NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dimitri Plotnikov, M.Sc.RWTH
aus Simferopol, Ukraine

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Abigail Morrison

Tag der mündlichen Prüfung: 07.12.2017

Kurzfassung

Biologische Nervensysteme weisen erstaunliche Komplexität auf. Neurowissenschaftler versuchen, diese Komplexität durch die Modellierung und Simulation der zugrunde liegenden biologischen Prozesse zu erfassen und zu verstehen. Modellkomplexe, die notwendig sind, um die neuromorphen Prozesse abzubilden. Dementsprechend schwierig ist es passende Modelle zu erstellen. Deswegen sind leistungsstarke Werkzeuge notwendig, die es Neurowissenschaftlern ermöglichen, diese Modelle zunächst kompakt, aber auch umfassend auszudrücken und aus diesen Modellen effizienten Code für digitale Simulationen zu generieren.

Domänenspezifische Sprachen erlauben gegenüber General Purpose Programmiersprachen begrenzten und problemorientierten Funktionsumfang an. Die erhöhte Produktivität der Entwickler und verbesserte Qualität der resultierenden Softwaresysteme [VDK98, SEHV12, FHR08] gelten als Vorteile einer DSL. Die erhöhte Produktivität resultiert aus der Tatsache, dass Modelle in einer passend gewählten Notation im Vergleich zu einer äquivalenten Darstellung in einer GPL kompakter sind. Verschiedene Modellierungssprachen für die Computational Neuroscience wurden bereits vorgeschlagen [GCC$^+$10, RCC$^+$11]. Da diese Sprachen jedoch typischerweise Simulatorunabhängigkeit anstreben, unterstützen sie oft nur eine Untermenge der vom Modellierer gewünschten Eigenschaften.

Diese Arbeit präsentiert den Entwurf und die Implementierung der modularen und erweiterbaren domänenspezifischen Sprache NESTML, die Konzepte aus den Neurowissenschaften als vollwertige Sprachkonstrukte zur Verfügung stellt und Neurowissenschaftler so bei der Erstellung von Neuronemodellen für das neuronale Simulationswerkzeug NEST unterstützt.

NESTML wurde mithilfe von MontiCore [GKR$^+$08, Kra10] entwickelt. MontiCore ist eine Language Workbench zur Erstellung von domänenspezifischen Sprachen. MontiCore verwendet und erweitert das Grammatikformat von *ANTLR4* [Par13], das auf dem EBNF-Formalismus [ASU86] basiert, um zusätzliche Konzepte für die Grammatikwiederverwendung. MontiCore stellt eine modulare Infrastruktur für das Parsen von Modellen, den Aufbau der Symboltabllen und zum Prüfen der Kontextbedingungen bereit. Damit können die Entwicklungskosten von NESTML signifikant gesenkt werden.

Abstract

Biological nervous systems exhibit astonishing complexity. Neuroscientists aim to capture this complexity by modeling and simulation of the underlying biological processes. Therefore it is hard to create suitable models which are necessary to depict the neuromorphic processes, which makes it difficult to create these models. Powerful tools are thus needed, which enable neuroscientists to express models in a comprehensive and concise way and generate efficient code for digital simulations.

Domain-specific languages allow limited and problem-oriented functional scope compared to general purpose programming languages. The increased productivity of the developers and improved quality of the resulting software systems [VDK98, SEHV12, FHR08] are regarded as advantages of a DSL. The increased productivity results from the fact that models in a suitably selected notation are more compact in comparison to an equivalent representation in a general purpose language. Several languages for computational neuroscience have been proposed [GCC+10, RCC+11]. However, as these languages often seek simulator independence they typically only support a subset of the features desired by the modeler.

This thesis presents the design and implementation of the modular and extensible domain specific language NESTML, which provides neuroscience domain concepts as first-class language constructs and supports domain experts in creating neuron models for the neural simulation tool NEST.

NESTML is developed using MontiCore [GKR+08, Kra10]. MontiCore is a language workbench for the creation of domain-specific languages. MontiCore expands the grammar format of *ANTLR4* [Par13], which is based on the EBNF formalism [ASU86] with additional concepts for the grammar reuse. MontiCore provides a modular infrastructure for parsing models, building symbol tables and verifying the context-conditions. This allows the development costs of NESTML to be significantly reduced.

Danksagung

An dieser Stelle möchte ich mich bei den lieben Menschen bedanken, die mich während meiner Promotion unterstützt und so zum Erfolg dieser Dissertation beigetragen haben. Mein ganz besonderer Dank gilt meinem Doktorvater Prof. Dr. Bernhard Rumpe für die spannende und lehrreiche Zeit am Lehrstuhl, für die ideenreichen und konstruktiven Diskussionen.

Prof. Dr. Abigail Morrison danke ich herzlich für die Betreuung meiner Arbeit aus der neurowissenschaftlichen Perspektive und die Übernahme des Zweitgutachtens. Ich möchte mich ebenfalls bei Prof. Dr. Klaus Wehrle für die Leitung meines Promotionskomitees und bei Prof. Dr. Thomas Noll für die Durchführung der Theorieprüfung bedanken.

Dr. Jochen Martin Eppler danke ich sehr für das Mentoring sowie für seine fachliche und freundschaftliche Unterstützung während meiner Promotion. Dr. Boris Orth danke ich für die Möglichkeit im Forschungszentrum Jülich arbeiten zu dürfen.

Vielen Dank an Jochen, Guido, Inga, Achim, Katrin, Alexander, Markus und Robert für das Korrekturlesen früher Fassungen der Dissertation.

Aber auch außerhalb der Büroräume mussten Familie und Freunde unter meiner knappen Zeit und meinen wechselhaften Launen leiden. Ich möchte mich daher zunächst entschuldigen und im gleichen Atemzug für die tolle und absolut nicht selbstverständliche Unterstützung bedanken. Durch meine Familie wurde mir nicht nur mein Ausbildungsweg ermöglicht, sondern eben auch die Chance, jederzeit spannende Dinge nebenbei zu erleben.

Ganz besonders möchte ich mich bei meiner Ehefrau Olga bedanken, die mich zu jeder Zeit liebevoll unterstützt und immer wieder motiviert hat, auch wenn das hieß, dass ich viele Samstage, Sonntage, Urlaubs- und Feiertage an der Dissertation gearbeitet und nicht meine Freizeit mit ihr verbracht habe. Dadurch hat sie erheblich zur Fertigstellung dieser Arbeit beigetragen. Meiner Tochter Marie danke ich für ihre freundliche Art, die mich mit einem einzigen Lächeln auch aus dem tiefsten Tal befreien kann.

Mönchengladbach, Dezember 2017
Dimitri Plotnikov

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die klassische Neurowissenschaft untersucht biophysikalische Prozesse, die für das Verhalten eines einzelnen Neurons bzw. höherer Hirnfunktionen, die durch Verschaltung dieser Neuronen zu Netzwerken entstehen, verantwortlich sind. Die ersten experimentellen Studien des Nervensystems wurden bereits Jahrhunderte zuvor gemacht [She91], dennoch war und ist die Beobachtung der Aktivität einer einzelnen Zelle in einer separierten Zellkultur (*in vitro*) oder gar in einem funktionierenden Hirn (*in vivo*) technisch eine herausfordernde Aufgabe. Deswegen benutzen Neurowissenschaftler eine Vielzahl an Methoden und Werkzeugen, um die elektrischen und chemischen Eigenschaften von Neuronen und deren Verbindungen untereinander zu messen. Zu diesen Methoden zählen mikroskopische Untersuchungen, Elektrophysiologie und unterschiedliche bildgebende Verfahren. Da diese Verfahren relativ neu sind, wurden erst im letzten Jahrhundert erste Details über die Struktur und Funktion der Gehirnbausteine bekannt. McCulloch und Pitts [MP43] verwendeten Anfang des letzten Jahrhunderts logische Schaltelemente, um das in Experimenten gemessene Verhalten von Nervenzellen nachzuahmen und vernetztes Verhalten der Nervenzellen zu verstehen.

Es stellte sich aber sehr schnell heraus, dass solche künstlichen Schaltungen zu stark vereinfacht und zu sehr eingeschränkt sind, um Funktionsprinzipien des Gehirns zu untersuchen. Diese Erkenntnis spaltete die Ansätze für die Untersuchung von neuronalen Netzwerken in zwei grundsätzliche Kategorien. Die erste Kategorie basiert auf den frühen Anfängen der neuronalen Netzwerke auf Basis der logischen Schaltungen und ist unter dem Begriff *künstliche neuronale Netzwerke* bekannt. Solche Netzwerke sind so entworfen, dass sie hoch spezialisierte Aufgaben wie beispielsweise Textübersetzung und Klassifizierung von Bildern besonders effizient lösen können. Die zweite Kategorie umfasst die biologischen Modelle, bei denen der Fokus auf dem Modellieren von biochemischen Prozessen in einem Gehirn liegt. Solche Modelle sind unter dem Begriff *biologische neuronale Netzwerke* bekannt.

1.1 Motivation und Kontext

Der Begriff eines *Modells* wird in der Softwaretechnik abhängig von dem Kontext mit unterschiedlicher Bedeutung eingesetzt. So gibt es Vorhersagemodelle, die den Ablauf

der Entwicklung eines Softwareproduktes beschreiben. Produktmodelle und Testmodelle [Rum12, Rum16] spezifizieren dann das modellierte System und definieren Testkriterien anhand derer das aus Modellen resultierende System validiert werden kann. Eine gute Kategorisierung der Modelle ist in [Sch00] und [SPHP02] zu finden. Die wesentlichen Eigenschaften des Modellbegriffs, die für diese Arbeit relevant sind, werden wie folgt zusammengefasst:

- Ein Modell ist seinem Wesen nach eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems [Sta73].
- Ein Modell ist eine vereinfachte, auf ein bestimmtes Ziel hin ausgerichtete Darstellung der Funktion eines Gegenstands oder des Ablaufs eines Sachverhalts, die eine Untersuchung oder eine Erforschung erleichtert oder erst möglich macht [Bal00].

Die wesentlichen Szenarien für die Verwendung solcher Modelle in der Entwicklung eines Softwareprodukts lassen sich in die folgenden Kategorien einteilen:

- Implementierungsmodelle
- Testmodelle
- Dokumentationsmodelle
- Kommunikationsmodelle

Sicherlich ist die Modellierung und Bildung der Modelle nicht auf das Gebiet der Softwaretechnik begrenzt. Modelle werden in Naturwissenschaften schon seit frühen Zeiten benutzt. Auch in den Neurowissenschaften kommt diese Technik zur Anwendung. Das Teilgebiet Computational Neuroscience [CKS93] erstellt unter anderem Modelle von Nervenzellen (*Neuronen*) und deren Verbindungen (*Synapsen*), mit dem Ziel diese Modelle in einem Computer zu simulieren. Diese Modelle spiegeln bestimmte Aspekte der anatomischen und physiologischen Eigenschaften der biologischen Vorbilder wider. Abhängig von den konkreten Zielen der Studie sind unterschiedliche Aspekte des Verhaltens oder der Struktur eines einzelnen Neurons oder eines Netzwerkes von Neuronen relevant. Daher erstellen Neurowissenschaftler vereinfachte Modelle, die auf die relevanten Eigenschaften des zu erforschenden Verhaltens reduziert sind. Diese Vorgehensweise hat dazu geführt, dass mittlerweile eine Vielzahl solcher Modelle existiert. Der Detailgrad dieser Modelle erstreckt sich von *Mehrsegment-Modellen* (engl: multicompartment models) [Ral64], die viele biologische und morphologische Details der biologischen Neuronen beinhalten, bis zu reduzierten *Punktneuronen* (engl: point neurons) [Mac12], bei denen Neuronen keine räumliche Ausdehnung haben und Synapsen nur durch gewichtete Verbindungen ohne morphologische Details modelliert werden. Abbildung 1.1 visualisiert diese Unterscheidung und demonstriert unterschiedliche Ansätze, um die aus den physikalischen

Messungen und Untersuchungen ermittelte Struktur eines Neurons, in unterschiedlichen Detailgraden nachzubilden.

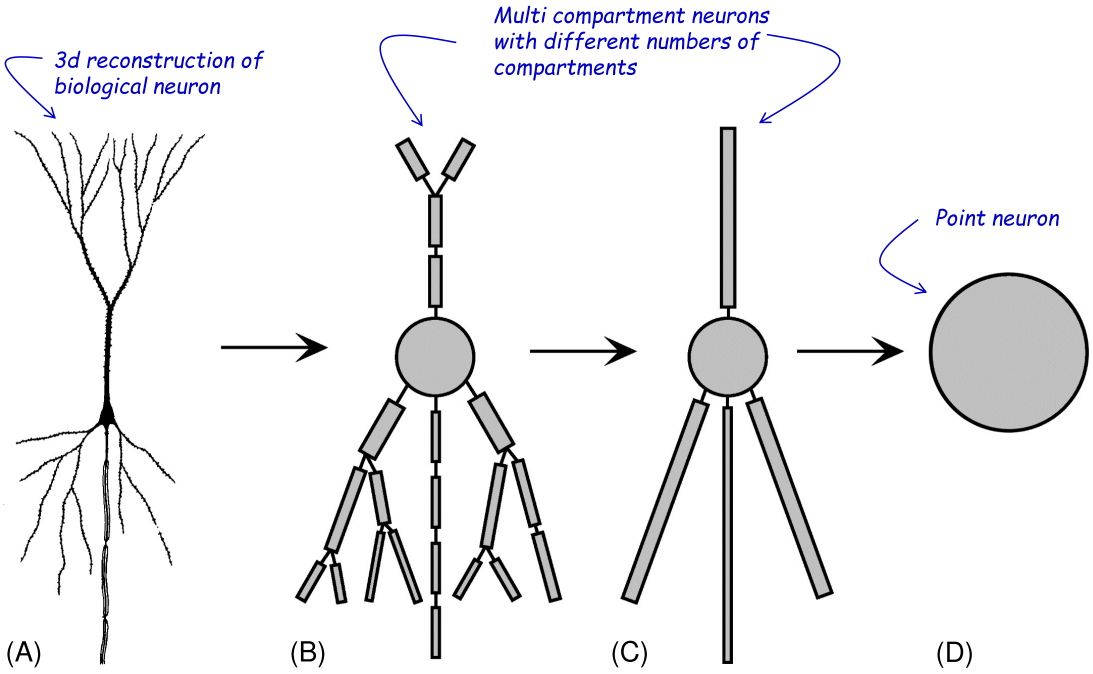


Abbildung 1.1: Unterschiedliche Abstraktionsstufen von Neuronenmodellen [DA01]. Die Darstellung variiert in der Anzahl von den diskreten Kompartimenten. Ausgehend von einer 3D rekonstuktion eines realen Neurons aus der Großhirnrinde vereinfacht sich der Abstraktionsgrad. (A) Ein Pyramidal-Neuron aus der Großhirnrinde. (B) und (C) Mehrsegment-Neuronen mit unterschiedlicher Anzahl von Segmenten. (D) Ein Punktneuron.

Unabhängig vom Detailgrad des Neuronenmodells wird seine Dynamik typischerweise durch Differenzialgleichungen beschrieben. Eine Simulation solcher Neuronenmodelle (d.h. die Evolution der entsprechenden Differenzialgleichungen über die Zeit, vgl. Abschnitt 2.3) erlaubt es, Experimente *in silico* als eine Simulation durchzuführen. Simulationen ermöglichen es, Hypothesen in einer stabilen und kontrollierten Umgebung zu verifizieren. Die Simulation von Neuronenmodellen mit verschiedenen Detailgraden erfordert jeweils andere technische Infrastruktur, um Neuronen an sich zu repräsentieren und die Verbindungen zwischen Neuronen effizient zu speichern bzw. die Kommunikation zwischen Neuronen zu simulieren. Deswegen existieren verschiedene Simulatoren, die sich jeweils für die Simulation eines bestimmten Ausschnittes des Spektrums der möglichen Neuronentypen besonders gut eignen. Als schwierig gestaltet sich auch der Vergleich der Ergebnisse zwischen unterschiedlichen Simulatoren, da die Modelle für je-

den einzelnen Simulator manuell und jedes Mal neu implementiert werden [CBB+12] bzw. simulatorspezifische Effekte in Betracht gezogen werden müssen.

1.2 Modellierungssprachen

Oft werden ausführbare neuronale Modelle [Rum02] als eine simulatorspezifische Implementierung in einer GSL erstellt. Solche Modelle können dann nur in diesem Simulator ausgeführt. Um die Modellteilung und Reproduzierbarkeit von neurowissenschaftlichen Experimenten zu vereinfachen, existieren Modellierungsansätze NeuroML [GCC+10], LEMS [CGC+14] und NineML [GRHLF11, RCC+11], die in Kapitel 3 näher beschrieben werden. In der Regel gehören zu einer Modellierungssprache Werkzeuge, mit denen sich eine ausführbare Implementierung für einen Simulator generieren lässt. Die meisten dieser Sprachen sind technologieagnostisch entworfen, daher können sie nicht direkt von den Vorteilen eines konkreten Simulators profitieren. Dies resultiert in einer schlechteren Performance bzw. Genauigkeit der simulierten Modelle im Vergleich zu einer handgeschriebenen Umsetzung desselben Models für einen konkreten Simulator.

NEST [GD07] ist ein neuronaler Simulator für große neuronale Netzwerke, bestehend aus puls-gekoppelten Punktneuronen. Die Quellen von NEST sind als Open Source verfügbar[1]. Aufgrund der hybriden Parallelisierung kann NEST sowohl auf einem leistungsschwachen Laptop als auch auf einem Supercomputer effizient ausgeführt werden [HKM+12, KSE+14]. Mit über 450 publizierten Studien, in denen NEST verwendet wurde, und 360 aktiven Mitgliedern auf der Mailing-Liste[2] gehört NEST zu den am meisten verbreiteten Simulatoren für biologische neuronale Netzwerke. Aufgrund seiner Zuverlässigkeit und Popularität wurde NEST als Simulator für Netzwerke auf der Skala von ganzen Gehirnen im Rahmen des EU Flagship Projektes *Human Brain Project* [AEM+16] ausgewählt.

Zu Beginn dieser Arbeit gab es 36 Neuronenmodelle in NEST. Jedes dieser Neuronenmodelle ist als eine handgeschriebene C++-Klasse implementiert, die eine vorgeschriebene und nicht transparente Modell-API erfüllt. Die Entwicklung neuer Neuronenmodelle erfordert Expertenwissen sowohl in den Neurowissenschaften als auch in der Programmierung mit C++ bzw. etlicher Implementierungsdetails von NEST, um Neuronenmodelle in die NEST-Infrastruktur einzubetten. Die starke Kopplung zwischen Modellen und der NEST-API hat zur Folge, dass bei einer Änderung der NEST-Infrastruktur bzw. Modell-API in NEST alle Modelle manuell angepasst werden müssen. Dies erhöht den Aufwand für die Wartung und Aktualisierung von Neuronenmodellen in NEST.

Die C++-Implementierung von Neuronenmodellen vermischt die Modellspezifikation (d.h. die Modellgleichungen und Algorithmen zur Beschreibung der Neuronendynamik) mit deren Implementierung. Modelle, die durch lineare Differenzialgleichungen mit

[1] https://github.com/nest/nest-simulator
[2] nest_developer@nest-initiative.org

konstanten Koeffizienten definiert sind, werden exakt gelöst [RD99]. Sobald die exakte Lösung als C++-Code ausgedrückt ist, lässt diese nur noch schwer Rückschlüsse auf die ursprünglichen Differenzialgleichungen zu. Einerseits erschwert dies das Verständnis der Neuronenmodelle. Andererseits entsteht dadurch eine starke Kopplung von den ansonsten modularen Modellierungskonzepten, die für die Spezifikation von dem Neuron verwendet werden. Dadurch wird die Wiederverwendung der Modellteile in anderen Simulatoren negativ beeinflusst oder unmöglich gemacht.

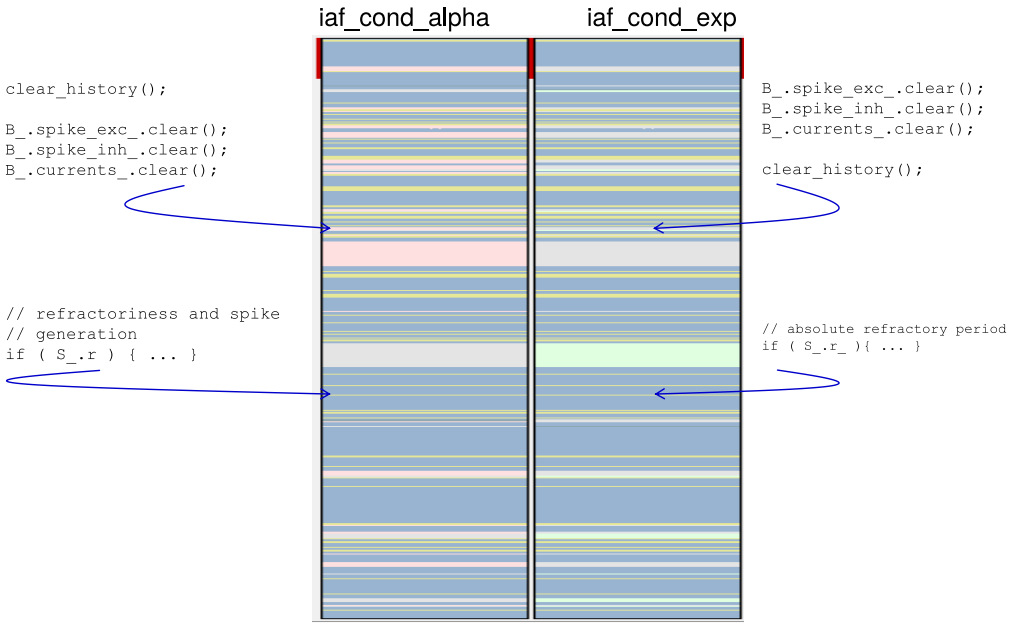


Abbildung 1.2: Schematischer Vergleich von zwei Neuronenmodellen aus dem NEST Quellcode mithilfe von *KDiff*. Die in dunkel dargestellten Dateiabschnitte stimmen in beiden Modellen überein.

Aufgrund der fehlenden Modularität des C++-Code von Neuronenmodellen in NEST werden neue Neuronenmodelle auf Basis des *Copy&Paste*-Verfahrens [JH07] erstellt. Ausgehend von einem Modell, das am besten passt und bereits existiert, wird eine Kopie der Implementierung erstellt und entsprechend der neuen Anforderungen ergänzt und modifiziert. Modelle werden durch Neurowissenschaftler entwickelt, die oft weder Experten in Programmierung sind, noch sich mit der internen Struktur der Modellinfrastruktur von NEST auskennen. Daher werden Neuronenmodelle durch diese Art der Erzeugung redundant, mit suboptimaler Performance und ungenügender Dokumentation erstellt. Vorläufigen Untersuchungen existierender NEST-Modelle offenbarten Fälle, in denen über 90% der Implementierung zweier Modelle gleich oder sehr ähnlich waren.

Der mithilfe von *KDiff3*[3] erstellte Vergleich zweier Neuronenmodelle in Abbildung 1.2 visualisiert dies an einem Beispiel. Die Grafik zeigt schematisch zwei C++-Dateien, die unterschiedliche Neuronenmodelle implementieren. Auffällig sind die dunkel markierten Bereiche, die komplett identischen Codeabschnitten entsprechen. Auch manche als unterschiedlich markierte Bereiche sind semantisch sehr ähnlich. Die Abbildung zeigt dies exemplarisch an zwei Fallbeispielen. Im ersten wird die Reihenfolge der Anweisungen permutiert. Im zweiten wird der Kommentar minimal angepasst. Dennoch ändert sich in beiden Fällen das Verhalten des Neurons aufgrund dieser Änderungen nicht.

