </#if></#list>
10 …
11 <#list cd.getTypes() as type>
12 def visit${type.getName()}(self, node):
13     pass
14 </#list>
```

dispatcher function

dispatch based on type

delegate to monomorphic hook functions

Figure 6.12: Generated Java and Python visitor: Python as the new target for code generation does not provide the concept of method overloading. In order to solve this problem, an intermediate *visit* dispatcher method is generated which delegates according to the type of the handed over instance. Java supports ad-hoc polymorphism, thus no dispatcher method is required.

In conclusion, we see how MontiCore implements a complex processing of a DSL's grammar, with the result being several generated components as required in a DSL toolchain. Moreover, we also see that only a small set of MontiCore's processes has to be adjusted to support a new target platform for code generation. Besides the presented elements, MontiCore also features a concept for the generation of several additional components, among others a symbol table and basic context conditions. Here, one of the main problems preventing an easy to achieve extension to a new target platform for code generation is the dependency of the generated code to libraries and components which are only available in Java. In order to support the generation of such components, it would be necessary to migrate a vast collection of classes and libraries to Python (or any other target platform). Future work may, therefore, focus on an extension of MontiCore which generates self-contained code without any external dependencies, or an evaluation of existing technologies in respective languages which can be used as counter pieces to the existing Java infrastructure.

We conclude this section by a brief discussion on code generation in the concrete use case of PyNestML. As demonstrated, we were able to generate several DSL-specific components as required in the model-processing frontend: A lexer and parser used to parse the model to a parse tree, a collection of AST classes, and finally a basic as well as an AST-building visitor. All these components share a commonality, namely the fact that they have to be rewritten whenever changes to the underlying DSL grammar are implemented. Especially in the case of visitors which implement a routine which is simple by itself, but also consists of a massive codebase, code generation can be beneficial. Analogously, AST classes are always constructed in the same manner, with their basic schema being completely stated in the source grammar. In both cases a generation of components can be beneficial whenever the DSL is often modified. However, in the case of the symbol table and symbols, an automated code generation is only partially applicable. Although MontiCore is able to generate the basic symbol classes, it is still not directly comprehensible which components in the grammar correspond to symbols, and which to scopes. Here, it would be necessary to extend MontiCore with an annotation concept which indicates that certain symbol classes have to be created. Handwritten solutions are therefore the better options since the underlying concept of a symbol table and concrete symbols of a handled DSL do not change

often. The same reasoning applies to context conditions [Sch12]. Here, code generation is almost impossible. Context conditions often embody domain-specific knowledge whose description is mostly provided in a textual form. Thus, an automated generation is limited. A possible approach here could be the usage of formal constraint specification languages such as the *Object Constraint Language* (OCL, [Gog09]). By annotating conditions in the grammar, e.g., stating that at most one block of a certain type can be declared in a target model, it would be possible to automatically generate a basic collection of context conditions. However, specific and complex context conditions still have to be written by hand, making manual writing of code inevitable. Nonetheless, the above-introduced extension to MontiCore enables us to generate well-defined components of the model-processing frontend, and therefore to reduce the overall required effort during the reengineering as well as extension process.

Chapter 7

Tutorial

The installation as well as the execution of PyNestML requires a Python installation in version of at least 2.6, or 3 in an arbitrary subversion. All required dependencies of PyNestML are collected in *requirements.txt* and can be installed via Python's package management system *pip* [pyt17a] with the following command on Linux as well as Windows operating systems:

```
1 $ pip install -r requirements.txt
```

After the installation of all dependencies has been finished, it is necessary to execute the setup:

```
1 $ python setup.py install --user
```

If no errors were detected, the processing of models can be started by:

```
1 $ python PyNestML.py [ARGUMENTS]
```

where the arguments are:

Argument	Description
-h	Show a message with all available arguments.
-path [PATH]	Set the path to a single model or a directory containing models.
-target [PATH]	(Optional) Set the output directory. Default: {current-dir}/target.
-dry	(Optional) Execute a dry run where models are only analyzed but not generated to target format. Default: Off.
-logging_level [LEVEL]	(Optional) Indicate the severity of messages which shall be printed to the log. Available: {INFO,WARNING,ERROR,NO}, being more restrictive from left to right. Default: ERROR.
-module_name [NAME]	(Optional) The name of the overall module as used to install generated code. Default: Name of directory containing the models.
-store_log	(Optional) Indicate that a log file in JSON format containing all printed messages should be stored in the target path. Default: Off.
-dev	(Optional) Execute the toolchain in developer mode where artifacts are generated even if correctness can not be guaranteed. Default: Off.

A default execution of the toolchain can, therefore, be initiated by:

```
1 $ python PyNestML.py –path PATH\TO\MODELS
```

Figure 7.1 demonstrates the command line output of a successful and unsuccessful execution of PyNestML.

```
[1,GLOBAL, INFO, START_PROCESSING_FILE]:
Start processing '/home/nest/pynestml/models/izhikevich.nestml'
…
[12,izhikevich_neuron, INFO, CODE_SUCCESSFULLY_GENERATED]:          ✓
Successfully generated NEST code for the neuron: 'izhikevich_neuron' in: '/home/nest/pynestml/target/models'

[1,GLOBAL, INFO, START_PROCESSING_FILE]:
Start processing '/home/nest/pynestml/models/izhikevich.nestml'
…
[3,izhikevich_neuron, ERROR, CAST_NOT_POSSIBLE, [45:5;45:25]]:
Type of lhs 'hasSpiked' does not correspond to rhs '0'! LHS: 'boolean', RHS: 'integer'.   ✗
…
[7,izhikevich_neuron, INFO, NEURON_CONTAINS_ERRORS]:
Neuron 'izhikevich_neuron' contains errors. No code generated!
```

Figure 7.1: A successful and unsuccessful execution of PyNestML.

If no critical errors were detected, a new directory *target* is created containing all generated artifacts, cf. section 4.1. Among others, a *cmake* file [gnu17] is attached which enables the integration of the generated artifacts into the NEST ecosystem. The integration can hereby be invoked by:

```
1 $ cmake –Dwith-nest=<nest_install_dir>/bin/nest-config .
2 $ make
3 $ make install
```

For a detailed overview of commands and interaction possibilities with NEST, we refer to the official documentation [GMP13, NES17].

Chapter 8

Conclusion and Future Work

We conclude this thesis by a brief overview of the achieved results and an outlook to future work. Chapter 1 gave a short introduction to the scope of this thesis and which problems it tackles, namely the reimplementation of an existing neuroscientific framework for the modeling of spiking point neurons. For a basic understanding of the matter, section 2.1 first introduced the concept of domain-specific modeling languages and outlined all required components. Section 2.2 summarized the results of a short survey conducted to find the most suitable components for reuse in PyNestML. Here, *Antlr* was selected as the lexer and parser generator, while the task of handling physical units was delegated to the *AstroPy* unit system. This chapter also briefly demonstrated three generator engines and underlined their similarities.

Chapter 3 subsequently introduced the model-processing frontend of PyNestML and illustrated all conducted reengineering steps. To this end, section 3.1 showed how a grammar artifact represents the starting point of PyNestML and denotes all concepts of the implemented language. By using Antlr, the respective lexer and parser components were generated. Due to the insufficient nature of the instantiated parse tree, a set of AST classes was introduced. Coupling a data structure with common utility, these classes are used by the *ASTBuilderVisitor* to initialize an AST representation of a textual model. The task of collecting all source comments is delegated to the *CommentCollectorVisitor* class. Section 3.2 introduced a data structure for storage of context-related details, namely the *Symbol* classes. Together with the *predefined* subsystem, these components enable PyNestML to store a set of predefined types, variables and functions. Model types are hereby derived by the *ASTExpressionsTypeVisitor* and the *ASTUnitTypeVisitor* classes. Utilizing these components, section 3.3 introduced a subsystem for semantical checks. The *SymbolTable* class is hereby used to store the context of a processed model, while the *ASTSymbolTable-Visitor* is responsible for collecting all required details. A set of *context conditions* finally ensures that semantically incorrect models are filtered out. This chapter was concluded in section 3.4 with an overview of assisting classes. The overall interface to PyNestML is encapsulated in the *PyNestMLFront* class, while the *Logger* class implements an easy to use logging concept. This section also introduced the *higher-order visitor*, an extension to the visitor concept which avoids unnecessary sub-classing in many cases.