Trotz der bereits existierenden Ansätze zur Modellierung von Neuronen mit den Markup-Sprachen NineML, NeuroML und LEMS bzw. die in die Simulatoren integrierten Sprachen stellen sie keine akzeptable Alternative zur Entwicklung neuer Modellierungssprachen dar. Markup-Sprachen sind sehr wortreich, was das Nachvollziehen und Erstellen von Neuronenmodellen stark erschwert. Die in Simulatoren eingebetteten Sprachen sind zwar syntaktisch besser aufgebaut, dennoch ist deren Wiederverwendbarkeit aufgrund der engen Verzahnung mit dem jeweiligen Simulator eingeschränkt.

1.3 Der NESTML-Ansatz

Im Laufe dieser Arbeit wurde die modular aufgebaute Sprache Domain Specific Language (DSL) NESTML entwickelt. NESTML eignet sich für die Modellierung von Punktneuronen besonders gut. Dennoch wird die Sprache so entworfen, dass ihre Komponenten sich in anderen Kontexten (z.B. für die Modellierung von Synapsen oder komplexeren Neuronenmodellen) einfach wiederverwenden lassen.

Auch wenn in der Literatur keine einheitliche Definition einer DSL existiert, fassen die folgenden Charakterisierungen die wesentlichen Eigenschaften einer DSL zusammen:

- A DSL is a language designed to be useful for a limited set of tasks, in contrast to general-purpose languages that are supposed to be useful for much more generic tasks, crossing multiple application domains [JB06].
- A domain-specific language is a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [VDKV00].

Das wichtigste Ziel von NESTML ist es, eine modulare, leichtgewichtige und einfach zu erlernende Sprache zur Beschreibung von Punktneuronen für Neurowissenschaftler zur Verfügung zu stellen. NESTML erlaubt es, Neuronenmodelle in einer präzisen und ausdrucksstarken Syntax zu formulieren. Die konkrete Syntax von NESTML ist primär den Neurowissenschaftlern vertraut. NESTML besteht aus verschiedenen Subsprachen,

[3] http://kdiff3.sourceforge.net/

die für die Modellierung einzelner Modellaspekte verantwortlich sind. Die Spezifikation der im Neuronenmodell vorkommenden Differenzialgleichungen kann direkt in einer mathematischen Notation abgefasst werden. Diese Gleichungen werden mithilfe eines Frameworks für symbolische Mathematik analysiert und stets optimal gelöst. Die eingebettete prozedurale Sprache erlaubt es, komplexe Kontrolllogik auszudrücken, die für die Implementierung der Zustandsaktualisierung notwendig sind. Physikalische Einheiten können direkt als Datentypen von Variablen verwendet werden.

Die Einfachheit und Abstraktion der NESTML-Syntax gewährleistet gute Verständlichkeit von Neuronenmodellen in NESTML. Der hohe Abstraktionsgrad garantiert eine klare Trennung zwischen der Modellspezifikation und der Modellimplementierung. Der NEST-Codegenerator produziert eine performante C++-Implementierung aus NESTML-Modellen, die dynamisch in NEST integriert wird. Andere Zielplattformen wie beispielsweise NeuroML [GCC+10] könnten durch weitere Codegeneratoren unterstützt werden.

Modularisierungskonzepte, die direkt in die Sprache eingebaut sind, vereinfachen die Modellspezifikation. Desweiteren fördern sie die Wiederverwendung von qualitätsgesicherten und getesteten Komponenten, anstatt diese jedes mal neu zu implementieren. Auch die Transformation von NESTML-Modellen in andere Notationen wird im Vergleich zu in C++ geschriebenen Modellen einfacher. NESTML ist im Vergleich zu C++ besser strukturiert, frei von technischen Details und einfacher in der Sprachstruktur. Daher lassen sich entsprechende Transformationen zu anderen Notationen einfacher implementieren.

NESTML wurde mithilfe von MontiCore [KRV07, KRV08] entwickelt. MontiCore erlaubt es, DSLs agil und modular aufzubauen, um sie schnell an neue Anforderungen anpassen zu können. Ausgehend von einer Grammatikdefinition, generiert MontiCore eine Sprachverarbeitungsinfrastruktur inklusive Lexer, Parser, Symboltabellen [ASU86] und Generierungsinfrastruktur [Sch12].

Abbildung 1.3 fasst den NESTML-Ansatz zusammen. Einerseits wird NESTML benutzt, um existierende NEST-Modelle durch aus NESTML-Modellen generierten Code zu ersetzten. Andererseits wird NESTML verwendet, um neue Modelle in NEST zu integrieren. Somit wird NESMTL zur Schnittstelle zum Einbinden neuer und als auch auch existierender Neuronenmodelle. Desweiteren wird es durch diesen modellbasierten Ansatz zum ersten Mal möglich, die abstrakten NESTML-Modelle in die Darstellungsformate anderer Simulatoren zu exportieren. Dadurch kann ein in NEST ausgeführtes Experiment einfacher in anderen Simulatoren wiederholt und validiert werden. Es existieren zwar bereits vergleichbare Modellierungsansätze und Sprachen, diese lassen sich aus folgenden Gründen jedoch nicht für NEST verwenden:

Modularität: Die Sprachen sind nicht modular aufgebaut, sodass sich Teile der Sprachdefinitionen nicht wiederverwenden lassen.

Notationsbarriere: Die Sprachen verfügen über eine sehr schwer les- und erstellbare Notation für die Neuronenmodelle.

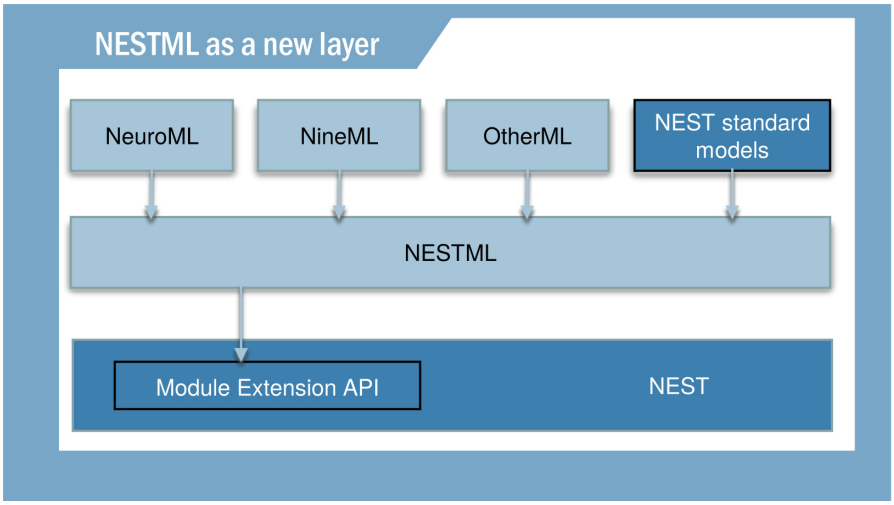


Abbildung 1.3: NESTML als Fassade für die Integration von Neuronenmodellen in NEST.

Werkzeugbindung: Die einem bestimmten Simulator eigene Sprache ist kaum außerhalb des jeweiligen Simulators nutzbar.

Die wichtigste Fragestellung dieser Arbeit lautet daher: *Wie muss eine problemadäquate, domänenspezifische Sprache für die Modellierung von biologischen Neuronen aussehen, die sich zur Simulation auf Basis des NEST-Simulators eignet?*
Aus dieser zentralen Frage lassen sich folgende Unterfragen ableiten:

(FF1): Wie sieht eine leichtgewichtige und einfach zu erlernende Neuronenbeschreibungssprache aus?

(FF2): Wie können Neuronenmodelle in eine für NEST ausführbare Form übersetzt werden?

(FF3): Wie sehen modulare und wiederverwendbare Sprachkomponenten aus?

(FF4): Welche Konzepte helfen dabei, Neuronenmodelle wiederverwendbar zu machen?

(FF5): Wie sieht die Methodik aus, um Modelle in der neuen DSL zu erstellen und in den NEST-Simulator zu integrieren?

1.4 Wichtigste Resultate

Insgesamt setzt sich die für NESTML entwickelte Umsetzung aus folgenden Teilaspekten zusammen:

- Einer modularen und anpassbaren DSL zur Beschreibung von Neuronen und Komponenten (vgl. Abschnitt 5.2).
- Einer modularen DSL zur Beschreibung von imperativer Programmlogik (vgl. Abschnitt 5.3). Diese DSL lehnt sich syntaktisch an die Programmiersprache Python an, um die Lernhürde für die Anwender von NESTML zu erleichtern, da Python auf dem Gebiet der Neurowissenschaften weit verbreitet ist [MBD+15].
- Einer modularen DSL für die Spezifikation von Differenzialgleichungen in einer mathematischen Notation (vgl. Abschnitt 5.4). Differenzialgleichungen werden analysiert und für eine performante Simulation stets optimal gelöst.
- Einer modularen DSL für die Spezifikation von Ausdrücken in einer an Python angelehnten Notation (vgl. Abschnitt 5.5). Diese DSL erlaubt eine nahtlose Interoperabilität mit Python.
- Einer modularen DSL zur Beschreibung und einer Laufzeitumgebung für die Korrektheitsprüfung von physikalischen Einheiten (vgl. Abschnitt 5.6). Diese DSL ermöglicht es, physikalische Einheiten als vordefinierten Datentypen zu verwenden. Die Ausdrücke, in denen Variablen von einem Einheitentyp vorkommen, werden auf die Typenkorrektheit automatisch überprüft.
- Eines Codegenerators für den NEST-Simulator (vgl. Kapitel 8). Mithilfe dieses Generators werden NESTML-Modelle in eine ausführbare Form übersetzt.
- Sprachverarbeitungswerkzeugen, die modular aufgebaut und auf die potenzielle Erweiterung ausgelegt sind. Dies beinhaltet Symboltabellen, Kontextbedingungen für die semantische Analyse von Neuronenmodellen, ein mathematisches Rahmenwerk zur Analyse der Differenzialgleichungen, einen Codegenerator für den NEST-Simulator und die Methodik für die Entwicklung neuer Neuronenmodelle.

Im Rahmen dieser Arbeit wurde NESTML in zwei Workshops mit insgesamt 27 Teilnehmern evaluiert. Dabei wurden die Hypothesen über die Benutzerfreundlichkeit von NESTML anhand von Umfragen validiert (vgl. Kapitel 9).

1.5 Aufbau der Arbeit

Diese Arbeit ist wie folgt strukturiert:

Kapitel 1 stellt den Kontext dieser Ausarbeitung, Forschungsfragen und Ziele vor.

Kapitel 2 stellt kanonische mathematische Modelle für das Modellieren von Punktneuronen vor, die auf Basis von äquivalenten elektrischen Schaltkreisen erstellt werden.

Kapitel 3 stellt die wichtigsten Anforderungen an Modellierungssprachen in dem neurowissenschaftlichen Kontext vor. Die Anforderungen werden anhand einer Literaturrecherche definiert und anhand einer szenariobasierten Analyse motiviert. Anschließend untersucht das Kapitel existierende Modellierungsansätze anhand der erarbeiteten Anforderungen und fasst die Vor- und Nachteile dieser Ansätze zusammen.

Kapitel 4 stellt die wesentlichen Mechanismen des MontiCore-Frameworks vor. Unter anderem führt das Kapitel die Begriffe einer Grammatik, eines Metamodells, eines Visitors, der Codegenerierung und Kontextbedingungen ein.

Kapitel 5 beschreibt die domänenspezifische Sprache NESTML für die Spezifikation von Punktneuronen. Dabei werden alle vorkommenden Subsprachen ausführlich erläutert.

Kapitel 6 beschreibt die Umsetzung von NESTML und aller Subsprachen mit der MontiCore Language Workbench.

Kapitel 7 enthält eine detaillierte Vorgehensweise für die Entwicklung neuer Neuronenmodelle mit NESTML. Anschließend werden Kontextbedingungen von NESTML erklärt.

Kapitel 8 beschreibt den Aufbau und Funktionsweise des Codegenerators für den NEST-Simulator.

Kapitel 9 stellt die Resultate der umfragebasierten Evaluierung von NESTML vor.

Kapitel 10 fasst die Ausarbeitung zusammen und diskutiert die wesentlichen Aspekte. Desweiteren enthält das Kapitel einen Ausblick auf mögliche Erweiterungen von NESTML.

Kapitel 2

Mathematische Modellierung von biologischen Neuronen

In diesem Kapitel werden die biologischen und mathematischen Grundlagen erläutert, die für das Verständnis der Modellbildung im Kontext von biologischen neuronalen Netzwerken notwendig sind. In Abschnitt 2.1 wird das Nervensystem mit seinen Hauptbestandteilen, Neuronen und Synapsen, aus der biochemischen Perspektive erläutert. In Abschnitt 2.2 und 2.3 werden die mathematische Grundlagen erläutert, die verwendet werden, um die biologischen Eigenschaften von Neuronen zu spezifizieren.

2.1 Neuronen als Teil des Nervensystems

Das Nervensystem ist bei Tieren und Menschen dafür verantwortlich, Signale und Reize aus der Umwelt zu empfangen, zu verarbeiten und darauf zu reagieren. Dazu zählen visuelle Reize, die im Auge empfangen werden, akustische Reize, olfaktorische Reize, gustatorische Reize, die als Geschmack empfangen werden, haptische Reize, die durch den Tastsinn empfangen werden, und Reize, die das Gefühl des Gleichgewichts beeinflussen und durch den vestibulären Sinn empfangen werden. Diese Reize kommen von außerhalb des Organismus. Aber nicht nur die Signale von außerhalb des Körpers spielen bei der Verarbeitung im Nervensystem eine Rolle. Auch innerhalb des Körpers können Reize entstehen und wahrgenommen werden. Dies sind Reize, die zum Beispiel durch Organe, Muskeln oder Körperbewegungen ausgelöst werden [BSW07].

Die Aufgabe des Nervensystems ist es, die aufgenommenen Reize zu verarbeiten, aufeinander zu beziehen und mit Veränderungen des Körpers zu reagieren. Dafür kommen sowohl bewusste Reaktionen wie z.B. Bewegungen der Muskulatur als auch unbewusste Reaktionen wie beispielsweise Organtätigkeiten infrage. Das Nervensystem ist somit das zentrale Element zur Steuerung des Körpers.

Das Nervensystem wird in das zentrale Nervensystem (zentrale Nervensystem (ZNS)) und das periphere Nervensystem (periphere Nervensystem (PNS)) unterteilt. Das ZNS wird aus dem Gehirn und dem Rückenmark gebildet. Es hat drei Funktionen: Integration, Koordination und Regulierung. Integration beinhaltet die Sammlung aller eingehenden Reize (Afferenzen) sowohl von innerhalb des Körpers als auch von außerhalb. Koordi-

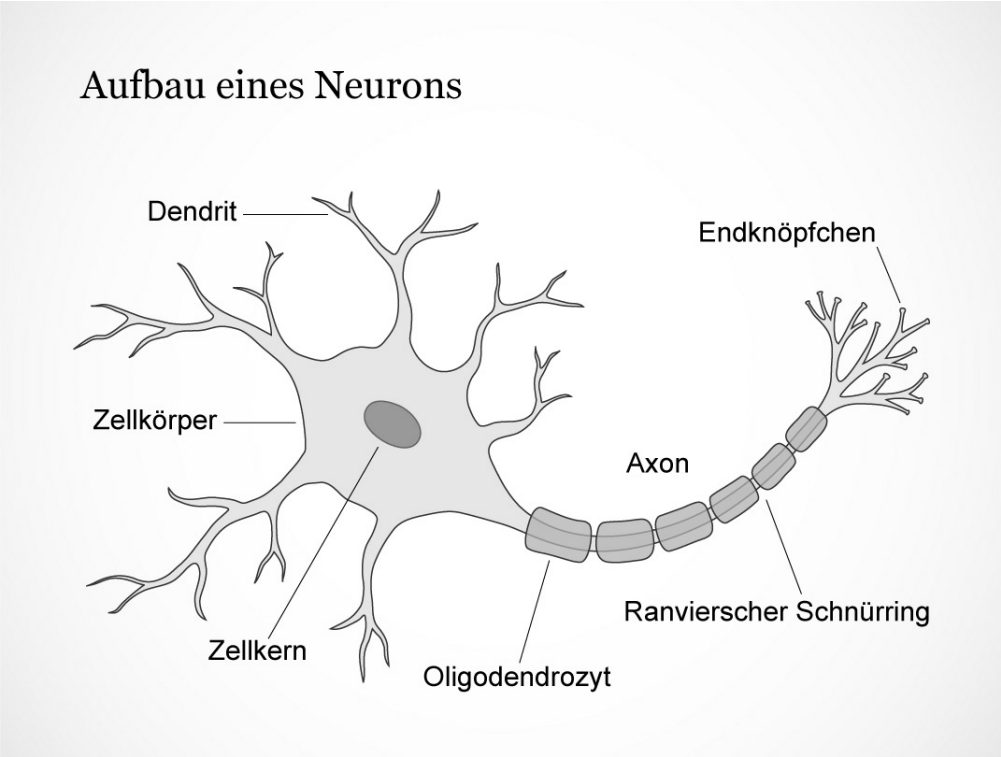


Abbildung 2.1: Schematischer Aufbau eines Neurons [eV12].

nation umfasst sämtliche Bewegungen des Körpers. Schließlich bestimmt Regulierung die Steuerung der organischen Abläufe. Das PNS umfasst den restlichen Teil des Nervensystems ohne das Gehirn und das Rückenmark. Das PNS ist direkt mit dem ZNS verbunden. Es hat den Zweck das ZNS mit den anderen Organen und Körperteilen zu verknüpfen.

Anatomisch setzt sich das gesamte Nervensystem, und damit sowohl das ZNS als auch das PNS, aus Nervenzellen (Neuronen) und Gliazellen zusammen. Gliazellen bilden eine stützende Matrix für die Neuronen und sind an Stoffwechselreaktionen der Neuronen sowie deren Nährstoffversorgung beteiligt. Neuronen dienen der Verarbeitung von Signalen im Nervensystem. Sie sind die Zellen, die die wesentlichen Funktionen des Nervensystems ermöglichen. Ein Neuron ist in der Lage Signale von anderen Neuronen zu empfangen und Signale an andere Neuronen weiterzugeben. Im Nervensystem sind mehrere Milliarden Neuronen verknüpft. Sie kommunizieren miteinander und beeinflussen sich gegenseitig, sodass die komplexen Abläufe im Körper abgebildet und verarbeitet werden können. Der typische Aufbau eines Neurons ist in Abbildung 2.1 skizziert. Ein Neuron besteht aus Dendriten, einem Zellkörper (Soma) mit Zellkern und einem Axon.

Bei einem Signal spricht man auf der Ebene eines Neurons von einer Erregung, die meist aus Potenzialänderungen besteht und innerhalb des Neurons sowie von Neuron zu Neuron weitergegeben werden kann. Die Erregungen von verschiedenen Neuronen werden dabei über die Dendriten in einem Neuron empfangen und laufen im Soma zusammen. Hat sich ein ausreichend großes Potenzial am Neuron gebildet, wird ein Aktionspotenzial (*Spike*) ausgelöst und über das Axon an verbundene Neuronen oder andere Empfänger wie Muskel- oder Drüsenzellen weitergeleitet. Man spricht auch davon, dass das Neuron *feuert*. Dieser Spike wird dann zu allen über Synapsen verbundenen Neuronen, die als *postsynaptische* Neuronen bezeichnet werden, übertragen. Auch bei diesen postsynaptischen Neuronen kommt es zu einer Membranpotenzialverschiebung, die möglicherweise in einem weiteren Spike resultiert. Ein Signal, das in Form eines Aktionspotenzials ankommt, führt zu einer kurzfristigen Verschiebung des Membranpotenzials. Die Richtung der Verschiebung hängt vom Typ des sendenden Neurons ab, das als *präsynaptisches* Neuron bezeichnet wird. Man unterscheidet zwischen erregenden (*excitatorischen*) Neuronen, die eine positive Verschiebung des postsynaptischen Membranpotenzials verursachen, und hemmenden (*inhibitorischen*) Neuronen, die eine negative Verschiebung des postsynaptischen Membranpotenzials verursachen.

Nach dem Feuern des Spikes bleibt das Neuron für eine bestimmte Zeit inaktiv. Diese Zeit wird als Refraktärzeit (engl: refractory period) bezeichnet [NMWF01]. Während dieser Zeit stellt die Zelle ihr Ruhepotenzial wieder her und es kann kein weiteres Aktionspotenzial ausgelöst werden. Beim Wiederherstellen des Ruhemembranpotenzials kann es vorkommen, dass die Zellmembran *hyperpolarisiert* wird, also das Membranpotenzial über das Ruhepotenzial hinausgeht und anschließend erst wieder dorthin zurückkehrt. Wenn Erregungen das Neuron erreichen, ohne ein Aktionspotenzial auszulösen, bildet sich die ausgelöste Potenzialdifferenz nach kurzer Zeit wieder zurück. Um ein Aktionspotenzial auszulösen, müssen also mehrere Reize in kurzer Zeit an einem Neuron eintreffen. Da das Verhalten des Neurons nur bis zum Erreichen eines bestimmten Potenzials, der sogenannten Spike-Schwelle, genau spezifiziert wird, bezeichnet man diese Art von Neuronendynamiken als eine unterschwellige Dynamik (engl: subthreshold dynamics).

Neuronen können unterschiedlich aufgebaut sein. Dabei kommt es maßgeblich auf die Anzahl der Fortsätze an. Abbildung 2.2 fasst die wesentlichen Neuronentypen zusammen. Unipolare Neuronen besitzen lediglich einen Fortsatz, der in der Regel dem Axon entspricht. Diese Neuronen findet man zum Beispiel in der Netzhaut des Auges. Bipolare Neuronen sind mit zwei Fortsätzen, einem Axon und einem Dendriten, ausgestattet. Sie dienen der Informationsübertragung verschiedener Sinne. Die Formen der Neuronen, die am häufigsten vorkommen, sind die multipolaren Neuronen mit mehreren Dendriten und einem Axon. Die letzte Art besteht aus *pseudounipolaren* Neuronen. Sie sind den bipolaren Neuronen ähnlich, allerdings gehen der Dendrit und das Axon nahe dem Zellkörper ineinander über, sodass eine Erregung nicht das ganze Soma durchläuft, sondern sie springt direkt vom Dendrit zum Axon.