Based on the results from the previous chapter, chapter 4 demonstrated the NEST code generator. The coordinating *NestCodeGenerator* class takes care of all processes necessary to generate C++ code and delegates individual steps to assisting subsystems. Here, the *SymPySolver* class is used to interact with the ODE-toolbox by Blundell et al. The computed AST-to-AST transformations are subsequently integrated into the AST by the *EquationsBlockProcessor*, before a set of templates is used to generate the respective C++

code. This chapter also showed how different syntaxes can be integrated into a single target artifact by means of the *ExpressionPrettyPrinter* class. Especially the composable nature of the *pretty printer* and the *reference converters* was demonstrated, making the implementation of new targets for code generation an easy to achieve task. This chapter also gave a reasoning why a coupling of code generation and AST-to-AST transformations makes sense in environments where several target platforms are – or shall be – supported.

In order to demonstrate how the provided implementation of PyNestML can be extended, chapter 5 introduced three typical use cases. Section 5.1 illustrated how the model-processing frontend has to be adapted to support new productions in the grammar, while section 5.2 outlined all required steps to implement new semantical checks. Section 5.3 showed how additional templates can be integrated into code generation. However, an error-prone implementation of these extensions by hand can usually be avoided by utilizing appropriate tools. For this purpose, chapter 6 introduced the state-of-the-art language workbench MontiCore. This chapter showed which components of MontiCore have to be extended to support Python as a new platform for code generation. Chapter 7 concluded the thesis with a short tutorial on how to use PyNestML.

With Python as the new platform, an integration of PyNestML into neuroscientific ecosystems, e.g., other tools and frameworks, can be easily conducted. Moreover, bridge technologies were made obsolete, making the setup and usage of PyNestML easy to achieve. Future work may, therefore, focus on an extension of the existing framework by new functionality. The underlying DSL can be enriched by new concepts and approaches for specifying entities as often required in neuroscientific simulations, e.g., synapses or topologies of neurons. By adding support for new target platforms such as *SpiNNaker* [FGTP14], *Neuron* [CH06] or *LEMS* [CGC+14], a wider user base can be included, while the exchange and validation of neuron models is no longer hindered by a manual transformation process.

With the continuing rise of computational power, new and more complex simulations become possible. More sophisticated modeling approaches will, therefore, be required to provide appropriate scalability while being able to capture all details. Here, (Py)NestML provides a good foundation for future work. Concluding with a quote by Harriett Jackson Brown Jr.:

The best preparation for tomorrow is doing your best today.

Bibliography

[AEM+16]   Katrin Amunts, Christoph Ebell, Jeff Muller, Martin Telefont, Alois Knoll, and Thomas Lippert. The Human Brain Project: Creating a European research infrastructure to decode the human brain. *Neuron*, 92(3):574–581, 2016.

[AGH00]   Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[And03]   Alex M Andrew. Spiking neuron models: Single neurons, populations, plasticity. *Kybernetes*, 32(7/8), 2003.

[ART+13]   Astropy Collaboration, T. P. Robitaille, E. J. Tollerud, P. Greenfield, M. Droettboom, E. Bray, T. Aldcroft, M. Davis, A. Ginsburg, A. M. Price-Whelan, W. E. Kerzendorf, A. Conley, N. Crighton, K. Barbary, D. Muna, H. Ferguson, F. Grollier, M. M. Parikh, P. H. Nair, H. M. Unther, C. Deil, J. Woillez, S. Conseil, R. Kramer, J. E. H. Turner, L. Singer, R. Fox, B. A. Weaver, V. Zabalza, Z. I. Edwards, K. Azalee Bostroem, D. J. Burke, A. R. Casey, S. M. Crawford, N. Dencheva, J. Ely, T. Jenness, K. Labrie, P. L. Lim, F. Pierfederici, A. Pontzen, A. Ptak, B. Refsdal, M. Servillat, and O. Streicher. Astropy: A community Python package for astronomy. 558:A33, October 2013.