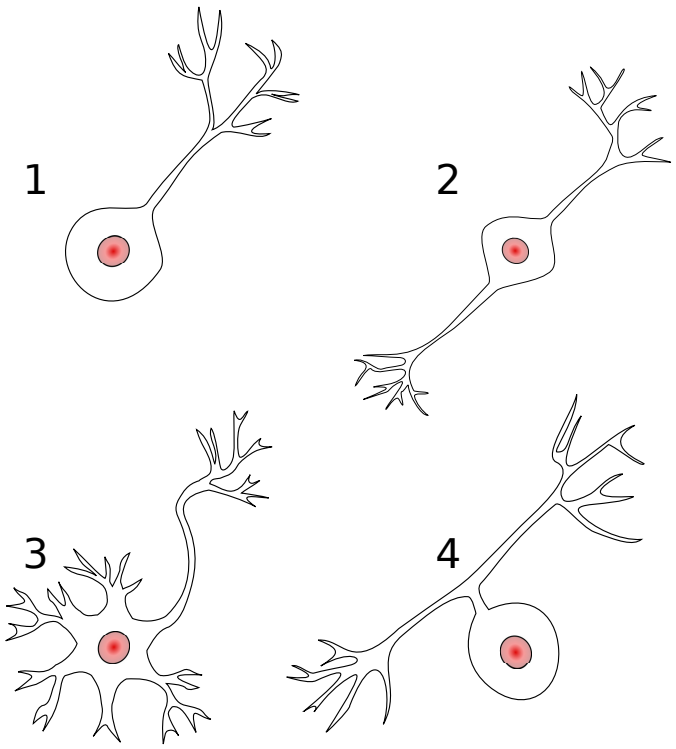


Abbildung 2.2: Aufbau verschiedener Neuronen. 1: unipolares Neuron; 2: bipolares Neuron; 3: multipolares Neuron; 4: pseudo-unipolares Neuron.

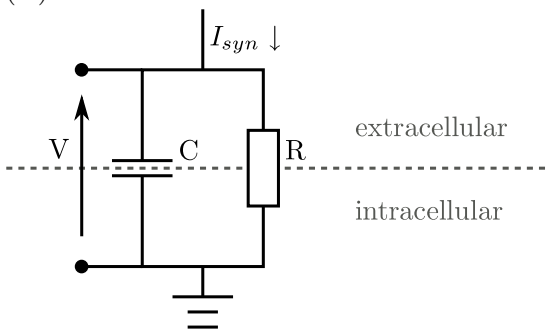
2.2 Mathematische Spezifikation der Neuronendynamik

In der *Computational Neuroscience* versteht man unter Neuronenmodellen mathematische Beschreibungen der biochemischen Abläufe im Nervensystem. Dabei werden charakteristische Eigenschaften eines Neurons wie beispielsweise das Membranpotenzial oder der Ionenstrom in der Membran mithilfe von mathematischen Gleichungen beschrieben. Diese Gleichungen können komplex aufgebaut sein und sowohl voneinander, als auch von der Zeit abhängen. Es gibt unterschiedliche Modelle, die biologische Vorgänge aus verschiedenen Perspektiven und mit der unterschiedlichen Genauigkeit abbilden. Einige Modelle verzichten zugunsten der Einfachheit und höherer Performance auf Genauigkeit. Andere Neuronenmodelle sind verhältnismäßig genauer, was jedoch oft zu höheren Rechenkosten führt. Neue Neuronenmodelle werden entwickelt, um neue und präzisere Forschungsdaten einzubeziehen. Bei der Beobachtung von Neuronen über einen längeren Zeitraum und mit verschiedenen Stimulationsprotokollen können die Ergebnisse unter anderem als eine Funktion von Aktionspotenzialen über die Zeit (sogenannte *Spike-Trains*) dargestellt werden. Verschiedene Neuronen und verschiedene Stimulationen füh-

ren zu unterschiedlichen Spike-Trains [GKNP14]. Je größer der Detailgrad des jeweiligen Neuronenmodells ist, desto genauer kann es die beobachteten Spike-Trains nachahmen.

In den Neurowissenschaften stehen große Datenmengen aus verschiedenen *in vivo* und *in vitro* Experimenten zur Verfügung. Diese Daten erlauben genauere Charakterisierungen der internen Abläufe in den Neuronen und insbesondere der Spike-Trains der unterschiedlichen Arten von Neuronen. Die gemessenen elektrischen und chemischen Eigenschaften der Neuronen stellen eine Basis dar, um das Verhalten durch passende Modelle zu reproduzieren. 3D-Rekonstruktionen von realen Neuronen in Form von elektrisch gekoppelten Kompartimenten erlauben eine exakte Modellierung der Vorgänge in komplexen morphologischen Neutronenstrukturen. Die Arbeit von Lapicque [Lap07] und später Hodgkin und Huxley [HH52] bilden die Grundlagen zur Neuronenmodellierung mit biologisch plausiblen Eigenschaften durch Differenzialgleichungen. Die Veränderung des Membranpotenzials des Neurons kann mithilfe eines äquivalenten elektrischen Schaltkreises untersucht werden, indem ein Schaltkreis bestehend aus einem Widerstand und Kondensator [TS90] erstellt wird. Das Membranpotenzial des Neurons bzw. eines Kompartiments entspricht in diesem Modell der Spannung am Kondensator `C`, Ionenkanäle entsprechen dem Widerstand `R` und externe Eingaben werden durch den Strom I_{syn} modelliert. Wenn ankommende Spikes so lange aufsummiert werden, bis eine Schranke überschritten wird und ein Spike gefeuert wird, spricht man von einem *Integrate-and-Fire* Neuronenmodell. Mithilfe der Kabeltheorie [JNT75] können die einzelnen Schaltkreise im Zusammenhang simuliert werden. Neuronenmodelle, die aus genau einem Kompartiment bestehen, werden als Punktneuron bezeichnet.

(A)



(B)

$$C \cdot \frac{d}{dt}V = -\frac{1}{R}V + I_{\text{syn}}$$

Abbildung 2.3: Exemplarische Darstellung eines Hodgkin-und-Huxley Schaltkreises [HH52] zur Modellierung des Membranpotenzials: (A) Schaltkreisdiagramm (B) Die Differenzialgleichung, die anhand der Kapazität C, des Widerstandes R und eines externen Stroms I_{syn} den Verlauf der Spannung V über die Zeit bestimmt [TS90].

Ein exemplarischer Schaltkreis wird in Abbildung 2.3 (A) gezeigt. Die Differenzialgleichung in Abbildung 2.3 (B) spezifiziert die Dynamik des so modellierten Membranpo-

tenzials des Neurons. Die von anderen Neuronen gefeuerte Spikes gelangen in Form eines externen Stroms I_{syn} zum Neuron. I_{syn} kann durch die Faltung aller einkommenden Spikes mit einer Kernel-Funktion modelliert werden. Dieser legt die Entwicklung des postsynaptischen Membranpotenzials (engl. *post-synaptic potential*) fest. Beispiele für oft verwendete Kernel-Funktionen sind die α-Funktion (vgl. Gleichung 2.3), exponentiell abklingende Funktionen und die Delta-Funktion.

Der zuvor beschriebene Ansatz, Spikes zu einem Strom I_{syn} aufzusummieren, wird als Strom-basierter (engl: *current-based*) Ansatz bezeichnet [KAM92]. Zwar führt der Strom-basierte Ansatz zu einem aus mathematischer Sicht einfacherem Problem, vereinfacht jedoch die in der Zelle ablaufenden Prozesse für manche Zwecke zu sehr. Der Leitwert-basierte (engl: *conductance-based*) Ansatz [KAM92] stellt hier eine bessere Alternative dar. Anstatt die durch präsynaptischen Spikes induzierte Ströme einfach aufzusummieren, wird eine Beeinflussung dieser Spikes auf die Leitfähigkeit der Membran modelliert. Zwar werden Leitwert-basierte Modelle als realistischer angesehen, erfordern bei der Lösung der entsprechenden Differenzialgleichungen aber in der Regel ein nummerisches und damit approximatives Verfahren.

2.3 Charakteristische Neuronendynamiken

Dieser Abschnitt stellt unterschiedliche charakteristische Neuronendynamiken vor, die einige wesentliche Eigenschaften der biologischen Neuronen modellieren. Die vorliegende Auswahl der Dynamiken beruht auf deren Verbreitung in der Literatur über neuronale Simulationen.

Im Folgenden wird anhand einer spezifischen Dynamik detailliert erläutert, wie eine exakte inkrementelle Lösung der darin vorkommenden Differenzialgleichung bestimmt werden kann, die eine ressourcensparende nummerische Simulation erlaubt. Diese Lösung ist in Unterabschnitt 8.1.6 als ein symbolisches mathematisches Programm implementiert.

Das Substituieren des Terms $C \cdot R$ in Abbildung 2.3 (B) durch die Membranzeitkonstante τ_{m}, ergibt die kanonische Form der Differenzialgleichung für das *Integrate-and-Fire* Neuronenmodell [DMW89]:

$$\frac{d}{dt}V = -\frac{V}{\tau_m} + \frac{1}{C}I_{\text{syn}} \tag{2.1}$$

Die oben eingeführte Differenzialgleichung beschreibt, wie sich das Membranpotenzial bei eingehenden Strömen, I_{syn}, verhält. Die Form der Ströme ist im Modell nicht beschrieben. In einer realistischen Simulation empfängt das Neuron die Ströme über Synapsen von präsynaptischen Neuronen. Der gesamte Strom $I_{\text{syn}}(t)$ zum Zeitpunkt t, der in ein Neuron fließt, lässt sich dabei als Summe aller eingehenden Ströme berechnen:

$$I_{\text{syn}}(t) = \sum_j w_j \sum_f \text{kernel}(t - t_j^{(f)}) \tag{2.2}$$

$\sum_j$ summiert die Ströme aller eingehenden Synapsen j mit dem Gewicht der jeweiligen Synapse w_j. $\sum_f$ summiert dabei die Ströme einer einzelnen Synapse über alle Aktionspotenziale f. Für jedes Aktionspotenzial f gibt $t_j^{(f)}$ den Zeitpunkt des Aktionspotenzials an, während die Funktion kernel(t) den zeitlichen Verlauf des postsynaptischen Stroms, angestoßen durch ein Aktionspotenzial, darstellt [GKNP14].

Der zeitliche Verlauf der kernel-Funktion wird unterschiedlich modelliert. Beispielsweise ist der α-Kernel durch folgende Gleichung gegeben:

$$\alpha(t) = \alpha_{\text{amp}} \frac{e}{\tau_\alpha} t e^{\frac{t}{\tau_\alpha}} \tag{2.3}$$

α_{amp} bestimmt die maximale Auslenkung. Diese α-Funktion hat eine feste Anstiegszeit, die durch die Zeitkonstante τ_α definiert ist. $\alpha(t)$ hat einen exponentiellen Verfall, wodurch die Dauer des Stromflusses eines Aktionspotenzials begrenzt wird. Beispielhafte Funktionsverläufe mit unterschiedlichen Zeitkonstanten sind in Abbildung 2.4 zu sehen.

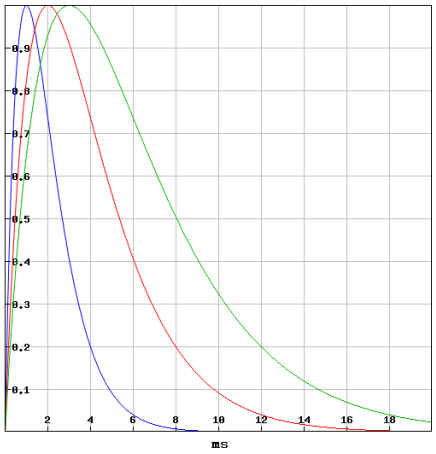


Abbildung 2.4: Unterschiedliche Verläufe der α-Funktion mit Anstiegszeiten $\tau_{syn} = 1ms$ (erste), $\tau_{syn} = 2ms$ (zweite), $\tau_{syn} = 3ms$ (dritte), jeweils für $\alpha_{\text{amp}} = 1$.

Durch das Einsetzen des α-Kernel in das *Integrate-and-Fire*-Modell, das durch Gleichung 2.1 definiert ist, ergibt sich der folgende synaptische Strom zum Zeitpunkt t:

$$\iota(t) = \hat{\iota} \frac{e}{\tau_\alpha} t e^{\frac{-t}{\tau_\alpha}}. \tag{2.4}$$

Im vorliegenden Fall ist $\hat{\iota}$ der Höchstwert des postsynaptischen Potentials und τ_α die Anstiegszeitkonstante des verwendeten α-Kernels. Die inhomogene Differenzialglei-

chung Gleichung 2.4 ($I = \iota$) kann wie folgt als ein homogenes Differenzialgleichungssystem umformuliert werden:

$$\frac{d}{dt}\begin{pmatrix} \frac{d}{dt}\iota + \frac{1}{\tau_\alpha}\iota \\ \iota \\ V \end{pmatrix} = \begin{pmatrix} -\frac{1}{\tau_\alpha} & 0 & 0 \\ 1 & -\frac{1}{\tau_\alpha} & 0 \\ 0 & \frac{1}{C} & -\frac{1}{\tau_m} \end{pmatrix} \cdot \begin{pmatrix} \frac{d}{dt}\iota + \frac{1}{\tau_\alpha}\iota \\ \iota \\ V \end{pmatrix} \tag{2.5}$$

Bei konstantem Zeitschritt h kann dieses homogene Gleichungssystem durch die einmalige Berechnung eines Matrixexponentials [HJ12] der vorliegenden Matrix exakt gelöst werden [RD99]. Diese Lösung berechnet numerisch das beste Ergebnis im Rahmen der Maschinengenauigkeit und sie ist auch algorithmisch effizient, da sie im Vergleich zum einem numerischen (nicht-analytischen) Verfahren weniger Rechenoperationen und stets dieselbe Anzahl von Schritten für die Lösung benötigt.

Mit einem ähnlichen Ansatz können auch synaptische Verbindungen zwischen Neuronen mathematisch modelliert werden. Man kann bereits jetzt erkennen, dass der Prozess der Simulation eines Neuronenmodells vom Modellierer ein tiefes Verständnis der Mathematik, insbesondere der Numerik, erfordert. Daher wird im Laufe dieser Arbeit eine dedizierte DSL entwickelt, mit deren Hilfe Differenzialgleichungen spezifiziert (vgl. Abschnitt 5.4) und anschließend analysiert werden können (vgl. Unterabschnitt 8.1.6).

Eingehende Ströme verändern das Membranpotenzial kontinuierlich in Abhängigkeit zur Membrankapazität C. Dies geschieht so lange, bis das Membranpotenzial einen vordefinierten Schwellenwert V_{th} erreicht. Beim Erreichen von V_{th} wird ein Aktionspotenzial ausgelöst, dessen Form im Modell allerdings nicht beschrieben ist. Auf die Aussendung des Aktionspotenzials folgt schließlich eine Refraktärphase, in der es nicht zu einem erneuten Aktionspotenzial kommen kann.

Im Gegensatz zur deklarativen Notation der Differenzialgleichungen, werden die Logik und Prüfung des Schwellendurchstoßes, die Spikeerzeugung und der Eintritt und das Verlassen der Refraktärphase imperativ spezifiziert (vgl. Abschnitt 5.3).

Das *Integrate-and-Fire* Modell beschreibt nur einige wichtige Eigenschaften der biologischen Neuronen. Ein wesentlicher Bestandteil, der in diesem Modell fehlt, ist die Diffusion der Ionen in der Zelle, die mit der Zeit das Ruhemembranpotenzial wiederherstellt. Im oben beschriebenen Modell wird jeder eingehende Strom aufgenommen und gespeichert, ohne dass dieser Strom das Neuron wieder verlässt. Nachfolgend werden detailliertere Modelle erläutert, die diese und weitere wichtige Eigenschaften ebenfalls mit einbeziehen.

Leaky Integrate-and-Fire: Das *Leaky Integrate-and-Fire* Neuronenmodell [Bur06] ist eine direkte Erweiterung des *Integrate-and-Fire*-Modells, das die Diffusion von Ionen und deren Auswirkung auf das Membranpotenzials mit in Betracht zieht. Dabei wird die Modellgleichung des *Integrate-and-Fire* Modell um einen zusätzlichen Term erweitert,

der die Diffusion beschreibt. Die resultierende Gleichung für dieses Modell lautet wie folgt:

$$\frac{d}{dt}V = -\frac{V}{\tau_m} + \frac{1}{C}(I + I_{\text{leak}}) \tag{2.6}$$

$$I_{\text{leak}} = -\frac{C}{\tau_m}V \tag{2.7}$$

Zusätzlich zur Gleichung aus Abbildung 2.3 (B) enthält das *Leaky Integrate-and-Fire* Modell einen Term I_{leak}, der die Leckströme repräsentiert. Der Strom I_{leak} wird in Abhängigkeit vom Membranpotenzial V definiert.

Adaptive Integrate-and-Fire: Aufgrund eingehender Ströme können in Neuronen Aktionspotenziale ausgelöst werden. Die zuvor erläuterten Neuronenmodelle beschreiben eine mögliche Art, die zu einem Aktionspotenzial führt. Beim Beobachten der Neuronen über einen längeren Zeitraum können allerdings Veränderungen in der Feuerrate der Neuronen festgestellt werden. Die Feuerrate bezeichnet die Häufigkeit, mit der ein Neuron in einem bestimmten Zeitraum Aktionspotenziale aussendet. Viele Neuronen zeigen unter bestimmten Umständen einen Rückgang der Feuerrate mit fortschreitender Zeit [BH03].

Um diese Eigenschaft abzubilden, kann das *Leaky Integrate-and-Fire* Modell um einen Adaptationsterm w erweitert werden [BG05]:

$$\frac{d}{dt}V = -\frac{V}{\tau_m} + \frac{1}{C}(I + I_{\text{leak}} - w) \tag{2.8}$$

w beschreibt hier einen Adaptationsstrom. Der Adaptationsstrom ergibt sich aus der folgenden Differenzialgleichung:

$$\frac{d}{dt}w = \frac{a}{\tau_w}(V - w) \tag{2.9}$$

Die Konstante a ist ein Parameter, der die Abhängigkeit vom Membranpotenzial beinflusst. Gleichzeitig wird w um einen konstanten Wert erhöht, wenn ein Aktionspotenzial aufgetreten ist. Somit ergibt sich eine Adaptation im unterschwelligen Bereich des Membranpotenzials durch den Parameter a und eine Adaptation bei Überschreiten des Schwellenwertes.

Hodgkin-Huxley: Wie alle anderen Körperzellen sind auch Neuronen durch eine Membran von ihrer Umgebung abgegrenzt. Diese Membran ist jedoch nicht komplett undurchlässig. Passive Kanäle (engl: *channels*), die in die Membran eingebettet sind, erlauben es bestimmten Arten von Ionen selektiv die Membran nach außen und nach innen zu passieren. Zusätzlich bewegen aktive Transportermoleküle Ionen in und aus der Zelle. Diese

beiden Mechanismen halten ein Landungsgefälle aufrecht, das als elektrisches Potenzial über die Membran gemessen werden kann.

Das Leitwert-basierte *Hodgkin-Huxley*-Modell [KAM92], gehört zu den grundlegenden Neuronenmodellen [GKNP14]. Seine Modellgleichungen beschreiben die biochemischen Abläufe im Neuron direkt, weshalb dieses Modell im Vergleich zu den zuvor beschriebenen abstrakteren Neuronenmodellen als realistischeres Modell für biologische Neuronen gilt.

Die Kernidee des Modells ist es, die Ionenkanäle in der Zellmembran zu modellieren und aus den resultierenden Strömen das Membranpotenzial zu bestimmen. Dabei werden im einfachsten Fall Kanäle für Natrium- (I_{Na}) und Kalium-Ionen (I_{K}) und ein weiterer Term für Leckströme (I_{L}) verwendet. Somit ergibt sich die folgende Gleichung für das Membranpotenzial:

$$\frac{d}{dt}V = I - I_{\mathrm{K}} - I_{\mathrm{Na}} - I_{\mathrm{L}} \tag{2.10}$$

Auch hier repräsentiert I den synaptischen Strom. Die Gleichungen für die Ionenströme I_{K}, I_{Na} und I_{L} sind wie folgt definiert:

$$I_{\mathrm{K}} = g_{\mathrm{K}} \cdot n^4 \cdot (V - V_{\mathrm{K}}) \tag{2.11}$$
$$I_{\mathrm{Na}} = g_{\mathrm{Na}} \cdot m^3 \cdot h \cdot (V - V_{\mathrm{Na}}) \tag{2.12}$$
$$I_{\mathrm{L}} = g_L \cdot (V - V_{\mathrm{L}}) \tag{2.13}$$

n, m und h sind sogenannte *Gating*variablen, die den Öffnungsgrad der Ionenkanäle jeweils als Wert zwischen 0 und 1 beschreiben. Gatingvariablen und maximale Leitwerte von $g_i, i \in K, Na, L$ bilden die Leitwerte für die Kanäle. Der Leitwert des Leckstroms ist stets konstant. Die Ströme ergeben sich aus den Leitwerten und der Potentialdifferenz zwischen dem Membranpotenzial und den Gleichgewichtspotenzialen des jeweiligen Ionenkanals V_{K}, V_{Na} und V_{L}.

Im Zusammenspiel mit den jeweiligen Ionenkanälen sorgen die Dynamiken der Gatingvariablen dafür, dass das Neuron ein Aktionspotenzial aufbauen und anschließend in eine Refraktärphase übergehen kann. Die biologischen Eigenschaften des Neurons sind damit direkt im Modell enthalten. Die Gatingvariablen werden durch folgende Differenzialgleichungen für jedes $x \in \{n, m, h\}$ wie folgt spezifiziert:

$$\frac{d}{dt}x = \alpha_{\mathrm{x}} \cdot (V) \cdot (1 - x) - \beta_{\mathrm{x}} \cdot V \cdot x \tag{2.14}$$

α_{x} und β_{x} sind vom Membranpotenzial V abhängige exponentielle Funktionen, deren genaue Form in Experimenten bestimmt wurde. Beispielsweise sieht die Definition von α_{h} wie folgt aus:

$$\alpha_{\mathrm{h}}(V) = 0.032 \cdot (15 - V)/(e^{((15-V)/5)} - 1) \tag{2.15}$$

Das einfache *Hodgkin-Huxley*-Modell sieht nur drei Ionenkanäle I_{K}, I_{Na} und I_{L} vor. Der grundsätzliche Formalismus lässt sich allerdings beliebig auf weitere Kanaltypen erweitern. Die Komplexität und Kopplung der Gleichungen führt dazu, dass sich das Modell nur approximativ und mit erhöhten rechnerischen Aufwand lösen lässt. Eine Möglichkeit zur Vereinfachung des *Hodgkin-Huxley*-Modells besteht darin, die vierdimensionale Differenzialgleichung durch ein zweidimensionales Gleichungssystem auszudrücken. Dabei werden die zeitabhängigen Eigenschaften der Gatingvariablen analysiert und ausgenutzt, um sich der vierdimensionalen Gleichung anzunähern. In diesem Sinne können die oben erläuterten *Integrate-and-Fire*-Modelle als Spezialfälle des *Hodgkin-Huxley*-Neurons angesehen werden.

Mehrsegment-Modelle: Neuronenmodelle, die aus mehreren Kompartimenten bestehen (engl: *multi-compartment models*) bilden die gesamte Form des Neurons, speziell seinem verästelten Dendriten, ab [GKNP14]. Dabei wird das Neuron in einzelne Abschnitte eingeteilt, zum Beispiel in Soma, Axon und Dendriten. Die Abschnitte sind miteinander verbunden und können beliebig verzweigt sein. Ihre biochemischen Eigenschaften können mithilfe des *Hodgkin-Huxley*-Neuronenmodells spezifiziert werden. Die Modellierung der Verbindung der Abschnitte untereinander findet auf der Basis der Kabeltheorie [JNT75] statt. Damit ergänzen die Mehrsegment-Modelle den *Hodgkin-Huxley*-Ansatz um die Möglichkeit einer genaueren Spezifikation der Morphologie des Neurons.

2.4 Zusammenfassung

Dieses Kapitel hat die biologischen und mathematischen Grundlagen der neuronalen Modellierung vorgestellt, die für das Verständnis der vorliegenden Arbeit notwendig sind. Mathematische Methoden zur numerischen Simulation von neuronalen Netzwerke wurden an einem Beispiel detailliert erläutert. Desweiteren wurden Ansätze zur zeitlichen Auswertung von neuronalen Netzwerken in Simulationen behandelt. Auf diesen Grundlagen aufbauend werden im kommenden Kapitel bereits existierende Modellierungssprachen vorgestellt und analysiert.

Kapitel 3

Nutzungsszenarien und Anforderungen für eine Neuronen-Modellierungssprache

Dieses Kapitel stellt die Nutzerszenarien und die wesentlichen Rollen vor, die für die Entwicklung von Neuronenmodellen und deren anschließende Simulation relevant sind. Diesen Überlegungen folgend und basierend auf einer Literaturrecherche werden die Anforderungen an Modellierungssprachen für biologisch motivierte Neuronen definiert. Danach werden die wichtigsten existierenden Modellierungsansätze für die Neuronenspezifikation präsentiert und evaluiert. Die Resultate dieser Evaluierung begründen die Entscheidung, eine dedizierte für NEST spezialisierte Sprache zu entwickeln, statte eine der bereits existierenden Sprachen für NEST zu verwenden.

3.1 Nutzungsszenarien mit ihren Rollen

Dieser Abschnitt stellt wichtige Rollen vor, die bei der Entwicklung von Experimenten in der Computational Neuroscience, von Neuronenmodellen bzw. der darunter liegenden Infrastruktur relevant sind. Grundlegend für eine Differenzierung sind hierbei unterschiedliche Sichtweisen auf die Entwicklung von Softwaremodulen und Experimenten. Einerseits werden Neuronenmodelle entwickelt und benutzt, um konkrete Simulationen durchzuführen, andererseits können die verwendeten Modellierungssprache und deren Verarbeitungstools an sich weiterentwickelt werden. Schließlich werden alle entwickelten Module benutzt, um die eigentliche Simulation durchzuführen.

Abbildung 3.1 illustriert dies anhand einer abstrakten Instanz der Simulationsumgebung bestehend aus einer Modellierungssprache, einem Simulator und einem Generator, der die Neuronenmodellbeschreibung in eine vom Simulator ausführbare [Rum02] Form überführt. Dabei sind die folgenden Komponenten relevant [KRV06]:

- Eine **Grammatik** ist ein formales Modell, das die Struktur der Modellierungssprache definiert. Des Weiteren dient die Grammatik der Definition des Metamodells der Modellierungssprache, das die maschinelle Verarbeitung der Neuronenmodelle erlaubt.

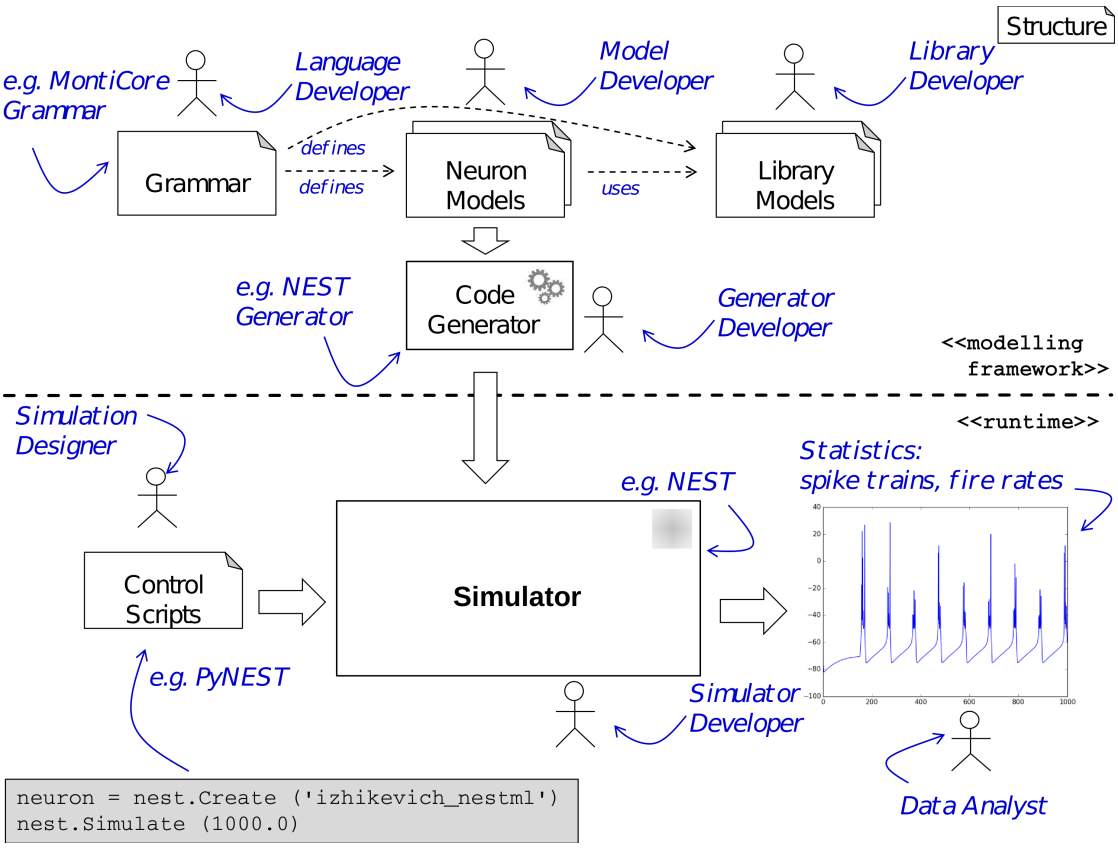


Abbildung 3.1: Ablauf bei der Entwicklung eines Modells und dessen Simulation einschließlich der entsprechenden Rollen.

- **Neuronenmodelle** sind Instanzen der Modellierungssprache. Diese Modelle liegen in Form von Dateien vor. Neuronenmodelle sind konform zur Grammatik.
- Eine **Modellbibliothek** besteht aus einer Sammlung der Neuronenmodellen, die komplett oder in Teilen wiederverwendet werden können.
- Ein **Codegenerator** ist ein Programm, das aus einer Menge von Input-Artefakten eine Menge von Output-Artefakten erzeugt (”generiert”). In diesem Fall produziert der Generator eine für einen Simulator ausführbare Repräsentation aus Input-Dateien. Typischerweise wird für jeden Simulator ein spezieller Generator entwickelt.
- Ein **Simulator** ist ein Programm, das die Neuronenmodelle ausführen kann. Zur Ausführung gehören sowohl die Simulation des einzelnen Neurons als auch eines Verbundes von Neuronen.

- **Steuerungsskripte** beschreiben ein bestimmtes wissenschaftliches Experiment für einen bestimmten Simulator.
- **Simulationsstatistiken** sind Ergebnisdaten, die während der Simulation erzeugt werden und die Grundlage für die Auswertung der Simulation darstellen. Zu solchen Statistiken gehören Daten wie Membranpotenzialverläufe und Spike-Trains.

An diesem Prozess sind die folgenden Rollen beteiligt:

- Der **Sprachentwickler** kennt die innere Struktur der Sprache und der entsprechenden Werkzeuge. Er realisiert neue Funktionen in dessen Kern. Dazu gehören die Definition und Änderungen der internen Struktur der Sprache bzw. der syntaktischen Form der Modelle dieser Sprache.
- Der **Modellierer** ist der eigentliche Benutzer der Modellierungssprache. Im Allgemeinen ist er kein Computerwissenschaftler oder Mathematiker und entwickelt Neuronenmodelle mit dem Ziel diese anschließend in einer Simulationsumgebung zu evaluieren.
- Der **Bibliothekentwickler** definiert wiederverwendbare qualitätsgesicherte Modelle bzw. Modellteile. Wie in der Softwareentwicklung existieren auch bei Neuronenmodellen wiederkehrende Aspekte der Modellspezifikation, die in einer Bibliothek zur allgemeinen Verfügung gestellt werden können.
- Der **Generatorentwickler** ist dafür zuständig, die in der Modellierungssprache spezifizierten Modelle, in eine ausführbare Form zu bringen. Es können mehrere Generatoren existieren, um die Modelle derselben Sprache für unterschiedliche Plattformen zu übersetzen.
- Der **Simulatorentwickler** beschäftigt sich mit der Entwicklung und Optimierung der Simulationsumgebung für neuronale Netzwerke, deren Bestandteil die Neuronenmodelle sind. Da an solchen Simulationen eine Vielzahl von Neuronen und Synapsen beteiligt sein können, werden die Modelle typischerweise in einer Systemprogrammiersprache wie C oder C++ implementiert, um möglichst optimal die verfügbaren Hardwareressourcen auszunutzen.
- Der **Simulationsentwickler** entwirft Simulationen auf Basis von Neuronenmodellen, um die Funktionsweise von realen biologischen Neuronen nachzuahmen. Dabei werden neuronale Aktivitäten oft in einem großen Netzwerk untersucht.
- Während einer Simulation fallen typischerweise sehr große Datenmengen an. Der **Datenanalyst** versucht aus den gesammelten Simulationsdaten wissenschaftliche Erkenntnisse zu gewinnen. Dabei benötigt er Kenntnisse über mathematische Eigenschaften der zugrundeliegenden Neuronenmodelle, sowie über die Biologie des simulierten Systems.

Im Weiteren werden Nutzungsszenarien auf Basis der eingeführten Rollen definiert, die von einer gewünschten Modellierungssprache und deren Werkzeugen unterstützt werden sollen. Die Szenarien beschreiben einen exemplarischen Vorgang und stellen die Benutzung des Systems durch eine Sequenz von Nutzeraktionen dar [Gli00]. Die Beschreibung der Szenarien erfolgt in natürlicher Sprache. Zur Strukturierung der Szenarien wird dabei die Interaktion der Nutzer mit der Modellierungssprache und Werkzeuge explizit hervorgehoben.

(S1) Modellbildung: In diesem Szenario wird ein neues Neuronenmodell erstellt. Dabei wird ein in der Literatur beschriebenes Modell bzw. ein eigens entwickelter Ansatz simuliert. Im allgemeinen Fall wird das Modell komplett vom Grund auf neu erstellt. Es findet keine Wiederverwendung statt. Dazu erstellt der *Modellierer* zuerst die mathematische Beschreibung des Modells in Form der Differenzialgleichungen. Der Modellierer dokumentiert dieses Modell mit Kommentaren und benutzt einen *Codegenerator*, um das Modell in den ausgesuchten *Simulator* zu integrieren.

- **(S1.1) Vererbung:** In diesem Szenario wählt der *Modellierer* ein bereits existierendes Modell aus und erweitert es um neue Funktionalität. Dabei wird das zu erweiternde Modell im Erweiterungsmodell lediglich referenziert. In diesem Erweiterungsszenario können sowohl Daten als auch das Verhalten des zu erweiternden Modells verändert werden. Beim Datenzustand können neue Variablen hinzukommen. Bei der Verhaltensänderung können sowohl das prozedurale Verhalten der Dynamik als auch die deklarativen Beschreibungen der Variablen angepasst werden.
- **(S1.2) Komposition:** Auch wenn das Modell neu erstellt wird, können manche wiederkehrende Programmabschnitte bereits in einer modularen Komponente zur Verfügung stehen. Bei diesem Szenario sucht der *Modellierer* ein passendes Modul in der Modellbibliothek aus und integriert dieses in das neue Neuronenmodell durch eine Kompositionsbeziehung. Dafür importiert er das Modul und benutzt es anhand der öffentlichen Schnittstelle. Der *Bibliothekentwickler* stellt eine qualitätsgesicherte Sammlung solcher Module zur Verfügung.

(S2) Spracherweiterung: In diesem Szenario wird die Modellierungssprache an sich verändert. Dabei wird die Sprache an neue Anforderungen angepasst. Der *Sprachentwickler* integriert ein neues Sprachonstrukt in die Modellierungssprache. Dafür verändert er direkt die Grammatik, die die Sprache definiert. Alternativ kann er die Spracherweiterungsmechanismen [HLMSN$^+$15, KKP$^+$09, KRV07, KRV08] benutzen, um die Änderungen in einer neuen Grammatik zu manifestieren.

(S3) Anpassung des Generierungsframeworks: In den folgenden Szenarien wird der Codegenerator entweder erweitert oder komplett neu entwickelt.

(S3.1) Neuer Generator: Der *Generatorentwickler* erstellt einen Generator, um eine neue Zielplattform zu bedienen.

(S3.2) Handgeschriebener Code: In bestimmten Fällen kann der generierte Code durch optimierte handgeschriebene Codeabschnitte ergänzt bzw. ersetzt werden. Der *Generatorentwickler* benutzt einen vordefinierten Mechanismus für die Integration des handgeschriebenen Codes.

(S4) Erweiterung des Simulators: In diesem Szenario wird während der Weiterentwicklung des Simulators durch den *Simulatorentwickler* die Schnittstelle der Neuronenmodelle verändert. Anstatt alle Modelle an die neue Schnittstelle manuell anzupassen, wird lediglich der entsprechende Codegenerator durch den *Generatorentwickler* entsprechend aktualisiert. Alle Neuronenmodelle bleiben dabei unverändert.

(S5) Durchführung einer Simulation: Der *Simulationsentwickler* erstellt eine Simulation für ein neurowissenschaftliches Experiment. Dabei ist für ihn die Dokumentation der Neuronenmodelle wichtig. Neben der statischen Schnittstelle des Modells (d.h. die Menge der möglichen einstellbaren Parameter) sind auch mathematische Eigenschaften des jeweiligen Modells von Bedeutung. Als erstes will der *Simulationsentwickler* sich einen Überblick über die statische Schnittstelle des Modells verschaffen bzw. über die Standardwerte der Variablen. In Kapitel 2 wurde bereits erläutert, dass Spike-Trains eine wichtige Rolle bei der Beurteilung des Simulationsergebnisses spielen. Um ein Modell mit einem bestimmten Spiking-Verhalten auszusuchen, will der *Simulationsentwickler* einen Überblick über Modellgleichungen haben.

(S6) Analyse der Simulation: Der *Datenanalyst* untersucht eine durchgeführte Simulation und versucht, daraus wissenschaftliche Erkenntnisse zu gewinnen. Um die Befunde besser zu verstehen, sind für ihn vor allem die mathematischen Eigenschaften der zugrunde liegenden Neuronenmodelle wichtig.

3.2 Anforderungen an die Neuronenbeschreibungssprache

Um bereits existierende Modellierungssprachen aus der Computational Neuroscience zu evaluieren, stellt dieser Abschnitt einen Anforderungskatalog vor. Dieser Katalog basiert einerseits auf den Forschungsfragen *Q1, Q2, Q3, Q4* und *Q5* aus Abschnitt 1.3 bzw. Nutzungsszenarien aus Abschnitt 3.1. Andererseits werden die in den Arbeiten [GHH+01, KKP+09] vorgestellten Anforderungen aufgearbeitet. Dabei werden die für die vorliegende Arbeit am meisten relevanten Anforderungen identifiziert und auf die neurowissenschaftliche Domäne angepasst und erweitert.

(RQ1) *Modellierungsstil:* Im Unterschied zu einer General Purpose Language (General Purpose Lanuage (GPL)) dienen Domain Specific Languages (DSLs) einem bestimmten wohldefinierten Zweck, der auf die Anwendungsdomäne eingegrenzt ist. Im Falle der vorliegenden Arbeit ist dieser Zweck die Spezifikation von Neuronen. Dadurch wird es erst möglich, die Sprache auf Definition von Neuronenmodellen zu spezialisieren. Um die Vorteile dieser Spezialisierung voll auszunutzen, sollte der Entwurf der Modellierungssprache die folgenden Anforderungen berücksichtigen:

- **(RQ1.1)** *Klarheit:* Da der Prozess der Entwicklung neuer Neuronenmodelle bzw. das Nachvollziehen der bereits existierenden Modelle eine komplexe Unternehmung ist, sollte die Komplexität der Sprache diesen Prozesse nicht unnötig erschweren. Der Kern des Neuronenmodells soll für den Modellierer direkt klar und ersichtlich sein. Wortreiche Definitionen sollen vermieden werden.
- **(RQ1.2)** *Portabilität:* Jede Simulationsumgebung verwendet eine eigene Modellspezifikation. Daher sollte die Modellierungssprache Simulator-agnostisch bzw. zumindest portierbar definiert werden, um das Teilen der Modelle zwischen verschiedenen Simulatoren zu ermöglichen. Um dies zu gewährleisten, dürfen keine simulatorspezifischen Modellierungskonzepte benutzt werden. Bei Bedarf können Neuoronmodelle erst zur Generierungszeit um diese kontextabhängigen Aspekte ergänzt werden. Dies kann durch den Codegenerator oder Markierungen [GLRR15] erfolgen, die in separaten Dateien gespeichert sind.
- **(RQ1.3)** *Modularität:* Da die Neuronenmodelle in unterschiedlichen Kontexten eingesetzt werden, müssen diese modular spezifiziert sein. Die Modularität bildet die Grundlage für die Wiederverwendung der Neuronenmodelle [BR07] (vgl. **(RQ3)**).
- **(RQ1.4)** *Kompaktheit:* Die Modellnotation soll kompakt sein, da die Kompaktheit sowohl das Verstehen als auch das Verfassen von Modellen positiv beeinflusst.

(RQ2) *Konkrete Syntax:* Dem Entwickler einer DSL stehen viele Möglichkeiten zur syntaktischen Realisierung einer domänenspezifischen Sprache zur Verfügung. Dennoch birgt dies auch die Gefahr, dass die DSL mit diversen syntaktischen Elementen überladen wird. Dann würde die DSL deren Vorteile im Vergleich zu einer GPL verlieren. Die konkrete Syntax der DSL soll so gewählt werden, dass die Syntax leicht verständlich und erlernbar ist. Beispielsweise sollten Sprachkonstrukte, die Neuronen oder Kanäle modellieren, als Konstrukte erster Klasse in der Sprache zur Verfügung stehen. Um die Vorteile der DSL im Vergleich zu einer GPL zu stärken, sind die folgenden Anforderungen an die konkrete Syntax der Sprache zu berücksichtigen:

- **(RQ2.1)** *Konsistenz:* Da die DSLs für die Lösung eines bestimmten Problems erstellt werden, soll jedes Konzept der DSL entweder diesem Zweck dienen

oder komplett ausgelassen werden [POB00]. Beispielsweise sollte die Sprache von der technischen Schicht abstrahieren und die technischen Details eines spezifischen Simulators in der konkreten Syntax nicht widerspiegeln.

(RQ2.2) *Minimalität:* Nur die nötigen Domänenkonzepte werden in der DSL widergespiegelt. Für jedes Konzept soll die DSL auch genau eine Alternative zur Verfügung stellen.

(RQ2.3) *Simplizität:* Die Simplizität einer Sprache trägt dazu bei, die Verständlichkeit und Erlernbarkeit dieser Sprache zu erhöhen [Hoa73, Wir74, POB00]. Diese Anforderung ist auch wichtig, weil eine einfache Sprache typischerweise die Entwicklung der Sprachwerkzeuge vereinfacht. Des Weiteren senkt eine einfache Sprache die Barriere, die Sprache bei potenziellen Anwendern einzuführen. Die folgenden Anforderungen **RQ2.4, RQ2.5, RQ2.6** helfen dabei, dieses Ziel zu erreichen.

(RQ2.4) *Vermeidung unnötiger Generalisierung:* Nur die für die Lösung des Problems der Neuronenspezifikation nötigen Konzepte sollen unterstützt werden. Die vorzeitige Verallgemeinerung bzw. Parametrisierung der Modelle sollte vermieden werden, da dies in Widerspruch zu **(RQ2.3)** stünde. Beispielsweise sollte eine Modellierungssprache für Neuronen ein eingebautes Konstrukt für die Definition der Neuronen besitzen und dies nicht durch eine Instanzierung einer abstrakten Klasse modellieren.

(RQ2.5) *Begrenzung der Sprachelemente:* Eine Modellierungssprache mit hunderten von Sprachelementen wäre sehr schwer zu verstehen. Um die Verständlichkeit und Erlernbarkeit der Modellierungssprache zu verbessern, soll die Anzahl der zu erlernenden Elemente begrenzt werden. Dies kann dadurch erreicht werden, dass die DSL in Subsprachen aufgeteilt wird, die dedizierte Aspekte der Neuronenmodellierung abdecken. Subsprachen würden es erlauben, im Nachhinein den Umfang der Modellierungssprache zu erweitern. Eine andere Möglichkeit wäre es, Konzepte in eine Modellbibliothek auszulagern.

(RQ2.6) *Bekannte Notationen:* Neurowissenschaftler sind klassischerweise keine Computerwissenschaftler. Es ist schwierig für sie, neue an Programmiersprachen angelehnte Notationen zu erlernen. Deswegen sollte die bereits bekannten und in der Domäne zur Anwendung kommenden Notationen möglichst wiederverwendet werden [Wil03]. Im Falle der Neurowissenschaften genießt insbesondere die Programmiersprache Python besonders große Verbreitung. Daher sollte die konkrete Syntax der Modellierungssprache der Python-Syntax ähneln. Auch eine mathematische Notation von Differenzialgleichungen und physikalische Einheiten müssen fester Bestandteil der DSL sein.

(RQ2.7) *Deskriptive Notationen:* Eine deskriptive Notation unterstützt die Erlernbarkeit der Sprache und vereinfacht das Nachvollziehen der Modelle, wenn

die in der Domäne bekannten Terme wiederverwendet werden. Um potenzielle Missverständnisse zu vermeiden, muss die ursprüngliche Semantik dieser Elemente erhalten bleiben. Beispielsweise müssen Neuronen in der konkreten Syntax der Sprache den Neuronen aus der neurowissenschaftlichen Domäne entsprechen.

(RQ2.8) *Unterscheidbare Notationen:* Eine leicht unterscheidbare Repräsentation der Sprachelemente ist eine wichtige Voraussetzung für die Verständlichkeit der DSL. In der konkreten Syntax der Modelle dienen die Schlüsselwörter dazu, unterschiedliche Konzepte voneinander zu trennen. Schlüsselwörter sollen an passenden Stellen in der konkreten Syntax des Modells benutzt werden. Daher muss die Struktur des Neurons in passende Blöcke aufgeteilt werden. Die Blöcke müssen leicht voneinander unterscheidbar sein und mit einleuchtenden Schlüsselwörtern eingeleitet werden.

(RQ3) *Wiederverwendung:* Die modulare Entwicklung ist ein wichtiger Beitrag zur Reduktion der Komplexität eines Systems. Durch die Definition von Komponenten, die eine explizite Schnittstelle zur Kommunikation anbieten, bleiben Details der internen Implementierung nach dem Prinzip des „information hidings“ verborgen [Par72]. Die Komponenten werden nur mithilfe der vorgesehenen Schnittstelle benutzt. Das erlaubt Änderungen an der internen Implementierung einer Komponente durchzuführen, ohne dass die benutzenden Komponenten angepasst werden müssen, wodurch eine Skalierung des Entwicklungsprozesses und die Verwendung heterogener Modelle erlaubt wird [HKR+09]. Auch ein transparenter Austausch der Komponenten, durch andere Komponenten mit kompatibler Schnittstelle, ist möglich [RSW+15, BMP+16]. Insbesondere für komplexe Modelle ist eine Aufteilung in kleine modulare Komponenten unabdingbar.

(RQ3.1) *Komponentenkonzept:* Die Modellierungssprache muss eine explizite Notation für die Definition eines Moduls bzw. einer Komponente zur Verfügung stellen. Komponenten sollen eine wohldefinierte Schnittstelle anbieten. Anhand dieser Schnittstelle findet die Kommunikation zwischen verschiedenen Neuronen statt.