[Bec03]   Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[BHS07]   Frank Buschmann, Kelvin Henney, and Douglas Schimdt. *Pattern-oriented Software Architecture: on patterns and pattern language*, volume 5. John wiley & sons, 2007.

[Ble05]   Blech, Jan Olaf and Glesner, Sabine and Leitner, Johannes. Formal verification of java code generation from UML models. *Fujaba Days*, 2005:49–56, 2005.

[BPEM18]   Inga Blundell, Dimitri Plotnikov, Jochen Martin Eppler, and Abigail Morrison. Automatically selecting a suitable integration scheme for systems of differential equations in neuron models. *Frontiers in Neuroscience*, 2018.

[BW84]   Alan Bundy and Lincoln Wallen. *Context-Free Grammar*, pages 22–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984.

[Car85]   Cardelli, Luca and Wegner, Peter. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[Car07]   Ted Carnevale. NEURON simulation environment. *Scholarpedia*, 2(6):1378, 2007.

[CE97]   Anthony Clark and Andy Evans. Foundations of the Unified Modeling Language. In *Proceedigs of the 2nd Northern Formal Methods Workshop*. Springer, 1997.

[CEC00]   Krzysztof Czarnecki, Ulrich W Eisenecker, and Krysztof Czarnecki. *Generative programming: methods, tools, and applications*, volume 16. Addison Wesley Reading, 2000.

[CGC+14]   Robert C. Cannon, Padraig Gleeson, Sharon Crook, Gautham Ganapathy, Boris Marin, Eugenio Piasini, and R. Angus Silver. Lems: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning neuroml 2. *Frontiers in Neuroinformatics*, 8:79, 2014.

[CH06]   Nicholas T Carnevale and Michael L Hines. *The NEURON book*. Cambridge University Press, 2006.

[che17]   Cheetah Template Engine, Documentation. `http://pythonhosted.org/Cheetah/users_guide/`, 2017.

[DA01]   Peter Dayan and Laurence F Abbott. *Theoretical neuroscience*, volume 806. Cambridge, MA: MIT Press, 2001.

[DBE+08]   Andrew P. Davison, Daniel Brüderle, Jochen M. Eppler, Jens Kremkow, Eilif Müller, Dejan Pecevski, Laurent U. Perrinet, and Pierre Yger. PyNN: A Common Interface for Neuronal Network Simulators. *Frontiers in Neuroinformatics*, 2:3637 – 3642, 2008.

[doc17]   Docker Homepage and Documentation. `https://www.docker.com/`, 2017.

[Dun11]   Jeff Duntemann. *Assembly language step-by-step: Programming with Linux*. John Wiley & Sons, 2011.

[EHM+09]   Jochen Eppler, Moritz Helias, Eilif Muller, Markus Diesmann, and Marc-Oliver Gewaltig. PyNEST: a convenient interface to the NEST simulator. *Frontiers in Neuroinformatics*, 2:12, 2009.

[FF06]   Martin Fowler and Matthew Foemmel. Continuous integration. `http://www.thoughtworks.com/ContinuousIntegration.pdf`, 122, 2006.

[FGTP14]   S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The SpiNNaker Project. *Proceedings of the IEEE*, 102(5):652–665, May 2014.

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

[fre17]   Freemarker Homepage and Documentation. `https://freemarker.apache.org/`, 2017.

[Gam95]   Erich Gamma. *Design patterns: Elements of reusable object-oriented software*. Pearson Education India, 1995.

[GBR04]   Benjamin Geer, Mike Bayer, and Jonathan Revusky. The FreeMarker template engine, 2004.

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

[GMP13]   Marc-Oliver Gewaltig, Abigail Morrison, and Hans Ekkehard Plesser. NEST by Example: An Introduction to the Neural Simulation Tool NEST Version 2.6. 0. 2013.

[gnu17]   Gnu Make Documentation. `https://www.gnu.org/software/make/manual/make.pdf`, 2017.

[Gog09]   Martin Gogolla. *Object Constraint Language*. Springer US, 2009.

[Gou09]   Brian Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[Hel96]   Balzert Helmut. Lehrbuch der Software-Technik, 1996.

[Hem93]   Thomas Hemmann. Reuse approaches in software engineering and knowledge engineering: a comparison. In *Position Paper Collection of the 2nd Int. Workshop on Software Reusability*, number 93-69, 1993.

[HH09]   Suzana Herculano-Houzel. The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in Human Neuroscience*, 3:31, 2009.

[HLP+15]   Arne Haber, Markus Look, Antonio Navarro Perez, Pedram Mir Seyed Nazari, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of heterogeneous modeling languages via extensible and composable language components. In *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*, pages 19–31. IEEE, 2015.

[HMSNR15]   Katrin Hölldobler, Pedram Mir Seyed Nazari, and Bernhard Rumpe. Adaptable symbol table management by meta modeling and generation of symbol table infrastructures. In *Proceedings of the Workshop on Domain-Specific Modeling*, pages 23–30. ACM, 2015.

[HNRW16]   Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional language engineering using generated, extensible, static type-safe visitors. In *European Conference on Modelling Foundations and Applications*, pages 67–82. Springer, 2016.

[Izh03]   Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

[JBW+10]   Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. *The Definitive Guide to Jython: Python for the Java Platform*. Apress, Berkely, CA, USA, 1st edition, 2010.

[JČMG12]   David Joyner, Ondřej Čertík, Aaron Meurer, and Brian E Granger. Open source computer algebra systems: SymPy. *ACM Communications in Computer Algebra*, 45(3/4):225–234, 2012.

[jin17]   Jinja Template Engine, Documentation. `http://jinja.pocoo.org/docs/2.10/`, 2017.

[Kra10]   Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering.* Shaker, 2010.

[KRV10]   Holger Krahn, Bernhard Rumpe, and Steven Voelkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *Int. J. Softw. Tools Technol. Transf.*, 12(5):353–372, September 2010.

[KSJ+00]   Eric R Kandel, James H Schwartz, Thomas M Jessell, Steven A Siegelbaum, A James Hudspeth, et al. *Principles of neural science*, volume 4. McGraw-hill New York, 2000.

[LMB92]   John R Levine, Tony Mason, and Doug Brown. *Lex & yacc.* O'Reilly Media, Inc., 1992.

[Lou]   Kenneth C Louden. Compiler construction: Principles and practice. 1997. *PWS. Boston.*

[M+03]   Jürgen Karl Müller et al. *The Building Block Method: Component-Based Architectural Design for Large Software-Intensive Product Families.* Universiteit van Amsterdam, Faculteit der Natuurwetenschappen, Wiskunde en Informatica, 2003.

[MAT17]   MATLAB. *version 9.3 (R2017b).* The MathWorks Inc., Natick, Massachusetts, 2017.

[MBD+15]   Eilif Muller, James A Bednar, Markus Diesmann, Marc-Oliver Gewaltig, Michael Hines, and Andrew P Davison. Python in neuroscience. *Frontiers in neuroinformatics*, 9, 2015.

[McG07]   Paul McGuire. *Getting started with pyparsing.* O'Reilly Media, Inc., 2007.

[Mey02]   Bertrand Meyer. *Design by contract.* Prentice Hall, 2002.

[MHDH13]   Sayed Mehdi Hejazi Dehaghani and Nafiseh Hajrahimi. Which factors affect software projects maintenance cost more? 21:63–6, 03 2013.

[moz17]   Mozilla Corporation Website. `https://www.mozilla.org/de/`, 2017.

[MR]   Daniel D. McCracken and Edwin D. Reilly. Backus-Naur Form (BNF). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK.

[NES17]   Nest Simulator Website. `http://www.nest-simulator.org/`, 2017.

[Nol02]   John Nolte. The human brain: an introduction to its functional anatomy. 2002.