(RQ3.2) *Anpassung durch Vererbung:* Das Konzept der Vererbung [Sch99] in Programmiersprachen ist eine bewährte Technik für die Wiederverwendung. Daher soll die Neuronenmodellierungssprache erlauben, Neuronenmodelle durch Überschreibung eines Teilaspekts in einem Erweiterungsmodell anzupassen.

(RQ4) *Metamodell:* Das Metamodell einer DSL stellt die strukturelle Essenz dieser Sprache dar [Küh06]. Es lässt sich auf verschiedene Arten beschreiben. In den grammatikbasierten Formalismen entspricht das Metamodell den Klassen des Abstract Syntax Trees (Abstract Syntax Tree (AST)). Da die Sprachen im Laufe der

Zeit weiterentwickelt werden bzw. an neue Anforderungen angepasst werden, ist eine geeignete Wahl des erweiterbaren Metamodells wichtig.

(RQ4.1) *Aufbau des Metamodells:* Der grammatikalische Aufbau der DSL und des entsprechenden Metamodells müssen flexibel, modular und erweiterbar sein (um **RQ2.5** zu ermöglichen). Es muss möglich sein, die Grammatikdefinition und die entsprechende Sprachinfrastruktur an neue Anforderungen anzupassen. Um die Implementierung der Sprachverarbeitungswerzeuge zu vereinfachen, sollten konkrete Syntax und das Metamodell aufeinander abgestimmt sein.

(RQ4.2) *Sprachverarbeitungsinfrastuktur:* Die Werkzeuge zum Parsen, Traversieren des Metamodells und Prüfung der Kontextbedingungen sollen für Sprachentwickler zur Verfügung stehen. Diese Werkzeuge sollen modular, kombinierbar und erweiterbar aufgebaut sein.

(RQ5) *Codegenerierung:* Das Modellierungsframework muss eine gute Unterstützung für die Codegenerierung anbieten.

(RQ5.1) *Mehrere Zielplattformen:* Es muss möglich sein, aus denselben Neuronenmodellen ausführbare Implementierung für unterschiedliche Zielplattformen zu generieren. Das Modellierungsframework soll eine unterstützende Infrastruktur anbieten, um Codegeneratoren komfortabel zu implementieren.

(RQ5.2) *Handgeschriebene Erweiterungen:* Für leistungskritische Implementierungsteile soll es möglich sein, eine handgeschriebene Implementierung in den generierten Code zu integrieren. Dabei muss der handgeschriebene Code nicht Teil des Neuronenmodells sein.

(RQ5.3) *Dokumentation:* Die Dokumentation bzw. Modellkommentare müssen im generierten Code widergespiegelt sein.

(RQ5.4) *Differenzialgleichungen:* Aus deklarativen Modellbeschreibungselementen soll eine lauffähige Implementierung generiert werden, die eine effiziente Lösung dieser Differenzialgleichungen unterstützt.

(RQ6) *Benutzerfreundlichkeit:* Die Modellierungssprache und deren Werkzeuge sollten einfach zu benutzen sein.

(RQ6.1) *Self-contained:* Die Modellierungssprache und deren Werkzeuge sollten alle Module, die für die Ausführung notwendig sind, beinhalten und keine aufwendige vorinstallierte Infrastruktur erfordern.

(RQ6.2) *API:* Es muss möglich sein, die Neuronenmodelle auch programmatisch zu erstellen. Dafür muss eine entsprechende API zur Verfügung stehen. Sprachwerkzeuge sollten sich über das `Command Line Interface` bedienen lassen.

(RQ6.3) *Tutorial:* Um den Modellierungsansatz Anwendern vorzustellen, bedarf es einer passenden Anleitung, die den Anwender durch die Entwicklung von Neuronen leitet. Dies schließt unter anderem die folgenden Punkte mit ein:

- Installation der Sprachverarbeitungs- und Analyseinfrastruktur.
- Modellierung eines illustrativen Beispiels.
- Nutzung von Neuronen in einem Simulator.

3.3 Verwandte Arbeiten

Auf dem Gebiet der Computational Neuroscience [CKS93] existieren bereits diverse Modellierungssprachen für die Spezifikation von Neuronenmodellen. Zu den am weitesten verbreiteten gehören NineML [GRHLF11, RCC+11, Dav15], NeuroML [GCC+10] und LEMS [CGC+14]. Diese Ansätze haben jedoch zwei entscheidende Nachteile. Zum einen sind diese Sprachen überwiegend XML basiert, was die Verständlichkeit, Pflege und Weiterentwicklung der so beschriebenen Neuronenmodelle erheblich erschwert. Zum anderen sind diese Ansätze rein deskriptiv und enthalten deshalb keine Werkzeugunterstützung für die Lösung oder Analyse der in Neuronenmodellen vorkommenden Differenzialgleichungen. Desweiteren verfügen Simulatoren wie Topographica [Bed15] und Brian [SGB15] über eingebettete Modellierungssprachen. Daher werden hier all diese Modellierungsansätze detailliert beschrieben und mithilfe der aufgestellten Anforderungen (vgl. Abschnitt 3.2) analysiert.

3.3.1 NineML

NineML [GRHLF11, RCC+11, Dav15] stellt einen Ansatz dar, biologische Neuronenmodelle und neuronale Netzwerkmodelle eindeutig zu beschreiben, um Ergebnisse zwischen unterschiedlichen Forschungseinrichtungen und Simulatoren zu teilen und wiederzuverwenden. Dafür definiert NineML ein generisches Objektmodel, um unterschiedliche Elemente eines neuronalen Netzwerks einheitlich zu modellieren. Im Kontext dieser Arbeit entspricht das Objektmodell dem Metamodell einer grammatikbasierten DSL [Küh06].

NineML ist in zwei semantische Schichten unterteilt: eine abstrakte Schicht (engl: abstract-layer) und eine Benutzer-Schicht (engl: user-layer). Die abstrakte Schicht beschreibt wesentliche Elemente der Neuronen- und Netzwerkmodelle zusammen mit den entsprechenden mathematischen Definitionen, Parametern, Zustandsvariablen und Regeln für die Aktualisierung der Zustandsvariablen (vgl. Abschnitt 2.2). Die Benutzer-Schicht erlaubt es, Elemente aus der abstrakten Schicht zu instanziieren und mit konkreten Werten zu parametrisieren. Unter anderem definiert diese Schicht Anfangs- und Standardwerte der jeweiligen abstrakten Elemente.

Im Weiteren werden die wesentlichen Elemente der abstrakten Schichten detaillierter beschrieben und an einem Beispiel erklärt. In der abstrakten Schicht werden Neuronen

und Ionenkanäle jeweils durch eine `ComponentClass` repräsentiert. Die `ComponentClass` besteht aus einem `Dynamics`-Block und aus einer Menge von `Interface`-Elementen. Der `Dynamics`-Block definiert das Laufzeitverhalten des Neurons bzw. Ionenkanals. Beispielsweise kann die Veränderung der Zustandsvariablen über die Zeit mithilfe von Differenzialgleichungen beschrieben werden. Ein `Interface` kann unterschiedliche `Parameter` und `Port`s beinhalten. Die `Parameter` können von der Benutzer-Schicht aus während der Instanziierung verändert werden. Beispiele für Parameter sind Werte für das initiale Membranpotenzial oder die Refraktärzeit des Neurons. `Port`s dienen dem Zweck, Nachrichten mit anderen Elementen des Netzwerkes auszutauschen. Beispielsweise werden über solche `Port`s Spikes an weitere Elemente des Netzwerkes gesendet. Ein Vorteil dieser Art der Modellierung aus der Sicht des Metamodells ist es, dass alle Elemente eines neuronalen Netzwerkes wie Neuronen, Synapsen oder Ionenkanäle gleichartig dargestellt werden können. Das hat wiederum einen signifikanten Nachteil für die Verständlichkeit solcher Modelle: Das Vorkommen eines `ComponentClass`-Elements im Modelltext erlaubt keine Rückschlüsse auf die Domänenkonzepte, die durch dieses Vorkommen modelliert werden. Es könnte sich um ein Neuron, eine Synapse oder einen Ionenkanal handeln, was nur durch weitere Analyse des vorliegenden Modells geklärt werden kann.

```
1  <ComponentClass name="izhikevich_cell">          Stored in a izhikevich.9ml      9ML
2    <Parameter name="a" dimension="none"/>                                      «abstract»
3    ...                                  a, c, theta are further parameters of the neuron
4    <AnalogPort name="iSyn" mode="reduce" reduce_op="+" dimension="current"/>
5    ...
6    <EventPort name="spikeOutput" mode="send"/>
7    ...
8    <Dynamics>
9      <StateVariable name="V" dimension="voltage"/>
10     <StateVariable name="U" dimension="voltage_per_time"/>
11     <Regime name="subthresholdRegime">
12       <TimeDerivative variable="U">
13       <MathInline>a*(b*V - U)</MathInline>
14       </TimeDerivative>
15       ...
16       <OnCondition target_regime="subthresholdRegime">
17         <Trigger>
18           <MathInline>V \> theta </MathInline>
19         </Trigger>
20         <StateAssignment variable="V" >
21           <MathInline>c</MathInline>          corresponds to: V = c
22         </StateAssignment>
23       ...
24       </OnCondition>
25     </Regime>
26   </Dynamics>
27 </ComponentClass>

2 state automaton
```

Abbildung 3.2: Ein Izhikevich Neurons [Izh03, Izh04] modelliert als NineML-Modell.

Der Modellausschnitt in Abbildung 3.2 demonstriert die erläuterten Konzepte an einem Beispiel. Die Definition des Neurons beginnt mit der `ComponentClass`-Deklaration.

Dabei wird der Name `izhikevich_cell` des Neurons festgelegt, der in der Benutzer-Schicht referenziert werden darf. In den Zeilen 2-8 werden Parameter, Input- und Output-Ports deklariert. Zusätzlich werden in den Zeilen 9-10 zwei Zustandsvariablen U und V deklariert. Schließlich wird in den Zeilen 11-26 der endliche Zustandsautomat [Bra84, HUM02, Rum96] definiert, der das Laufzeitverhalten des Neurons spezifiziert. Der Automat besteht aus einem Zustand mit einer Transition, die von diesem Zustand ausgeht und im selben Zustand endet. Der Automat spiegelt die Logik des Aktualisierungsschrittes des Izhikevich-Neurons wider. U wird stets anhand der Differenzialgleichung $U' = a * (b * V - U)$ verändert. Wenn das Membranpotenzial V den vordefiniert Schwellenwert *theta* überschreitet, wird die Transition ausgeführt. Während der Ausführung der Transition wird die Variable V auf den Wert c gesetzt (vgl. Zeilen 20-22).

Das `Component`-Objekt ist der Hauptbaustein der Benutzer-Schich. Ein `Component`-Objekt ist eine konkrete Instanz eines `ComponentClass`-Elements der abstrakten Schicht. Ein `Component`-Objekt spezifiziert die `Parameter` und ausgehenden `Port`s der dazugehörigen `ComponentClass`. `Parameter` werden dazu konkrete Werte zugeordnet. Bei ausgehenden `Port`s wird spezifiziert, welche Werte gesendet werden. Es ist möglich unterschiedliche Instanzen einer `ComponentClass` in der Benutzer-Schicht zu erstellen.

9ML «user»

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <nineml xmlns="http://www.NineML.org/9ML/1.0" ... >
3    <import language="NineML">
4      http://www.NineML.org/1.0/dimensions.9ml
5    </import>
6    <component name="izhikevich_neuron">
7      <definition url="izhikevich.9ml">
8        izhikevic_cell
9      </definition>
10     <property name="c">
11       <quantity>
12         <value>
13           <scalar> -65 </scalar>
14           <unit> none </unit>
15         </value>
16       </quantity>
17       <note><String> Paper:Izhikevich03 at el. </String></note>
18     </property>
19     ...
20 </nineml>
```

Component which is defined in the abstract layer

c is defined as a parameter in the abstract layer

Abbildung 3.3: Instanziierung des Izhikevich-Neurons (vgl. Abbildung 3.2). Der Parameter c wird dabei auf den Wert -65 gesetzt.

Abbildung 3.3 demonstriert die Instanziierung der zuvor eingeführten `ComponentClass`. In den Zeilen 3-5 werden vordefinierte physikalische Einheiten importiert, damit diese nicht erneut im Neuronenmodell definiert werden müssen. Die Zeilen 7-8 referenzieren das `izhikevich_cell`-Neuron und spezifizieren, dass das vorliegende Neuron eine Instanz des Neurons aus Abbildung 3.2 ist. Anschließend findet in den Zeilen 10-18 eine Zuweisung eines skalaren Wertes zum Parameter c des Izhikevich-Modells statt. Zeile 17

definiert eine Metainformation, die eine Literaturangabe für die vorliegende Parametrierung des Neurons angibt.

Modelle in NineML sind generell Simulator-agnostisch konzipiert. Auf der einen Seite hat das den Vorteil, dass die Neuronenmodelle in unterschiedlichen Simulationsumgebungen ausgeführt werden können. Auf der anderen Seite überlässt dieser Ansatz die Wahl des richtigen Lösers und deren Integration mit der Modellspezifikation dem Anwender der Sprache. Das setzt das Wissen des Anwenders sowohl über die mathematischen Hintergründe des Modells als auch über die Implementierungsdetails des jeweiligen Simulators voraus. Die Modellierung der Dynamik von Neuronen und Synapsen ausschließlich durch einen deterministischen Automaten ist nicht optimal, da die Codierung des Automaten in Form des XML Textes mühsam und fehleranfällig ist.

RQ1 Modellierung: NineML erlaubt plattformunabhängige und somit portable Neuronenmodelle zu erstellen. NineML bietet durch die Trennung von Modellen und Modellinstanzen ein Konzept, welches ein gewisses Maß an Modularität ermöglicht. Der generische Ansatz alle Modellierungselemente im Metamodell homogen zu behandeln ist als Nachteil zu bewerten. Da das XSD-basierte Metamodell sich unmittelbar auf die Syntax der Sprache auswirkt, beeinflusst dies sowohl die Klarheit als auch die Kompaktheit der NineML-Modelle eher negativ.

RQ2 Konkrete Syntax: In NineML ist die Anzahl der Sprachelemente stark beschränkt; grundsätzlich unterscheidet man zwischen `ComponentClass`- und `Component`-Elementen. Dies kann man zunächst als positiv auffassen, da damit die Sprache hinsichtlich der Elemente sehr kompakt bleibt. Mit Hinblick auf Benutzerfreundlichkeit und Unterscheidbarkeit von Sprachelementen ist dies jedoch als negativ zu werten. So kann ein `ComponentClass`-Element einen einzelnen Ionenkanal darstellen, es kann aber auch ein ganzes Neuron beschreiben. Ohne weitere Kommentare bzw. Inspektion des Modells wäre der Unterschied zunächst nicht direkt erkennbar. Somit sind die Modellierungselemente unnötig allgemeingültig konzipiert. Die XML-Repräsentation der Modelle ist schwer nachvollziehbar und schwer zu erstellen. Beide Aktionen erfordern die Erstellung neuer Werkzeuge für die benutzerfreundliche Darstellung und Bearbeitung der Modelle. Dies steht dem Vorteil der Wiederverwendung von existierenden Werkzeugen für XML-Verarbeitung [RCC+11] entgegen. Die Elemente der konkreten Syntax sind nur bedingt an die Domänenbegriffe der Neurowissenschaften angelehnt, was die Akzeptanz und Erlernbarkeit der Sprache erschwert. Die verpflichtende Modellierung der Neuronendynamik mithilfe eines Mealy-Automaten erschwert die Modellbildung der Neuronen für codeaffine Modellierer. Insbesondere gab es zum Zeitpunkt der Erstellung dieser Arbeit keine unterstützenden Werkzeuge für die grafische Modellierung solcher Automaten für NineML. Der Automat musste von Hand im XML-Format codiert werden.

RQ3 Wiederverwendung: Die Wiederverwendung findet auf zwei Ebenen statt. Zum einen können die `ComponentClass`-Elemente während der Instanziierung parametrisiert werden. Zum anderen können die `ComponentClass`-Elemente selber in einer Vererbungsbeziehung stehen. Zusammenhängende Funktionalität kann in Form eines Moduls wiederverwendet werden. Dabei arbeitet NineML auf der Abstraktion von XML und nicht auf der Modellebene, was negativ zu werten ist.

RQ4 Metamodell: Das Metamodell von NineML ist durch das zugrundeliegende XSD-Schema definiert. Somit bietet es nur eingeschränkt eine Unterstützung für eine Spracherweiterung. Das Metamodell ist nicht modular aufgebaut. Der aus dem Modell resultierende AST ist nicht stark typisiert. Dies beeinträchtigt die Entwicklung der Sprachverarbeitungswerkzeuge. Somit können mögliche Typfehler erst zur Laufzeit des Werkzeugs identifiziert werden. Die Sprachberarbeitungswerkzeuge können nur bedingt wiederverwendet bzw. kombiniert werden.

RQ5 Generatoren: NineML ist eine rein deskriptive Sprache zur Modellspezifikation und bietet somit vorerst keine direkte Unterstützung für die Codegenerierung bzw. für die Lösung der Differenzialgleichungen. Daher wird bei NineML ein externes Generierungsframework bzw. ein externer Solver für Gleichungen vorausgesetzt. NineML gibt darüber hinaus auch keine methodischen und technologischen Vorgaben für diese Komponenten vor.

RQ6 Benutzerfreundlichkeit: Das Metamodell von NineML ist durch ein XSD-Schema beschrieben. Somit könnten die NineML Modelle durch ein kompatibles Framework eingelesen werden. NineML stellt zwei unterschiedliche Schnittstellen bereit, um mit Modellen zu arbeiten [Nin17a]. Zum einen ist dies eine Python-basierte Schnittstelle, die das Einlesen bzw. das Verarbeiten der Modelle erlaubt. Zum anderen existiert eine Schnittstelle im LISP Dialekt Chicken Scheme [Chi17], die denselben Funktionsumfang anbietet. Die Arbeitsweise von NineML wird in einem technischen Bericht beschrieben. Die Benutzung wird in einem Tutorial [nin17b] dokumentiert. Da NineML keine Möglichkeit zur Simulation bzw. Analyse der Differenzialgleichungen anbieten, müssen diese Funktionen von dem Endanwender eigenständig implementiert und integriert werden.

3.3.2 Low Entropy Model Specification

Low Entropy Model Specification (LEMS) [CGC+14] ist eine XML-basierte Modellierungssprache, mit der die Struktur und das dynamische Verhalten von biologischen Modellen spezifiziert werden können. LEMS stellt eine Grundstruktur für andere Modellierungssprachen bereit, auf welcher höhere Modellierungssprachen aufbauen können. Die konkrete Syntax von LEMS wird wie im Fall von NineML durch ein XML-Schema definiert.

Der grundlegende Gedanke hinter LEMS ist, eine klare Trennung zwischen Modellen und Modellinstanzen zu schaffen. In LEMS bedeutet dies speziell, dass mathematische Gleichungen und Parameter, welche biologische und chemische Vorgänge beschreiben, strikt von konkreten Instanzen getrennt werden. Somit ist es zum Beispiel möglich, die generelle Verhaltensweise eines Ionenkanals durch mathematische Ausdrücke mit Parametern zu beschreiben und dann einen konkreten Ionenkanal durch eine Belegung dieser Parameter zu definieren. Ein Modellelement wird in LEMS durch ein `ComponentType`-Element definiert. Dieses spezifiziert lediglich die Struktur und Dynamik einer Menge von Modellen, welche die gleichen zugrundeliegenden mathematischen Definitionen verwenden. `ComponentType`-Elemente erlauben es, andere `ComponentType`-Elemente zu erweitern. Dies ist sehr ähnlich zum Prinzip der Vererbung in einer polymorphen objektorientierten Programmiersprache.

Ein wesentlicher Bestandteil eines `ComponentType` ist der `Dynamics`-Block, welcher auf der rechten Seite in Abbildung 3.4 zu sehen ist. Der `Dynamics`-Block spezifiziert das Laufzeitverhalten eines Modellelements (beispielsweise eines Neurons oder eines Ionenkanals). Es gibt die Möglichkeit, zeitabhängige Gleichungen aufzustellen. Diese Gleichungen definieren, wie sich Zustandsvariablen in Abhängigkeit mit der Zeit verändern. Weiterhin ist es möglich, sogenannte Events zu beschreiben. Events können einen diskreten Einfluss auf die Werte der Zustandsvariablen haben.

LEMS

abstract layer

```
1 <ComponentType name="cell1">
2   <Parameter name="threshold" dimension="voltage" />
3   <Parameter name="refractoryPeriod" dimension="time" />
4   <Parameter name="capacitance" dimension="capacitance" />
5 </ComponentType>
6
7 <ComponentType name="cell2" extends="cell1">
8   <Parameter name="leakConductance" dimension="conductance" />
9   <Parameter name="leakReversal" dimension="voltage" />
10  <Parameter name="deltaV" dimension="voltage" />
11  <EventPort name="spikes-in" direction="in" />
12  <Exposure name="v" dimension="voltage" />
13  <Dynamics>
14    <StateVariable name="v" exposure="v" dimension="voltage" />
15    <TimeDerivative variable="v" value="leakConductance *
16              (leakReversal - v) / capacitance" />
17    <OnEvent port="spikes-in">
18      <StateAssignment variable="v" value="v + deltaV" />
19    </OnEvent>
20  </Dynamics>
22 </ComponentType>
23
```

Variable which is defined as a parameter

user layer