[NPRI09]   Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of JSON and XML data interchange formats: a case study. *Caine*, 2009:157–162, 2009.

[Par09]   Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* Pragmatic Bookshelf, 1st edition, 2009.

[PLW⁺00]   Terence Parr, John Lilly, Peter Wells, Rick Klaren, M Illouz, J Mitchell, Scott Stanchfield, J Coker, M Zukowski, and C Flack. ANTLR reference manual. *MageLang Institute, document version*, 2(0), 2000.

[PRB⁺16]   Dimitri Plotnikov, Bernhard Rumpe, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, and Abigail Morrison. NESTML: a modeling language for spiking neurons. *CoRR*, abs/1606.02882, 2016.

[pyt17a]   Python pip documentation. `https://pip.pypa.io/en/stable/`, 2017.

[pyt17b]   Python Ply documentation. `http://www.dabeaz.com/ply/`, 2017.

[RBF⁺05]   Daniel A Reed, Ruzena Bajcsy, Manuel A Fernandez, Jose-Marie Griffiths, Randall D Mott, Jack Dongarra, Chris R Johnson, Alan S Inouye, William Miner, Martha K Matzke, et al. Computational science: Ensuring america's competitiveness. Technical report, PRESIDENT'S INFORMATION TECHNOLOGY ADVISORY COMMITTEE ARLINGTON VA, 2005.

[Rey09]   John C Reynolds. *Theories of programming languages.* Cambridge University Press, 2009.

[RH17]   Bernhard Rumpe and Katrin Hoelldobler. MontiCore 5 Language Workbench Edition 2017. `http://www.se-rwth.de/`, 2017.

[Rie96]   Arthur J Riel. *Object-oriented design heuristics.* Addison-Wesley Publishing Company, 1996.

[Ron08]   Armin Ronacher. Jinja2 Documentation, 2008.

[Rum11]   Bernhard Rumpe. *Modellierung mit UML*, volume 2nd Edition. Springer, 2011.

[Rum17]   Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring.* Springer, 2017.

[Sch98]   Herbert Schildt. *C++: the complete reference.* Osborne/McGraw-Hill, 1998.

[Sch12]   Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*, volume 11. RWTH Aachen University, Germany, 2012.

[Sch17]   Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P.* PhD thesis, RWTH Aachen University, 2017.

[Som15]   Ian Sommerville. *Software Engineering.* Pearson, 10th edition, 2015.

[sou17]   Sourceforge Website. `https://sourceforge.net/`, 2017.

[SSV⁺a]   Klemens Schindler Schindler, Riccardo Solmi12, Vlad Vergu10, Eelco Visser10, Kevin van der Vlist13, Guido Wachsmuth10, and Jimi van der Woning13. The state of the art in language workbenches.

[SSV+b]   Klemens Schindlerf Schindlerf, Riccardo Solmim, Vlad Vergui, Eelco Visseri, Kevin van der Vlistk, Guido Wachsmuthi, and Jimi van der Woningl. Evaluating and Comparing Language Workbenches.

[ten17]   Tenjin Template Engine, Documentation. `http://www.kuwata-lab.com/tenjin/pytenjin-users-guide.html`, 2017.

[van95]   Guido van Rossum. Python tutorial, April 1995.

[VDKV00]   Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[VHJG95]   John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[Wes02]   J. Christopher Westland. The cost of errors in software development: evidence from industry. *Journal of Systems and Software*, 62(1):1–9, 2002.