```
24
25 <Component id="cell2cpt" type="cell2" leakReversal="-50mV"
26           deltaV="50mV" threshold="-30mV" leakConductance="50pS"
27           refractoryPeriod="4ms" capacitance="1pF" />
```

Abbildung 3.4: Ein einfaches Integrate-and-Fire Modell als LEMS Model [lem17]. Die Vererbungsbeziehung zwischen `cell1` und `cell2` erlaubt die Benutzung von Parametern in `cell1` im `cell2`-Neuron.

Darüberhinaus bietet LEMS die Möglichkeit, das Verhalten mittels eines endlichen Automaten zu spezifizieren. Die zur Anwendung kommende Automatenart basiert auf dem Konzept des endlichen Automaten [Bra84, HUM02, Rum96]. Bei diesem Formalismus kann eine Ausgabe erfolgen, während ein Zustand des Automaten aktiviert ist. Innerhalb eines Zustandes können verschiedene Gleichungen für Zustandsvariablen definiert werden, welche genau dann aktualisiert werden, wenn dieser Zustand aktiv ist. Verschiedene Zustände können via Transitionen mit optionalen Bedingungen verbunden werden. Eine Transition wird genau dann ausgeführt, wenn der Quellzustand aktiv ist und die Bedingung der Transition erfüllt ist.

Abbildung 3.4 demonstriert ein Beispiel der Definition zweier Neuronenmodelle im `Abstract-Layer` und deren exemplarische Instanziierung im `User-Layer`. Zunächst werden zwei `ComponentType`-Elemente deklariert, welche jeweils Neuronen darstellen. `cell1` (vgl. Zeilen 1-5) enthält lediglich drei Parameter. `cell2` (eine Spezialisierung von `cell1`) konkretisiert die Beschreibung weiter durch neue Parameter. Dabei erbt `cell2` alle Elemente von `cell1`. Das `cell2`-Element demonstriert auch, wie eine Dynamik in LEMS spezifiziert werden kann. Die Zustandsvariable `V` (vgl. Zeile 14) gibt an, wie sich das Potential über die Zellmembran in Abhängigkeit der Zeit innerhalb der Zelle verändert. Dies wird mithilfe einer mathematischem Gleichung definiert (vgl. Zeile 15). Schließlich können die zuvor definierten Modelle im `User-Layer` benutzt werden. Die Zeilen 25-27 demonstrieren die Instanziierung des `cell2`-Neurons.

RQ1 Modellierung: Ähnlich zu NinenML bietet LEMS durch die Trennung von Modellen und Modellinstanzen ein Konzept an, welches ein gewisses Maß an Modularität ermöglicht. Die abstrakte Spezifikation der LEMS-Modelle ermöglicht deren einfache Portierung auf unterschiedliche Plattformen. Das XSD-basierte Metamodell wirkt sich unmittelbar auf die Syntax der Sprache aus, was sowohl die Klarheit als auch die Kompaktheit der NineML-Modelle negativ beeinflusst.

RQ2 Syntax: Die LEMS-Syntax hat zwei wesentliche Defizite. Zum einen ist sie aufgrund der XML-Repräsentation schwer nachvollziehbar. Zum anderen sind die Modelle schwer zu schreiben. Zwar wäre es möglich passende Editoren dafür zu schreiben, zum Zeitpunkt der Erstellung dieser Arbeit existierte jedoch keine Werkzeugunterstützung. Die verwendeten sprachlichen Elemente sind sehr abstrakt und Allgemeingültig definiert. So kann ein `ComponentType` ein Neuron oder ein Ionen-Kanal sein.

RQ3 Wiederverwendung: Ähnlich zu NineML findet die Wiederverwendung in LEMS auf zwei Ebenen statt. Zum einen können die `ComponentClass`-Elemente während der Instanziierung parametrisiert werden. Zum anderen können die `ComponentClass`-Elemente selbst in einer Vererbungsbeziehung stehen um auf diese Art und Weise Modellabschnitte gemeinsam und nicht redundant wiederzuverwenden.

RQ4 Metamodell: Das Metamodell wird mithilfe eines XSD-Schemas definiert. Das hat den Vorteil, dass es durch existierende XSD-Bibliotheken und Werkzeuge verarbeitet werden kann. Andererseits gibt es keine direkte Unterstützung für die sprachliche Wiederverwendung und Erweiterung.

RQ5 Generatoren: Es gibt eine Menge an Codegeneratoren für LEMS, die verschiedene Zielplattformen unterstützen [VCC+14, CGC+14]. Dazu gehören Modelica [Fri10], MATLAB [GBY08] und die Simulatoren NEURON und Brian (vgl. Unterabschnitt 3.3.5 für Details dieser Simulatoren). Auch benutzerdefinierte Codegeneratoren können entwickelt werden. Für die Codegenerierung benutzt LEMS einen templatebasierten Ansatz. Dennoch stellt LEMS keine dedizierten Mechanismen für das Traversieren der Modelle bereit: Die Unterstützung der Codegenerierung beschränkt sich auf das Bereitstellen eines Template-Frameworks, bietet aber keine Verzahnung mit dem Modell-AST.

RQ6 Benutzerfreundlichkeit: LEMS bietet unterschiedliche Schnittstellen um Modelle zu parsen, validieren und simulieren. Zum einen existiert eine Java-Schnittstelle. Zum anderen können LEMS-Modelle aus einer Python-Schnittstelle verarbeitet werden. Es existieren zwei Simulatoren [VCC+14] (einer ist Java-basiert, der andere ist Python-basiert), die in der Lage sind, LEMS Modelle zu validieren und zu simulieren. Während der Simulation werden auch die Differenzialgleichungen gelöst. Dafür wird stets das Runge-Kutta-Verfahren [AP98] verwendet.

3.3.3 NeuroML

Die NeuroML [GHH+01, CGC+15] ist ein Ansatz um detaillierte Neuronenmodelle und neuronale Netzwerke zu beschreiben, mit dem Ziel des Austausches komplexer Neuronen- und Netzwerkmodelle zwischen Simulatoren. Im Unterschied zu NineML und LEMS können NeuroML-Modelle sehr komplexe morphologische Eigenschaften aufweisen. Ein Alleinstellungsmerkmal von NeuroML ist die Möglichkeit, die dreidimensionale Struktur von Neuronen, Synapsen und darauf aufbauenden Netzwerken zu spezifizieren, indem dreidimensionale Koordinaten für die Modellelemente angegeben werden.

NeuroML ist dafür optimiert komplexe Neuronen aus mehreren Segmenten (engl: compartmental models) darzustellen. Strukturell wird die Sprache in drei Schichten unterteilt, die für die Beschreibung der Elemente auf unterschiedlichen Skalen der biologischen Detaillierung verantwortlich sind. Jede Schichte wird durch eine eigene Sprache definiert.

Level 1 Diese Schicht beschreibt die Morphologie der Zelle mithilfe der *MorphML*. Ein MorphML-Modell besteht aus mehreren Segmenten (compartments) einschließlich deren Raumposition, Größe und Form. Zusätzlich ist es möglich, Metadaten zu modellieren. Zum Beispiel können Angaben zum Experiment bzw. der Publikation, aus welchem das Modell hervorgegangen ist, als Metadaten gespeichert werden.

Level 2 Diese Schicht benutzt die *ChannelML*, um spannungsgesteuerte Membraneigenschaften einschließlich der statischen und plastischen Leitwerte zu beschreiben. Die ChannelML erweitert die MorphML um die Möglichkeit, die genaue Platzierung und Dichte der Membran im Zellmodell des Neurons zu definieren.

Level 3 Diese Schicht benutzt *NetworkML*, um neuronale Netzwerke aus einzelnen Neuronen zu beschreiben, die mithilfe von MorphML und ChannelML spezifiziert sind. Dies schließt deren synaptische Verbindungen mit ein.

Vordefinierte Modellierungselemente können in NeuroML benutzt werden, um die Dynamik der Segmente, Ionenkanäle und Synapsen zu beschreiben. Die Wiederverwendung der vordefinierten Basiselemente erlaubt eine kompakte Darstellung der NeuroML-Modelle. Falls kein passendes vordefiniertes Element bereits existiert, kann ein neues Element mithilfe von LEMS (vgl. Unterabschnitt 3.3.2) definiert werden. Auch bei NeuroML handelt es sich um eine XML-basierte Modellierungssprache. In Weiteren wird ein konkretes Beispiel für ein Neuronenmodells mit mehreren Segmenten vorgestellt. Es handelt sich um ein Modell aus der MorphML.

MorphML verfolgt hauptsächlich zwei Ziele. Zum einen wird sie verwendet, um die Morphologie (d.h. die Struktur und Form) von Neuronen zu definieren. Zum anderen können weiterführende Informationen (Metadaten) über Modellbestandteile ergänzt werden. Im Gegensatz zu NineML setzt NeuroML einen großen Fokus auf die Morphologie einzelner Neuronen. Um Neuronen und deren Bestandteile, wie etwa Dendriten, in Form und Struktur zu beschreiben, ist es nötig, von der Realität zu abstrahieren, da die Morphologie eines Neurons nicht mit beliebigem Detailgrad dargestellt werden kann. Dabei hängt der Grad der Abstrahierung vom jeweiligen Ziel des Modells ab. Um dreidimensionale Elemente wie Neuronen zu beschreiben, verwendet MorphML `segment`-Elemente. Diese Elemente bestehen aus einem Start- und Endpunkt. Diese Punkte sind dreidimensionale Vektoren. Zusätzlich wird der Durchmesser angegeben. Dadurch wird eine dreidimensionale Form mit dem gewünschten Detaillierungsgrad approximiert.

In Abbildung 3.5 wird dieser Ansatz verdeutlicht. `segment`-Elemente stellen den fundamentalen Baustein für die Beschreibung der Form und Struktur von Neuronen in NeuroML dar. Neben den einzelnen `segment`-Elementen besteht die Möglichkeit, mehrere `segment`-Elemente zu einer semantischen Einheit zusammenzufassen. Dieser Zusammenschluss von mehreren `segment`-Elementen wird durch ein `cable`-Element realisiert. Für `cable`-Elemente ist es möglich, auf höheren Schichten elektrische Eigenschaften zu definieren. Dies kann beispielsweise dazu genutzt werden, um die Leitfähigkeit einzelner Dendriten zu beschreiben.

Die zuvor erwähnten Metadaten ermöglichen es Forschern, die Herkunft von Modellkomponenten zu verfolgen und Hintergrundinformationen zu Modellen bereitzustellen. Weitere Metadaten können unter anderem Autorenlisten, Literaturangaben, Referenzen und Ähnliches sein. Diese Informationen helfen dabei, das Modell besser zu verstehen.

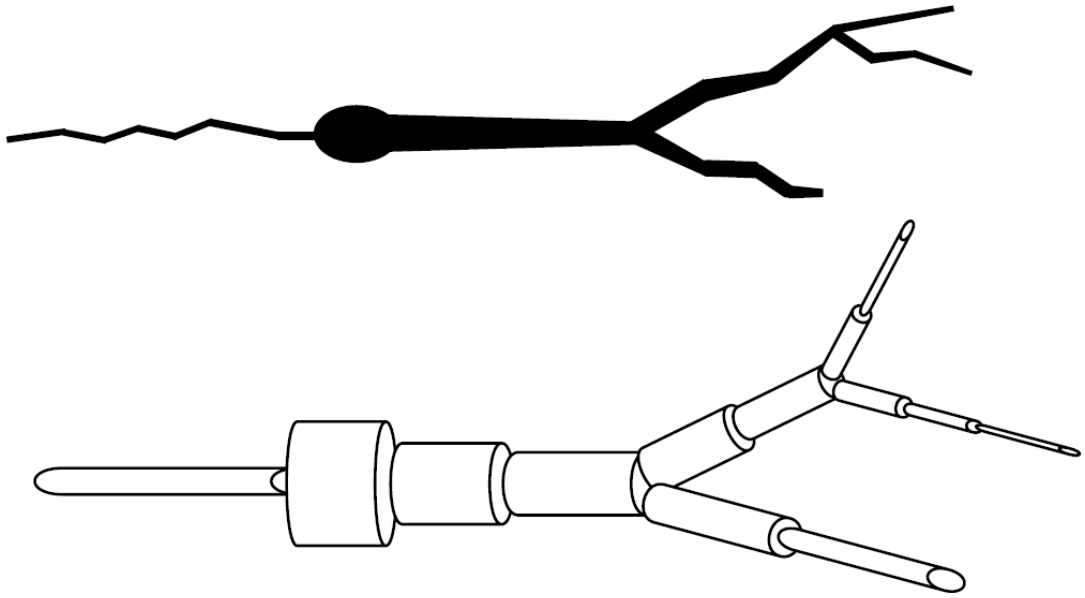


Abbildung 3.5: Stilisierte Darstellung eines Neurons mithilfe einer endlichen Approximation durch zylindrische Segmente [GHH+01]

MorphML

```
1  <morphml>
2    <cells>
3      <cell name = "SimpleCell">
4        <segments>
5          <segment id ="0" name="Soma" cable="0">
6            <proximal x="0" y="0" z="0" diameter="10"/>
7            <distal x="10" y="0" z="0" diameter="10"/>
8          </segment>
9            <segment id ="1" name="Dendrite" parent="0" cable="1">
10           <proximal x="10" y="0" z="0" diameter="3"/>
11           <distal x="20" y="0" z="0" diameter="3"/>
12         </segment>
13       </segments>
14       <cables>
15         <cable id="0" name="SomaCable">
16           <meta:group>soma_group</meta:group>
17         </cable>
18         <cable id="1" name="DendriteCable" >
19           <meta:group>dendrite_group</meta:group>
20         </cable>
21       </cables>
22     </cell>
23   </cells>
24 </morphml>
```

Abbildung 3.6: Eine Umsetzung des Beispieles aus Abbildung 3.5 als MorphML-Modell

In Abbildung 3.6 ist ein MorphML-Beispiel zu sehen, in dem eine Zelle definiert wird. Die Zellform wird durch zwei `segment`-Elemente beschrieben (vgl. Zeilen 5-11). In den

Zeilen 15-20 werden semantische Einheiten aus den `segment`-Elementen gebildet. So beschreibt ein `segment`-Element den Zellkörper und das andere `segment`-Element beschreibt einen Dendriten.

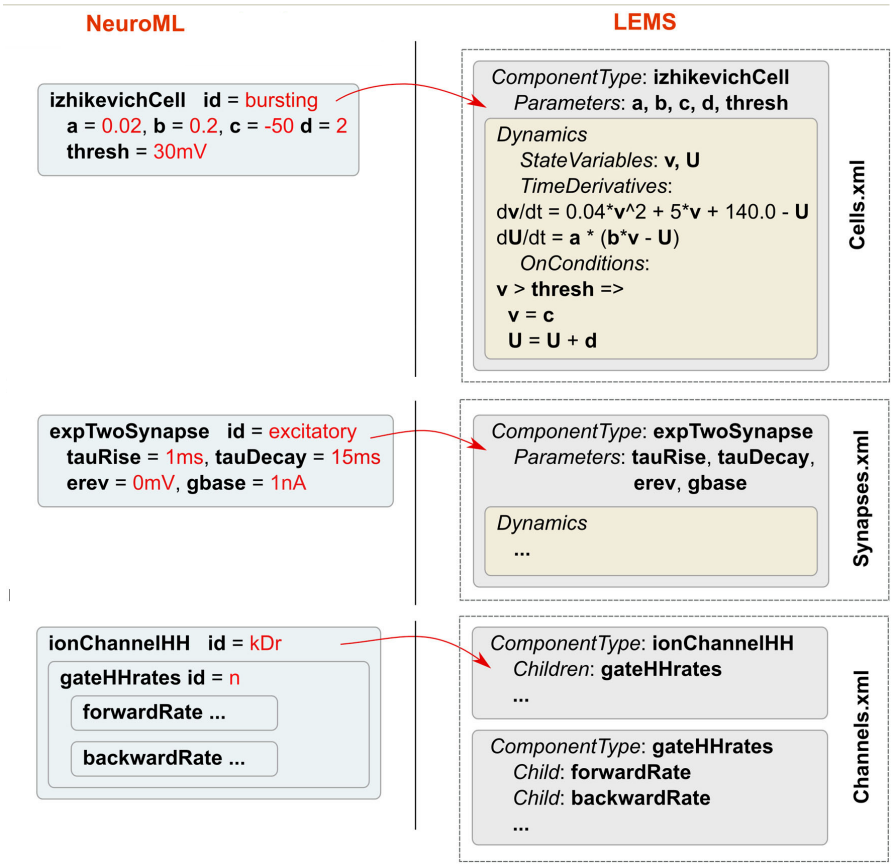


Abbildung 3.7: Zusammenhang zwischen LEMS und NeuroML. NeuroML greift für die Deklaration von Neuronen und Ionen-Kanälen auf die LEMS-Definitionen zurürck [VCC+14]. Im vorliegenden Beispiel wird ein Neuron, eine Synapse und ein Ionen-Kanal aus LEMS in NeuroML instanziiert.

Um das physikalische Verhalten der einzelnen Bestandteile zu beschreiben, greift NeuroML auf LEMS zurück. Abbildung 3.7 visualisiert diesen Zusammenhang. LEMS definiert die Dynamik der einzelnen Bestandteile beispielsweise von einem Neuron. Dieses Neuron kann ähnlich wie im `User-Layer` von LEMS instantiiert werden. `Level 2` und `3` der NeuroML beschäftigen sich mit dem Aufbau von neuronalen Netzwerken und werden, da es nicht der Fokus dieser Ausarbeitung ist, nicht tiefer behandelt. Abbildung 3.7 visualisiert den erläuterten Zusammenhang zwischen NeuroML und LEMS.

RQ1 *Modellierung:* NeuroML in der neusten Version greift für das Modellieren des Neuronenverhaltens [VCC+14] auf LEMS zurück. Das XSD-basierte Metamodell wirkt sich unmittelbar auf die Syntax der Sprache aus, was sowohl die Klarheit als auch die Kompaktheit der NeuroML-Modelle negativ beeinflusst. Im Gegensatz zu NineML gibt es in NeuroML jedoch wesentlich mehr Sprachelemente, welche klar voneinander unterscheidbar sind. Dadurch sind Modelle besser zu verstehen, da Sprachelemente direkt Elementen aus den Neurowissenschaften zugeordnet werden können. Ein weiterer Vorteil ist, dass aufgrund der Aufteilung der Sprache in drei Ebenen die Modularität einzelner Sprachkomponenten gegeben ist.

RQ2 *Syntax:* Ein wesentlicher Kritikpunkt bei NeuroML ist, dass Modelle in XML geschrieben werden. XML wurde in erster Linie dafür entwickelt, um Daten plattformunabhängig auszutauschen. Daher scheint es zunächst sinnvoll, diese Technik zu verwenden. Jedoch ist es schwierig, große neuronale Modelle, welche in XML definiert wurden, zu lesen und zu verstehen. Dieser Kritikpunkt wird etwas abgeschwächt, da es für die oben genannten Sprachen viele Tools gibt, die das Modellieren erleichtern. Zum Beispiel ist neuroConstruct [GSS07] ein grafisches Tool, welches es ermöglicht, Modelle für NeuroML grafisch zu erstellen.

RQ3 *Wiederverwendung:* Ähnlich zu NineML und LEMS findet die Wiederverwendung auf der Ebene der Instanziierung von bereits existierenden Modellen statt. Dabei besteht die Möglichkeit, Modelle an den vorgesehenen Stellen zu parametrisieren.

RQ4 *Metamodell:* Da das Metamodell von NeuroML nach demselben Prinzip wie das von LEMS aufgebaut ist, teilt es die selben Vor- und Nachteile. Das Metamodell wird durch ein XSD-Schema definiert. Das hat den Vorteil, dass es durch existierende XSD-Bibliotheken und -Werkzeuge verarbeitet werden kann. Es gibt jedoch keine direkte Unterstützung für die Sprachwiederverwendung und Spracherweiterung. Auch die Ausdruckskraft des Metamodells ist durch die Allgemeingültigkeit des XML Formats eher schwach ausgeprägt.

RQ5 *Generatoren:* Da NeuroML gemeinsam mit LEMS entwickelt wird, stehen dieselben Ansätze für die Codegenerierung zur Verfügung. Die Codegeneratoren von NeuroML unterstützen verschiedene Zielplattformen [VCC+14, CGC+14]. Dazu gehören Modelica [Fri10], MATLAB [GBY08], sowie die Simulatoren NEURON und Brian (vgl. Unterabschnitt 3.3.5). Auch eigene Codegeneratoren können entwickelt werden. Für die Codegenerierung benutzt NeuroML einen Template-basierten Ansatz. Auch NeuroML stellt keine Mechanismen für das Traversieren der Modelle bereit. Die Unterstützung der Codegenerierung beschränkt sich auf das Bereitstellen eines Template-Frameworks. Es bietet ebenfalls keine Verzahnung mit dem Modell-AST.

RQ6 *Benutzerfreundlichkeit:* Auch NeuroML bietet unterschiedliche Schnittstellen, um Modelle zu parsen, validieren und simulieren. Zum einen ist es eine Java-Schnittstelle, zum anderen eine Python-Schnittstelle.

3.3.4 XML als konkrete Syntax

Alle oben vorgestellten Sprachen basieren auf XML [YBP+04]. Die XML-Notation wird auch dafür verwendet, um die konkrete Syntax der jeweiligen Sprache zu repräsentieren. Der Grund für diese Wahl ist, dass ein ausgereiftes Ökosystem für die XML-Verarbeitung existiert. Deswegen können Lexer und Parser [ASU86] aus diesem Ecosystem wiederverwendet werden. Auch wenn dieser Ansatz auf den ersten Blick Vorteile verspricht, überwiegen doch die Nachteile.

Als erstes erschwert die Ausführlichkeit der XML-Modelle sowohl das Nachvollziehen als auch das Erstellen dieser Modelle [Che01]. Deswegen benötigt die praktische Anwendung dieser Sprachen die Entwicklung neuer aufwendiger Visualisierungs- bzw. Verarbeitungswerkzeuge. Das wiederum nivelliert die Vorteile der Wiederverwendung der Sprachverarbeitungswerkzeuge aus dem XML-Ökosystem. Als zweites reichen die existierenden Sprachwerkzeuge wie beispielsweise Parser öftmals nicht aus, um beispielsweise mathematische Ausdrücke zu verarbeiten, was abermals eine Eigenentwicklung von Sprachinfrastruktur bedeutet. Im Falle von NineML gibt es die Möglichkeit, mathematische Ausdrücke in einem `MathInline`-Block zu definieren (vgl. Zeile 13 in Abbildung 3.8). Um die syntaktische Korrektheit dieses Ausdrucks in NineML zu überprüfen, muss jedoch wieder ein neuer Parser entwickelt werden.

XML

```
1  <Dynamics >
2    <StateVariable name ="V" dimension ="voltage" />
3    <StateVariable name ="U" dimension ="voltagePerTime " />
4    <Alias name ="rv" dimension = "none">
5      <MathInline >V*U</ MathInline>
6    </Alias>
7    <Regime name = "subthresholdRegime">
8      <TimeDerivative variable ="U">
9        <MathInline>a*(b*V - U)</MathInline>
10     </TimeDerivative>
11     <TimeDerivative variable ="V">
12       <MathInline>0.04* V*V + 5*V + 140.0 - U + iSyn </MathInline>
13     </TimeDerivative>
14   </Regime>
15 </Dynamics>
16
```

Abbildung 3.8: Ausschnitt eines NineML-Modells. Um einen mathematischen Term $rv = V/U$ zu definieren, sind drei Zeilen XML-Code notwendig (vgl. Zeilen 4-6). Der `MathInline`-Block in Zeile 13 enthält eine syntaktisch unstrukturierte Zeichenkette, die nicht mithilfe von XML-Tools auf die syntaktische Korrektheit validiert werden kann.

3.3.5 Simulator-spezifische Modellierungssprachen

Neben den externen deskriptiven Modellierungssprachen existieren auch DSLs, die nahtlos in einen Simulator eingebettet sind. Aufgrund der Verbreitung und Qualität stellt dieses Kapitel den Simulator NEURON [Hin93, HC97] für biologisch realistische Neuronen und Brian als Beispiel für die schnelle und leichtgewichtige Entwicklung von Neuronen und Netzwerksimulationen mit eher prototypischem Charakter vor.

NEURON

NEURON [Hin93, HC97, Bed15] ist ein neuronaler Simulator mit einer vielseitigen Umgebung für die Simulation von detailierten biologischen neuronalen Netzwerken. Für die Modellierung von Neuronen und die Durchführung von Simulationen stellt NEURON ein grafisches Werkzeug und eine interpretierte DSL Higher Order Calculator (HOC) [HC15] zur Verfügung. Auf der zellulären Ebene kann NEURON die Zusammenhänge von präund postsynaptischen Prozessen darstellen. Auch fein granulierte Dendriten und morphologische Eigenschaften eines Neurons können modelliert werden. Auf der Netzwerkebene ist es möglich, mithilfe von Neuronen Prozesse der Informationsverarbeitung in biologischen neuronalen Netzwerken zu untersuchen.

Die eingebaute Sprache HOC lehnt sich syntaktisch an C an. Neben vordefinierten biologischen Neuronenmodellen bietet NEURON mit NMODL [CH06a, HC00] eine DSL an, um neue Neuronenmodelle zu definieren. NMODL strukturiert Neuronen mithilfe von Blöcken, die bestimmte Namen und Attribute zusammenfassen. Mit solchen Blöcken kann das Verhalten des Neurons durch imperativen Code bzw. Differenzialgleichungen definiert werden. Schließlich erlaubt NMODL es, C-Code direkt ins Neuronenmodell zu integrieren, um performancekritische Bereiche zu optimieren. Ein wichtiges Merkmal von NMODL ist, dass Syntax und Semantik der Domänennotation und -terminologie ähneln. Dadurch können Neurowissenschaftler kompakte Modellspezifikation erstellen, ohne sich mit Implementierungsdetails zu beschäftigen.

Der Ansatz von NEURON ist es, biologische und morphologische Aspekte des Neuronenmodells, wie z.B. die Form eines Neurons, von der numerischen Modellierung zu trennen. Der grundlegende Baustein des Neurons in NMODL ist ein Abschnitt (engl: section). Ein Abschnitt ist ein unverzweigtes durchgehendes Kabel mit variierenden anatomischen und biophysiologischen Eigenschaften entlang dessen Länge. Da solche Modelle sehr komplex werden können, bietet NEURON einen grafischen Editor, in dem diese komplexen verzweigten Gebilde grafisch aufgebaut werden können. Der Editor erlaubt es, Neuronen auf Basis von Kabelabschnitten zu erstellen, denen physikalische Eigenschaften zugeordnet werden können, ohne programmieren zu müssen.

In Abbildung 3.9 wird ein Beispiel eines NMODL-Neurons vorgestellt. Im `UNITS`-Block definiert das Beispiel kürzere Synonyme für die physikalischen Einheiten. Danach folgt die Definition des eigentlichen Neurons. Die Anweisung `SUFFIX CaT` definiert einen Na-

men, mit dem dieses Neuron aus anderen Modellen referenziert werden kann. Somit kann auf die im Neuron definierten Variablen nach folgendem Schema zugegriffen werden: $varName\_CaT$.

```
UNITS {
    (mV) = (millivolt)
    (mA) = (milliamp)
}

NEURON {
    SUFFIX CaT
    USEION ca READ eca WRITE ica
    RANGE gmax
}


PARAMETER {
    gmax  = 0.002 (mho/cm2)
}

STATE {
    r s d
}
```

```
PROCEDURE settables(v (mV)) {
    LOCAL  bd                                                    NMODL

    ralpha = 1.0/(1.7+exp(-(v+28.2)/13.5))
    rbeta  = exp(-(v+63.0)/7.8)/(exp(-(v+28.8)/13.1)+1.7)

    salpha = exp(-(v+160.3)/17.8)
    sbeta  = (sqrt(0.25+exp((v+83.5)/6.3))-0.5) *
                    (exp(-(v+160.3)/17.8))

    bd     = sqrt(0.25+exp((v+83.5)/6.3))
    dalpha = (1.0+exp((v+37.4)/30.0))/(240.0*(0.5+bd))
    dbeta  = (bd-0.5)*dalpha
}

DERIVATIVE states {
    settables(v)
    r' = ((ralpha*(1-r)) - (rbeta*r))
    d' = ((dbeta*(1-s-d)) - (dalpha*d))
    s' = ((salpha*(1-s-d)) - (sbeta*s))
}
```

Abbildung 3.9: Eine exemplarische NMODL-Implementierung eines Neurons basierend auf der Dynamik aus [WH91]. Dieses Beispiel zeigt die wesentlichen Komponenten: Zustands-, Parameter-, Alias- und Aktualisierungsblock für die Zustandsvariablen. Dieses Modell lehnt sich an [GS117] an.

Mit der `USEION`-Anweisung im `NEURON`-Block werden die Ionenkanäle des Neurons definiert. Im vorliegenden Beispiel wird ein Calcium-Ionenkanal verwendet. Dieser Kanal bekommt das Equilibriumpotential als Eingabe und berechnet den Strom von Calciumionen als Ausgabe. Auch andere Ionenmechanismen könnten definiert werden. Die Variablen aus dem `PARAMETER`-Block sind dadurch charakterisiert, dass sie sich nicht aus dem Zustand des Neurons ableiten lassen und während der Simulationszeit konstant bleiben. Diese Werte können aus dem grafischen Editor oder einem HOC-Programm gesetzt werden. NMODL bietet zudem einen Mechanismus, um abgeleitete Variablen zu definieren. In Abbildung 3.9 definiert die `settables`-Methode Regeln dafür, wie sich beispielsweise `ralpha` oder `rbeta` aus dem Wert von `v` bestimmen lassen [WH91]. Die Aktualisierungsvorschrift von Zustandsvariablen aus dem `STATE`-Block wird im vorliegenden Beispiel anhand dreier Differenzialgleichungen im `DERIVATIVE`-Block definiert.

RQ1 *Modellierung:* NEURON bietet die Möglichkeit, Neuronen, Netzwerke und Simulationen in einer kompakten domänennahen Notation zu formulieren. Das führt zu einer klaren Modellbeschreibung. Dennoch leidet die Portabilität aufgrund der starken Kopplung der Modelle bzw. der entsprechenden Werkzeuge an die NEURON-Infrastruktur. Desweiteren erlaubt NEURON, nativen C-Code direkt in die Modellspezifikation einzubetten und von da aus auf die Interna des Simulators zuzu-

greifen. In diesem Fall beeinflusst dies die Portabilität und Klarheit der Modelle jedoch negativ. Die Wiederverwendung der Modelle basiert auf der Verwendung der impliziten Namensgebung, was bei einer Vielzahl von Modelle unübersichtlich werden kann.

RQ2 *Sprache:* Die Syntax von NMODL dient dazu, das Problem der Definition eines Neurons zu lösen. Dafür benutzt NMODL eine kleine Menge von einfachen Sprachelementen, die spezifische Aspekte der Modellierung lösen. Die Syntax von NMODL lehnt sich an die Syntax der Programmiersprache C an. Diese Wahl richtet sich daher eher an Computerwissenschaftler, die mit der Sprache vertraut sind. Manche Sprachelemente sind dennoch nicht deskriptiv und konsistent (vgl. die `USEION`-Anweisung).

RQ3 *Wiederverwendung:* Die Wiederverwendung ist kaum gegeben. Zwar können Ionenkanäle wiederverwendet werden, es gibt jedoch keine Möglichkeit für die Wiederverwendung von Neuronenteilen.

RQ4 *Metamodell:* Das Metamodell von NMODL steht für die Erweiterung nicht zur Verfügung. Daher ist es auch nicht möglich, die Sprache an eigene Anforderungen anzupassen, zu erweitern oder wiederzuverwenden. Die Modellierungssprache ist mit dem Simulator monolithisch verzahnt.

RQ5 *Generatoren:* Aus NMODL-Modellen wird eine C++-Implementierung für den NEURON-Simulator generiert. Dieser Generator ist fest in den NEURON-Simulator eingebettet und lässt sich nicht austauschen. Der Generator ist mit einem *Visitor*-basierten Ansatz implementiert, der den Zielcode während der Traversierung des ASTs erstellt.

RQ6 *Benutzerfreundlichkeit:* NMODL ist ein inhärenter und untrennbarer Teil von NEURON, der mit dem Simulator stets ausgeliefert wird. Für die Benutzung des Simulators bzw. der NMODL-Sprache existiert eine Vielzahl von Anleitungen und Bücher [HC97, CH06a].

Brian

Der Simulator Brian [GB08], der zum Zeitpunkt der Erstellung dieser Arbeit in Version 2 vorliegt [SGB15], ist ein Python-basierter Simulator für die Entwicklung und Simulation von biologischen Neuronen und Netzwerken. Ein wichtiges Ziel von Brian ist es, eine einfach nutzbare Simulations- und Modellierungsumgebung zur Verfügung zu stellen, um somit die Zeit für die Entwicklung und Formulierung des neuronalen Simulationsquellcodes zu minimieren. Brian und die eingebettete interne DSL soll es Forschern erlauben, eigene Modelle einfach in den Simulator zu integrieren. Somit sind Forscher nicht auf die von Brian bereitgestellte Auswahl von eingebauten Modellen eingeschränkt [GB08].

Brian wurde ausschließlich mit Python [Lut96] entwickelt, da Python eine leichtgewichtige und flexible Programmiersprache mit hohem Verbreitungsgrad in den Neurowissenschaften [LHB14] ist. Zum einen vereinfacht das den Zugang zum Simulator, da die Anwender grundsätzlich mit der Technologie vertraut sind. Zum anderen erlaubt die Wahl einer interpretierten Sprache es, den Simulator auf unterschiedlichen Zielplattformen zu portieren.

Der Brian-Simulator ist so entworfen, dass er gleichzeitig auch ein Python-Modul ist. Dadurch kann der Brian-Simulator transparent in Python-Scripts direkt als Bibliothek eingebunden werden. Um die Leistung des Simulators zu verbessern, verwendet Brian die Bibliotheken NumPy [VDWCV11] und SciPy [JOP14]. Diese können mathematische Berechnungen signifikant beschleunigen. Um eine Simulation durchzuführen, kann entweder ein Skript erstellt oder interaktiv gearbeitet werden.

Um ein eigenes Neuronenmodell zu erstellen, bietet Brian vordefinierte Klassen an. Die Modellierung eines Neurons wird durch eine Klasse `MembraneEquation` realisiert. Abbildung 3.10 demonstriert die Definition eines *Integrate-and-Fire*-Neurons mit Brian. Um mögliche Fehler zu vermeiden, erlaubt Brian es, SI Einheiten zu benutzen. Diese werden sowohl in der Annotation (vgl. Zeile 1), als auch in der Gleichung (vgl. Zeile 7) selbst deklariert. Eine wesentliche Rolle spielt bei der Spezifiaktion die textuelle Definition der Differenzialgleichung, die die Dynamik des Neurons beschreibt (vgl. Zeile 7).

Python

```
1 @check_units(tau=second, v0=volt)
2 def integrate_and_fire(tua, v0):
3 '''
4 A leaky integrate-and-fire model.
5 tau*dv/dt = v0 - vm
6 '''
7   return MembraneEquation('dv/dt = -(v-v0)/tau:volt')
```

Abbildung 3.10: Modellierung eines IaF-Neurons im Brian-Simulator,.

RQ1 *Modellierung:* Die Mischung aus Python-Klassen und textuellen deklarativen Bausteinen für die Definition von Differenzialgleichungen beeinflusst die Klarheit und die Kompaktheit der Neuronenmodelle negativ. Die enge Bindung der Neuronenmodelle an die Simulationsinfrastruktur macht die Neuronenmodelle weder modular noch portierbar.

RQ2 *Syntax:* Syntaktisch baut Brian auf der Python-Programmiersprache auf. Ein Beispiel dafür sind die Differenzialgleichungen, die als Zeichenketten dargestellt werden. Dies hat als Konsequenz, dass keine statischen semantischen Prüfungen durchgeführt werden können, sondern potenzielle Fehler erst während der Laufzeit des Programms erkannt werden. Dadurch leidet auch die Konsistenz und Minimalität der Modellierunssprache, da alle Pythonkonstrukte erlaubt sind.

RQ3 *Wiederverwendung:* Der Brian-Simulator greift auf die Mechanismen der Python-Infrastruktur zurück, um die Wiederverwendung der Neuronenmodelle zu unterstützen. Da Brian-Neuronenmodelle valide Python-Programme sind, können auch alle Mechanismen für die Modulverwaltung und Wiederverwendung benutzt werden, die in Python zur Verfügung stehen.

RQ4 *Metamodell:* Das Metamodell von Brian-Neuronen ist vollständig im Simulator selbst gekapselt. Es kann nicht erweitert werden. Anstatt des Metamodells stehen dem Anwender die vordefinierten Bibliothekenklassen zur Verfügung, die entweder erweitert oder parametrisiert werden können.

RQ5 *Generatoren:* Der Brian-Simulator ist so entworfen, dass eine Codegenerierung aus Modellen nicht notwendig ist. Die deklarativen Teile der Neuronenmodelle werden intern im Simulator interpretiert. Der Ansatz von Brian für die Portierung auf neue Plattformen ist es, den vollständigen Simulator auf der jeweiligen Plattform direkt auszuführen. Dafür wird ein plattformspezifischer Python-Interpreter vorausgesetzt.

RQ6 *Benutzerfreundlichkeit:* Der Brian-Simulator ist ein gut dokumentiertes Projekt. Alle für eine Simulation notwendigen Komponenten sind nach seiner Installation verfügbar. Da der Brian-Simulator als eine Python-Bibliothek umgesetzt ist, bietet er eine nahtlose Anbindung an die Python-Infrastruktur. Die notwendigen Bibliotheken müssen manuell installiert werden.

3.4 Evaluierung der existieren Ansätze

Die Evaluierung der Anforderungen aus Abschnitt 3.2 ist in Tabelle 3.1 zusammengefasst. Es ist leicht ersichtlich, dass keiner der vorgestellten Ansätze alle Anforderungen erfüllen kann. Die meisten Modellierungssprachen sind im Prinzip geeignet, um Neuronenmodelle auszudrücken. Somit wären diese auch Kandidaten für den NEST-Simulator. Dennoch handelt es sich bei den existierenden Sprachen entweder um rein deskriptive XML-basierte Modellierungssprachen oder die jeweilige Modellierungssprache ist sehr stark mit einem Simulator verzahnt und kann nicht erweitert werden.

Eine Möglichkeit die restlichen Anforderungen zu erfüllen wäre eines der existierenden Werkzeuge zu erweitern. NineML, NEUROML und LEMS sind dafür leider ungeeignet, da sie auf einer Seite rein deklarativ und auf der anderen Seite XML-basiert sind. Der Aufwand, die Sprachen an die neuen Anforderungen anzupassen, wäre somit vergleichbar mit der Neuentwicklung einer eigenen DSL. Am passendsten wären die Sprachen von Brian und NEURON, da diese syntaktisch gut strukturierte DSLs sind. Diese Simulatoren bieten jedoch keine Möglichkeit, deren Sprachverarbeitungswerkzeuge modular wiederzuverwenden bzw. zu erweitern [Völ11]. Die jeweiligen Lexer, Parser und Symbol-

tabellen sind eng mit dem jeweiligen Simulator verzahnt und lassen sich nicht modular wiederverwenden.

Requirement	NineML	LEMS	NeuroML	NEURON	Brian
Modellierung					
(RQ1.1) Klarheit:	nein	nein	ja	ja	ja
(RQ1.2) Portabilität:	ja	ja	ja	nein	nein
(RQ1.3) Modularität:	ja	ja	ja	nein	nein
(RQ1.4) Komaktheit:	nein	nein	nein	ja	partiell
Syntax					
(RQ2.1) Konsistenz	ja	ja	ja	ja	nein
(RQ2.2) Minimalität	nein	nein	ja	ja	nein
(RQ2.3) Simplizität	nein	nein	ja	ja	nein
(RQ2.4) Keine Generalisierung:	nein	nein	nein	ja	nein
(RQ2.5) Begrenzung der Elemente:	nein	nein	ja	ja	nein
(RQ2.6) Bekannte Notationen:	nein	nein	nein	ja	ja
(RQ2.7) Deskriptive Notationen:	nein	nein	ja	partiell	nein
(RQ2.8) Unterscheidbare Konstrukte:	nein	nein	ja	ja	nein
Wiederverwendung					
(RQ3.1) Anpassung durch Erweiterung:	ja	ja	nein	nein	ja
(RQ3.2) Bibliotheken:	nein	nein	nein	nein	nein
Metamodel					
(RQ4.1) Modulares Metamodells:	nein	nein	nein	nein	nein
(RQ4.2) Sprachverarbeitungsinfrastuktur:	nein	nein	nein	nein	nein
Generatoren					
(RQ5.1) Unterschiedliche Zielplattformen:	nein	partiell	partiell	nein	ja
(RQ5.2) Handgeschriebene Erweiterungen:	nein	nein	nein	ja	nein
(RQ5.3) Dokumentation:	ja	nein	nein	nein	ja
(RQ5.4) Differenzialgleichungen:	nein	partiell	indirekt	ja	ja
Benutzerfreundlichkeit					
(RQ6.1) Selfcontained	partiell	ja	ja	ja	ja
(RQ6.2) API:	ja	ja	ja	nein	ja
(RQ6.3) Tutorial	ja	ja	ja	ja	ja

Tabelle 3.1: Überblick der Anforderungsauswertung der verwandten Arbeiten

Kapitel 4

Die MontiCore Language Workbench

Die *MontiCore* Workbench [KRV07, KRV08, Kra10] generiert die konkrete und abstrakte Syntax einer domänenspezifischen Sprache auf der Grundlage einer formalen Grammatik [ASU86]. Die konkrete Syntax beschreibt dabei die äußere Form der Sprache, die textuell oder grafisch sein kann. Die abstrakte Syntax definiert die interne Repräsentation der konkreten Syntax in einer für einen Computer effizient verarbeitbaren Form. In [HR04] werden die Begriffe der konkreten und abstrakten Syntax einer DSL detalliert diskutiert. Die MonitCore Workbench stellt zusätzlich eine modulare Infrastruktur für das Parsen von Modellen, den Aufbau der Symboltabllen und zum Prüfen der Kontextbedingungen bereit. Die Codegenerierung wird durch ein erweitertes Freemarker-Framework [Sch12] unterstützt.

Somit unterstützt die MontiCore Workbench alle wesentlichen Aktivitäten während der Sprach- und Generatorentwicklung im Kontext einer Domain Specific Language (DSL). Neben der MontiCore Workbench existieren andere Language Workbenches. Zu den bekanntesten Vertretern zählen: *XText* [EB10], *Spoofax* [KV10] und *MetaEdit+* [KLR96]. Diese und weitere Language Workbenches wurden in [EVDSV$^+$13, Voe14] ausführlich analysiert und diskutiert.

Die Entscheidung für die MontiCore Workbench resultiert aus ihrer Eigenschaft, DSLs und deren Verarbeitungswerkzeuge agil, modular und wiederverwendbar definieren zu können, wodurch sich die Komplexität der Entwicklung einer DSL erheblich reduziert.

4.1 Domänenspezifische Sprachen

Im Laufe dieser Arbeit wird die neue DSL Nest Modeling Language (NESTML) [PBI$^+$16] für die Beschreibung der Neuronen zur Simulation im NEST-Simulator entwickelt. Nachdem die grundlegende Definition einer DSL bereits in der Einleitung gegeben wurde, wird dieser Begriff hier nun detaillierter erklärt. In [Fow10] wird eine DSL definiert als:

Definition 4.1. *(DSL) A computer programming language of limited expressiveness focused on a particular domain.*

Diese Definition hebt zwei wesentliche Eigenschaften einer DSL hervor. Zum einen ist eine DSL eine Programmiersprache, zum anderen besitzt eine DSL eine eingeschränkte

Ausdrucksfähigkeit, die im Vergleich zu einer General Purpose Language (GPL) eine verbesserte Modellierung in einem bestimmten fachlichen Bereich ermöglicht. DSLs orientieren sich an der fachlichen Anwendung in einem bestimmten wohldefinierten Bereich und nicht an deren technischer Realisierung. Die verkleinerte Ausdrucksfähigkeit bedeutet wiederum, dass es möglich ist, bessere semantische Analysen der in dieser Sprache formulierten Modelle durchzuführen.

DSLs können aufgrund ihrer Beschaffenheit in zwei grobe Kategorien eingeteilt werden. [Fow10] unterscheidet zwischen internen und externen DSLs. Eine interne DSL entsteht im einfachsten Fall durch eine sprechende API (engl: fluent API) [Blo08] oder durch die in der Sprache verfügbaren Erweiterungspunkte. Beispielsweise wären das Präprozessor-Framework in den Programmiersprachen Go [DK15] und C++ [Str86] solche Erweiterungspunkte. Mithilfe dieser eingebauten Codegenerierungsfunktionalität kann der Quellcode dieser Sprachen während der Kompilierung verändert und erweitert werden. Einerseits ist es dadurch möglich, kompaktere Programme zu schreiben. Andererseits leidet die Lesbarkeit und Nachvollziehbarkeit solcher Programme, da die vollständige Intention des Programms nicht direkt aus dem Programmcode ersichtlich ist. Hingegen ist eine externe DSL eine eigenständige Sprache, deren Modelle typischerweise in eigenständigen Dateien gespeichert sind. Eine externe DSL hängt weder syntaktisch noch semantisch von einer übergeordneten Sprache ab. Sie verfügt somit über größeres Potential, die fachlichen Konzepte adäquat abzubilden.

Die erhöhte Produktivität der Entwickler und verbesserte Qualität der resultierenden Softwaresysteme [VDK98, SEHV12, FHR08] gelten als Vorteile einer DSL. Die erhöhte Produktivität resultiert aus der Tatsache, dass Modelle in einer passend gewählten Notation im Vergleich zu einer äquivalenten Darstellung in einer GPL kompakter sind. Studien berichten dabei von einer Produktivitätssteigerung um den Faktor drei [KMB+96] bis zehn [Met10]. Die verbesserte Qualität des Softwaresystems hängt vor allem mit der engen Einbindung der Domänenexperten zusammen. Experten können das modellierte System aufgrund ihrer fachlichen Expertise besser verstehen. Durch die Abstraktion von technischen Details können die Experten Modelle optimieren oder sogar neue Modelle eigenständig entwickeln [VS10].

Ein weiterer Vorteil von DSLs ist die klare Trennung der fachlichen Logik von der technischen Implementierung dieser Logik. Diese klare Trennung macht es auch einfacher, Fachwissen in unterschiedlichen Kontexten wiederzuverwenden. Dieses Wissen kann für viele Aufgaben verwendet werden. Zum Beispiel wird dadurch die Migration zu einer anderen technischen Plattform enorm vereinfacht. Die kompakte und fachliche Notation der Modelle führt auch zu einer verbesserten Selbstdokumentation der so modellierten Systeme.

Neben den erläuterten Vorteilen existieren aber auch Risiken im Softwareentwicklungsprozess bei der Verwendung von DSLs. Die hohen Entwicklungskosten für die Konzeption und Umsetzung einer neuen DSL sind das größte Risiko. Zu diesen Kosten zählen sowohl die Kosten für die Implementierung, die Weiterentwicklung und die Wartung der

DSL selbst, als auch Kosten für die Entwicklung der Infrastruktur für die Analyse, die Codegenerierung bzw. die Interpretierung der Modelle.

Die Vorteile einer DSL im Vergleich zur konventionellen Softwareentwicklung werden erst dann bemerkbar, wenn die Kosten der Anfangsinvestition für die DSL-Entwicklung amortisiert sind. Paul Hudak verwendet die ökonomische *Break-Even*-Analyse [Hud98], um Potenziale der DSL-basierten Entwicklung im Vergleich zu entsprechenden Entwicklungskosten abzuschätzen. Leider ist es nicht ohne weiteres möglich, die Maßzahl für den konkreten *Break-Even*-Punkt in einem spezifischen Projekt zu bestimmen. Dennoch helfen Language Workbenches, wie MontiCore, diese Maßzahl signifikant zu senken.

4.2 MontiCore

Die MontiCore Workbench[1] ist eine Workbench zur Erstellung von domänenspezifischen Sprachen. MontiCore verwendet und erweitert das Grammatikformat vom *ANTLR4*-Parsergenerator [Par13], das auf dem EBNF-Formalismus [ASU86] basiert, um zusätzliche Konzepte für die Grammatikwiederverwendung. Die MontiCore-Grammatik ermöglicht lediglich die Definition kontextfreier Sprachen. Neben dem Parser, dem Lexer und Klassen für die abstrakte Syntax, die aus der Grammatik automatisch abgeleitet werden, stellt die MontiCore Workbench eine Infrastruktur bereit, um domänenspezifische Sprachen in den Softwareentwicklungsprozess besser zu integrieren.

Folgende Eigenschaften heben MontiCore gegenüber konkurrierenden Ansätzen hervor:

- MontiCore nutzt ein erweitertes Grammatikformat, das sowohl konkrete als auch abstrakte Syntax definiert. Aus der Grammatik werden Sprachverarbeitungswerkzeuge und Sprachinfrastruktur abgeleitet. Auf der Grundlage einer Grammatik können Delta-Sprachen [HRRS12, HHK+15] transparent spezifiziert werden.

- MontiCore erlaubt eine modulare Definition der konkreten und abstrakten Syntax [Völ11].

- MontiCore realisiert explizite Schnittstellen zwischen Modellen, die eine unabhängige und modulare Verarbeitung und Verschmelzung heterogener Modelle erlaubt [MSNRR16].

- MontiCore unterstützt Techniken für die Sprachkombination einschließlich der Spracherweiterung, Sprachvererbung, Spracheinbettung und Komposition der entsprechenden Sprachverarbeitungswerkzeuge [Völ11].

[1] MontiCore in der Version 4.5.0 wurde für die Erstllung dieser Ausarbeitung verwendet.

- MontiCore besitzt eine Infrastruktur für semantische Modellanalysen [Sch12] sowie Modelltransformationen, die die konkrete Syntax der zu transformierenden DSL wiederverwenden [Wei12, HRW15].
- MontiCore erlaubte die teilweise Generierung der sprachspezifischen Teile der Symboltabellen [HMSNR15].
- MontiCore ermöglicht eine transparente Integration mit der *Eclipse* IDE[2] [Bud04]. Die MontiCore Workbench generiert dabei die Grundstruktur eines Editors [Völ11], der durch manuelle Erweiterungen um weitergehende Funktionalität ergänzt werden kann.
- MontiCore bietet eine explizite Verwaltung der Variabilität sowohl in Sprachdefinitionen als auch in Generierungswerkzeugen [KRR15, GR11, CGR09].

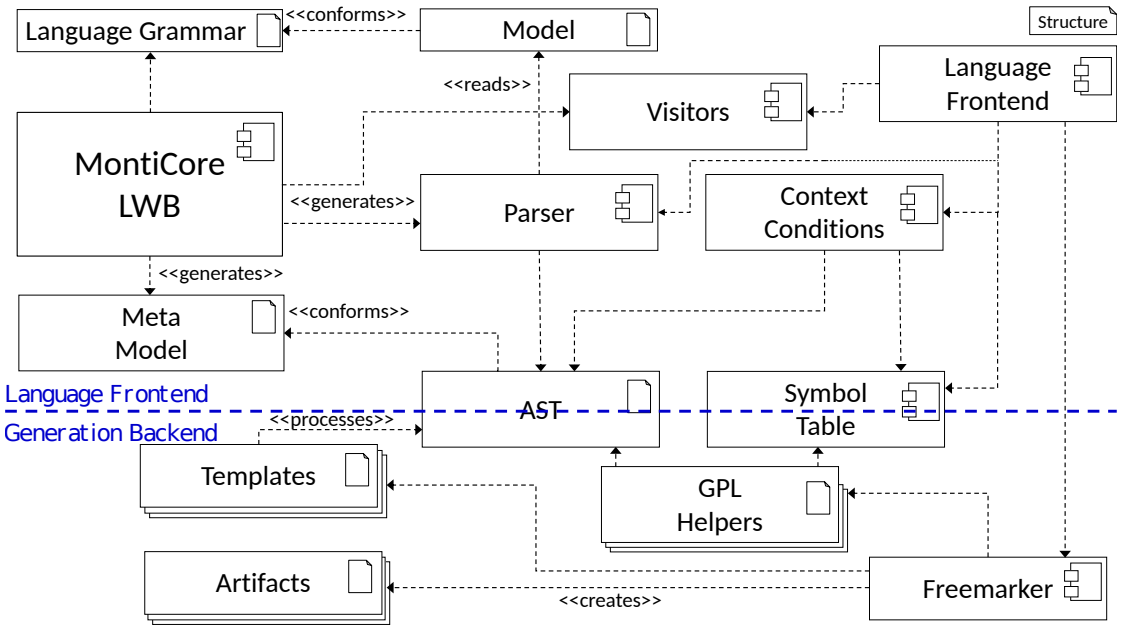


Abbildung 4.1: Wesentliche Komponenten und Aufbau der Sprachverarbeitungsinfrastruktur in der MontiCore Language Workbench.

Abbildung 4.1 gibt eine Übersicht der wesentlichen Bestandteile der MontiCore Workbench anhand eines konkreten Grammatikbeispieles. Die primären Artefakte für die Sprachdefinition sind formale Grammatiken. Das Grammatikformat wird im kommenden Abschnitt detailliert vorgestellt. Auf Basis dieser Grammatiken generiert die MontiCore

[2] `http://www.eclipse.org/`

Workbench Klassen der abstrakten Syntax, die in dieser Ausarbeitung als Metamodell bezeichnet werden. Anschließend erstellt MontiCore Parser-Werkzeuge und Visitoren, die in der Lage sind, das Metamodell als AST zu instanziieren und dabei zu helfen, Algorithmen auf ASTs auszuführen. Zusammen mit der Symboltabelle und den Kontextbedingungen bilden diese Komponenten das Sprachfrontend.

Das Generierungsbackend besteht aus einer Menge von Generierungstemplates und Generierungshilfsklassen. Im Folgenden werden diese Komponenten anhand der NESTML-Sprache erläutert.

4.3 Integriertes MontiCore-Grammatikformat

Die MontiCore Workbench benutzt eine erweiterte Version der Extended Backus–Naur Form (EBNF)-Grammatik, um eine DSL zu spezifizieren. MontiCore leitet anhand der Grammatik sowohl den Parser als auch die Klassen des korrespondierenden Metamodells ab, die dann vom Parser zum einen AST instanziiert werden.

Eine wichtige Eigenschaft der MontiCore-Grammatik ist es, dass die Grammatik nicht nur das Aussehen der konkreten Syntax definiert, sondern sie kann auch die Struktur des Metamodells kontrollieren. Dabei können in der Grammatik spezielle Konstrukte benutzt werden, um einen direkten Einfluss auf die Namen von Attributen, Methoden und Vererbungsbeziehungen der Klassen des Metamodells zu nehmen. Im Weiteren werden diese Konstrukte exemplarisch an ausgewählten Grammatikabschnitten der NESTML-Grammatik erklärt.

MCG