Appendix A

PyNestML Grammar

```
1  grammar PyNESTML{
2
3     token SL_COMMENT =
4   ('#' (~('\n' |'\r' ))*) : { self._channel=2;};
5
6     token ML_COMMENT =
7   ("/*" .*? "*/" | "\"\"\"" .*? "\"\"\"") : {self._channel=2;};
8
9     token NEWLINE =
10  ('\r' '\n' | '\r' | '\n' ): {self._channel=3; };
11
12    token WS =
13  (' ' | '\t') :{ self._channel=1; };
14
15    token LINE_ESCAPE =
16  '\\' '\r'? '\n':{ self._channel=1; };
17
18    token BLOCK_OPEN =  ':';
19
20    token BLOCK_CLOSE = "end";
21
22    token BOOLEAN_LITERAL =
23  "true" | "True" | "false" | "False";
24
25    token STRING_LITERAL =
26  '"' ( 'a'..'z'| 'A'..'Z' | '_' | '$' )
27  ( 'a'..'z'| 'A'..'Z' | '_' | '0'..'9' | '$' )* '"';
28
29    token NAME =
30  ( 'a'..'z'|'A'..'Z' | '_' | '$' )
31  ( 'a'..'z'|'A'..'Z' | '_' | '0'..'9' | '$' )*;
32
33    token INTEGER = NON_ZERO_INTEGER | '0';
34
35    fragment token NON_ZERO_INTEGER = '1'..'9' ('0'..'9')*;
36
37    token FLOAT = POINT_FLOAT | EXPONENT_FLOAT;
38
39    fragment token POINT_FLOAT = (NON_ZERO_INTEGER |'0')? FRACTION
40                             | (NON_ZERO_INTEGER |'0')  '.';
```

```
41
42    fragment token EXPONENT_FLOAT =
43  ( NON_ZERO_INTEGER | POINT_FLOAT ) EXPONENT ;
44
45    fragment token EXPONENT =
46  ( 'e'|'E' ) ( '+'|'−' )? ( NON_ZERO_INTEGER |'0' );
47
48    fragment token FRACTION = '.' ('0'..'9')+;
49
50    /*******************************************************
51    * NestML−Language
52    *******************************************************/
53
54    NestmlCompilationUnit = (Neuron | NEWLINE )* EOF;
55
56    Neuron = "neuron" NAME Body;
57
58    Body = BLOCK_OPEN
59            (NEWLINE | BlockWithVariables | UpdateBlock |
60             EquationsBlock | InputBlock | OutputBlock | Function)*
61          BLOCK_CLOSE;
62
63    BlockWithVariables =
64        (isState:"state"|isParameters:"parameters"|
65         isInternals:"internals"|isInits:"initial_values")?
66        BLOCK_OPEN
67          (Declaration | NEWLINE)*
68        BLOCK_CLOSE;
69
70    UpdateBlock = "update" BLOCK_OPEN
71                     Block
72                     BLOCK_CLOSE;
73
74    EquationsBlock = "equations" BLOCK_OPEN
75                      (OdeFunction|OdeEquation|OdeShape|NEWLINE)+
76                       BLOCK_CLOSE;
77
78    InputBlock = "input" BLOCK_OPEN
79                    (InputLine | NEWLINE)*
80                     BLOCK_CLOSE;
81
82    InputLine =
83        name:NAME
84        ("[" sizeParameter:NAME "]")?
85        (Datatype)?
86        "<−" InputType*
87        (isCurrent:"current" | isSpike:"spike");
88
89    InputType = (isInhibitory:"inhibitory" |
90                 isExcitatory:"excitatory");
91
92    OutputBlock = "output" BLOCK_OPEN
93                    (isSpike:"spike" | isCurrent:"current") ;
94
95    Function = "function" NAME "
```

```
96              ("  (Parameter  (","  Parameter)*)?  ")"  (returnType:Datatype)?
97                  BLOCK_OPEN
98                      Block
99                  BLOCK_CLOSE;
100
101     Parameter = NAME Datatype;
102
103     /*********************************************************
104      *  Units-Language
105      *********************************************************/
106
107     Datatype = isInt:"integer"
108              | isReal:"real"
109              | isString:"string"
110              | isBool:"boolean"
111              | isVoid:"void"
112              | unit:UnitType;
113
114     UnitType = leftParentheses:"("
115                compoundUnit:UnitType rightParentheses:")"
116              | base:UnitType powOp:"**" exponent:INTEGER
117              | left:UnitType (timesOp:"*" | divOp:"/") right:UnitType
118              | unitlessLiteral:INTEGER divOp:"/" right:UnitType
119              | unit:NAME;
120
121     /*********************************************************
122      *  Expressions-Language
123      *********************************************************/
124
125     Expression =
126             leftParentheses: "(" term:Expression rightParentheses:")"
127          | <rightassoc> left:Expression powOp:"**" right:Expression
128          | UnaryOperator term:Expression
129          | left:Expression (timesOp:"*" | divOp:"/" | moduloOp:"%")
130            right:Expression
131          | left:Expression (plusOp:"+"  | minusOp:"-")
132            right:Expression
133          | left:Expression BitOperator right:Expression
134          | left:Expression ComparisonOperator right:Expression
135          | logicalNot:"not" term:Expression
136          | left:Expression LogicalOperator right:Expression
137          | condition:Expression "?" ifTrue:Expression ":"
138            ifNot:Expression
139          | SimpleExpression
140          ;
141
142     SimpleExpression = FunctionCall
143                      | BOOLEAN_LITERAL // true & false;
144                      | ( INTEGER|FLOAT ) (Variable)?
145                      | string:STRING_LITERAL
146                      | isInf:"inf"
147                      | Variable;
148
149     UnaryOperator = ( unaryPlus:"+"|unaryMinus:"-"|unaryTilde:"~"  );
150
```

```
151    BitOperator = ( bitAnd :"&" | bitXor :"^" |
152                    bitOr :"|" | bitShiftLeft :"<<" |
153                    bitShiftRight :">>" );
154
155    ComparisonOperator = ( lt :"<" | le :"<=" | eq :"==" |
156                           ne :"!=" | ne2 :"<>" | ge :">=" |
157                           gt :">" );
158
159    LogicalOperator = ( logicalAnd :"and" | logicalOr :"or" );
160
161    Variable = name:NAME ( DifferentialOrder )*;
162
163    DifferentialOrder = "\'";
164
165    FunctionCall = calleeName :NAME
166        "(" (Expression ("," Expression )*)? ")";
167
168    /*******************************************************
169    * Equations−Language
170    *******************************************************/
171
172    OdeFunction = ( recordable :"recordable")? "function"
173      variableName :NAME Datatype "=" Expression (";")?;
174
175    OdeEquation = lhs : Variable "=" rhs : Expression (";")?;
176
177    OdeShape = "shape" lhs : Variable "=" rhs : Expression (";")?;
178
179    /*******************************************************
180    * Procedural−Language
181    *******************************************************/
182
183    Block = ( Stmt | NEWLINE )*;
184
185    Stmt = SmallStmt | CompoundStmt;
186
187    CompoundStmt = IfStmt
188                 | ForStmt
189                 | WhileStmt ;
190
191    SmallStmt = Assignment
192              | FunctionCall
193              | Declaration
194              | ReturnStmt ;
195
196    Assignment = lhsVariable : Variable
197        ( directAssignment :"=" |
198         compoundSum :"+="      |
199         compoundMinus :"−="    |
200         compoundProduct :"*=" |
201         compoundQuotient :"/=") Expression ;
202
203    Declaration =
204       ( isRecordable :"recordable")? ( isFunction :"function")?
205        Variable ("," Variable )* Datatype
```

```
206            (" [" sizeParameter:NAME "]")?
207            ("=" rhs:Expression )?
208            (" [[" invariant:Expression "]]")?;
209
210        ReturnStmt = "return" Expression?;
211
212        IfStmt = IfClause
213                ElifClause*
214              (ElseClause)?
215               BLOCK_CLOSE;
216
217        IfClause = "if" Expression BLOCK_OPEN   Block;
218
219        ElifClause = "elif" Expression BLOCK_OPEN Block;
220
221        ElseClause = "else" BLOCK_OPEN Block;
222
223        ForStmt = "for" var:NAME "in" startFrom:Expression "..."
224                endAt:Expression "step" step:SignedNumericLiteral
225                BLOCK_OPEN
226                 Block
227                BLOCK_CLOSE;
228
229        WhileStmt = "while" Expression BLOCK_OPEN Block BLOCK_CLOSE;
230
231        SignedNumericLiteral = (negative:"−"?) (INTEGER|FLOAT);
232    }
```