```
1 package org.nest;
2
3 /**
4  Grammar defining the structure of neurons and components
5  */
6 grammar NESTML extends org.nest.Procedural, org.nest.Equations {
7   // Elements of the NESTML grammar
8 }
```

Abbildung 4.2: Auszug der NESTML-Grammatik, die die Struktur eines Neurons bzw. einer Komponente definiert.

Abbildung 4.2 enthält eine exemplarische MontiCore-Grammatikdeklaration. Die MontiCore Workbench ermöglicht es, Grammatiken hierarchisch zu strukturieren, indem sie in einem Paket hinterlegt werden. Dafür wird die Grammatikdefinition um eine `package`-Deklaration ergänzt (vgl. Zeile 1).

Die Grammatikdefinition beginnt mit dem Schlüsselwort `grammar` gefolgt von einem Namen. In Anlehnung an die Namenskonvention in Java beginnt der Grammatikname stets mit einem Großbuchstaben. Mit dem Schlüsselwort `extends` gefolgt vom Namen

wird das Konzept der Grammatikvererbung unterstützt, das in Abschnitt 4.4 genauer vorgestellt wird. Die Grammatikvererbung erlaubt es, Teile anderer Grammatiken wiederzuverwenden oder zu erweitern. Alle Grammatikproduktionen werden innerhalb dieser benannten Grammatikdefintion deklariert.

Innerhalb der Grammatikdefinition unterscheidet das MontiCore-Grammatikformat zwischen Lexer-Produktionen und Parser-Produktionen. Eine Lexer-Produktion startet mit dem Schlüsselwort *token*. Sie besteht aus einem Namen und einem Rumpf, der durch einen regulären Ausdruck definiert ist. Um die Grammatik zu dokumentieren, ist es möglich, Kommentare im Java-Stil zu verwenden.

```
1 token Name = ( 'a'..'z' | 'A'..'Z' | '_' | '$' )                    MCG
2              ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' )*;
```

Abbildung 4.3: Eine Lexer-Produktion in MontiCore-Syntax, die einen `Name` definiert.

Abbildung 4.3 demonstriert eine Lexer-Produktion `Name`, mit der valide Bezeichner von Neuronen, Variablen und Methoden definiert werden. Im Unterschied zu Parser-Produktionen werden die Lexer-Produktionen von einem *Lexer* erkannt. Ein Lexer liefert eine Sequenz von Tokens für die weitere syntaktische Analyse. Weiterführenden Informationen zur lexikalischen Analyse finden sich in [ASU86].

Im Beispiel aus Abbildung 4.3 werden Alternativen durch einen `|`-Operator modelliert. Zusammenhängende Wertebereiche können durch den `..`-Operator spezifiziert werden. Beispielsweise entspricht der Term `a..z` einem beliebigen Kleinbuchstaben des lateinischen Alphabets. Kardinalitäten können benutzt werden, um die Häufigkeit eines Terms zu spezifizieren. Die unterstützten Kardinalitäten sind: `*`, `+`, `?`. Dabei steht die Stern-Kardinalität (`*`) für beliebig viele Vorkommen eines Terms. Die Plus-Kardinalität (`+`) steht für beliebig viele, aber mindestens einem Vorkommen eines Terms. Die `?`-Kardinalität steht für genau ein oder kein Vorkommen eines Terms. Zudem können runde Klammern zur Strukturierung der Produktion verwendet werden.

Man kann bereits an dieser einfachen Regel sehen, dass die regulären Ausdrücke in Lexer-Produktionen durchaus sehr komplex werden können. Dennoch unterscheiden sich diese Produktionen in unterschiedlichen Sprachen kaum voneinander. Beispielsweise kann die `Name`-Produktion genauso in den Programmiersprachen Java und C++ zur Anwendung kommen. Daher stellt MontiCore eine Menge vordefinierter Grammatiken mit gängigen lexikalischen Produktionen für beispielsweise Namen, Zeichenketten und Zahlen bereit. Diese Regeln werden in NESTML durch die Erweiterung der existierenden `Types`-Grammatik[3] wiederverwendet.

Eine Parser-Produktion besteht aus einem Regelkopf und dem Regelrumpf. Da die MontiCore Workbench auf kontextfreien Grammatiken basiert, enthält der Regelkopf eine Deklaration eines Nichtterminals. Das Nichtterminal ist durch den Regelrumpf auf der

[3] https://github.com/MontiCore/monticore/tree/master/monticore-grammar/

```
1 Neuron = "neuron" Name ":"                                    MCG
2            (BodyElement)*
3          "end";
```

Abbildung 4.4: Eine Parserproduktion in MontiCore-Syntax, mit der die Definition eines Neurons modelliert wird.

rechten Regelseite definiert. Der Rumpf besteht aus einer Folge von Symbolen, die entweder Zeichenketten, Tokens oder andere Parser-Produktionen sind. Dabei können diese Elemente durch runde Klammern, Alternativen und Kardinalitäten verknüpft werden.

Abbildung 4.4 zeigt eine Parser-Produktion für die Deklaration eines Neurons. Jede Definition eines Neurons beginnt mit dem Schlüsselwort `neuron`, das ein Schlüsselwort der konkreten Syntax von NESTML ist. Danach folgt ein Name mit einem Doppelpunktzeichen (`:`) am Ende. Die Neuronendefinition endet mit dem Schlüsselwort `end`. Innerhalb der Neurondefinition können beliebig viele Nichtterminale `BodyElement` vorkommen.

Auf Basis der Grammatik wird mithilfe von ANTLR [Par13, Par17] ein Parser generiert. Zusätzlich zum Parser erstellt MontiCore Klassen des Metamodells. Diese Klassen werden dabei nach einem bestimmten Schema erzeugt.

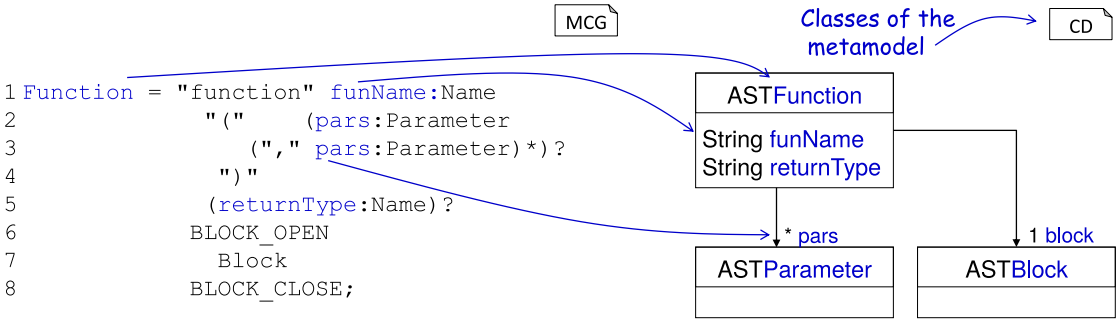


Abbildung 4.5: Parser-Produktion, die die Möglichkeit zur Spezifikation von Listen und expliziten Attributnamen im generierten Metamodell demonstriert. Auf der rechten Seite werden die abgeleiteten Klassen des Metamodells und deren schematische Ableitung aus der Grammatikproduktion dargestellt.

Für jede Parser-Produktion in der Grammatik wird eine Klasse im Metamodell erstellt. Die generierten Klassen erhalten dabei die Namen der Nichtterminale mit vorangestelltem Präfix `AST` (vgl. `ASTFunction`, `ASTParameter` und `ASTBlock` in Abbildung 4.5). Die Attribute dieser Klasse werden auf Basis der rechten Seite der Parser-Produktion generiert. Aufgrund des einfachen Vorkommens eines Nichtterminals wird ein Attribut in der AST-Klasse erstellt. Zum Beispiel wird das `block`-Attribut in der `ASTFunction`-Klasse deswegen erzeugt. Für die Kardinalitäten $+$ und $*$ werden außerdem Listen generiert (vgl. `pars`). Für jede `Interface`-Produktion wird ein Interface generiert. Für die Lexer-

Produktion kann der Typ präzisiert werden. So werden beispielsweise die `Name`-Tokens direkt auf den Stringtyp abgebildet (vgl. `funName` und `returnType`).

Darüber hinaus bietet MontiCore die Möglichkeit, einen Namen vor einem Nichtterminal im Regelrumpf zu spezifizieren. Dies führt dazu, dass innerhalb des Metamodells an dieser Stelle ebenfalls eine Variable mit demselben Namen generiert wird. Abbildung 4.5 demonstriert dies am Beispiel einer Produktion, mit der die Funktionsdefinition modelliert wird. Dabei wird das generierte Metamodell so modifiziert, dass die `ASTFunction`-Klasse zwei Attribute `funName` und `returnTyp` erhält. Alle Funktionsparameter werden zu einer Liste `pars` zusammengefasst. Dies wird durch die Angabe des gleichen Namens bei jedem Vorkommen der `Parameter`-Produktion im Regelrumpf erreicht.

Die `BodyElement`-Produktion aus Abbildung 4.4 spielt eine besondere Rolle innerhalb der NESTML-Grammatik. Die MontiCore Workbench erlaubt die Definition von *Interface*-Nichtterminalen. Dabei können Interface-Nichtterminale wie normale Parser- bzw. Lexer-Produktionen innerhalb NESTML-Grammatik verwendet werden. Eine Produktion kann das Interface implementieren. Somit können die implementierenden Produktionen überall dort verwendet werden, wo auch das implementierte Interface zulässig war.

```
1 interface BodyElement;
2
3 Dynamics implements BodyElement =
4     "update" ":"
5         Block
6     "end";
7
8 Equations implements BodyElement =
9     "equations" ":"
10        OdeDeclaration
11    "end";
```

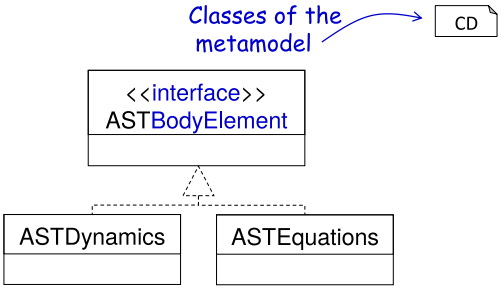


Abbildung 4.6: Definition des `Interface`-Nichtterminals `BodyElement` und dessen Implementierungen durch die `Dynamics`- und `Equations`-Produktionen.

Abbildung 4.6 zeigt die Definition des `BodyElement`-Interfaces. Zudem zeigt dieses Beispiel noch zwei weitere Parser-Produktionen, die dieses *Interface* implementieren: die `Dynamics`-Produktion und die `Equations`-Produktion. Auf der AST-Ebene führt das dazu, dass die entsprechenden AST-Klassen das Interface `ASTBodyElemente` implementieren.

Der wesentliche Vorteil der Benutzung einer Interface-Produktion besteht darin, dass die NESTML-Grammatik zu einem späteren Zeitpunkt an dieser Stelle erweitert werden kann, sodass zusätzliche Modellierungselemente innerhalb des Neuronenrumpfes vorkommen können. Dafür muss nur das Interface durch Produktionen aus Subsprachen implementiert werden. Alle Sprachverarbeitungswerkzeuge, die mit dem `BodyElement`-

Interface arbeiten, können in diesem Fall unverändert weiterverwendet werden.

4.4 Wiederverwendung der Grammatikdefinition

Die zuvor vorgestellten Konzepte in MontiCore beziehen sich fast ausschließlich auf die Definition einer einzelnen Grammatik und die Modellierung einer einzelnen DSL. In MontiCore existieren zuätzlich Konstrukte, die die Sprachkomposition auf unterschiedliche Arten ermöglichen. Dem Entwickler stehen unterschiedliche Konzepte für die Sprachintegration [Völ11] zur Verfügung. Die einzelnen Sprachdefinitionen bleiben dadurch kompakt und modular. Mithilfe der Sprachintegration können Subsprachen in eine homogene Gesamtsprache integriert werden. Dies verbessert die Wiederverwendbarkeit der Sprachgrammatiken und senkt die Gesamtkomplexität der einzelnen Grammatiken. Details zu den Integrationsmechanismen sind in [Völ11, LNPR+13, HLMSN+15] zu finden.

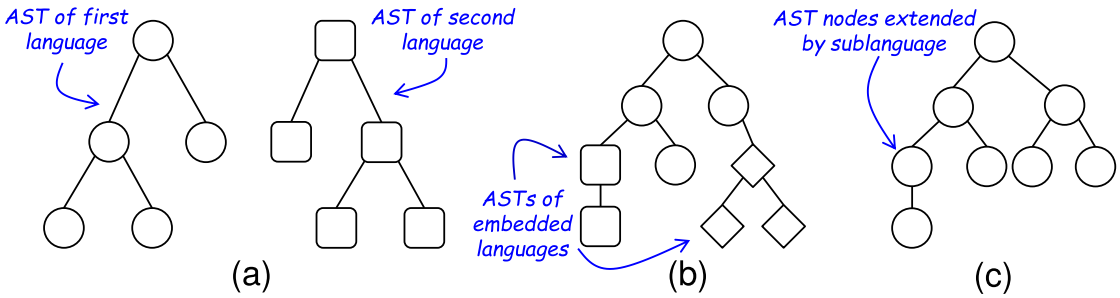


Abbildung 4.7: Schematische Darstellung der unterschiedlichen Sprachwiederverwendungsmechanismen [HLMSN+15, LNPR+13, Loo17]. Dabei entspricht (a) der Sprachaggregation (b) der Einbettung (c) der Sprachvererbung

Abbildung 4.7 fasst diese Mechanismen zusammen. Die erste Möglichkeit ist die Sprachaggregation, die Referenzen zwischen Modellen verschiedener Sprachen erlaubt (vgl. Abbildung 4.7 (a)). Bei der Sprachaggregation werden aus unterschiedlichen Modellen unabhängige ASTs instanziiert. Die Verbindung zwischen Modellelementen ist über Namensreferenzen angegeben.

Die zweite Möglichkeit ist die Spracheinbettung (vgl. Abbildung 4.7 (b)). Sie ermöglicht es, unterschiedliche Sprachen ineinander einzubetten. Auf der Ebene des Metamodells einsteht ein AST, der gleichzeitig Knoten der einbettenden und der eingebetteten Sprache enthalten kann.

Die letzte Möglichkeit, Sprachen zu integrieren, ist die Sprachvererbung. Diese Möglichkeit kommt bei der Implementierung von NESTML haupsächlich zur Anwendung. Die Sprachvererbung erlaubt es, die Produktionen aus einer übergeordneten Sprache bzw. Grammatik wiederzuverwenden oder zu verfeinern. In Grammatiken wird die Vererbung durch das Schlüsselwort `extends` definiert, wie in Abbildung 4.2 dargestellt. Dadurch

können in der NESTML-Grammatik alle Produktionen verwendet werden, die in den `Procedural-` und `Units-`Grammatiken definiert werden (vgl. Kapitel 6 für den Überblick aller NESTML-Subsprachen). Auf der AST-Ebene führt das dazu, dass Knoten der übergeordneten Sprache im AST der erweiternden Sprache enthalten sein und durch Subklassenbildung erweitert werden können.

4.5 Symboltabelle

Eine weitere Stärke von MontiCore ist die Möglichkeit, die Symboltabelleninfrastruktur [Par09, Völ11, Sch12, MSNRR16] für eine DSL zu erstellen. Die MontiCore Workbench ist dabei in der Lage, Teile der Symboltabellenimplementierung auf der Basis der Grammatik zu generieren. Die restlichen Teile werden durch das Implementieren der vordefinierten Schnittstellen in das Symboltabellenframework integriert. Schließlich stellen generierte Klassen und vorgeschriebene Schnittstellen wohldefinierte Anbindungspunkte für handgeschriebenen Code dar.

Die Symboltabelle bildet unter anderem die Grundlage zur Überprüfung der Kontextbedingungen. Da die MontiCore Workbench auf einem kontextfreien Grammatikformat basiert, ist die Ausdrucksstärke der resultierenden Parser eingeschränkt [ASU86], sodass die Kontextbedingungen erst nach dem Parsen geprüft werden können. Zum Beispiel wäre es mithilfe eines kontextfreien Parsers nicht möglich, zu prüfen, ob ein in einem Ausdruck verwendeter Bezeichner im Modell auch definiert ist. Diese Prüfung setzt einen kontextsensitiven Parser voraus. In MontiCore wird dafür die Symboltabelle benutzt, die erst nach dem Parsen des Modells aufgebaut wird.

Definition 4.2. *(Symboltabelle). Eine Symboltabelle ist eine Datenstruktur zum Speichern und Auflösen von Bezeichnern in einer Sprache. Kernaufgabe besteht im Auflösen von Namen mit dem Ziel, weitere Informationen wie Typ oder Signatur zu diesem Namen zu erhalten [Völ11].*

Die Bezeichner werden in MontiCore durch Symbole modelliert, die Instanzen eines `Symbol`-Interfaces sind. Symbole erlauben eine schnelle und transparente Navigation zwischen Benutzung und Deklaration eines Bezeichners anhand des Namens. Ein Symbol ist genau einmal definiert. Die Definition eines Symbols in MontiCore lautet wie folgt:

Definition 4.3. *(Symbol) Eine Symboldefinition oder ein Symbol enthält alle wesentlichen Informationen über ein benanntes Modellelement. Es hat eine bestimmte Art (SymbolKind) abhängig von dem Modellelement, das es bezeichnet [MSN17].*

Das Beispiel in Abbildung 4.8 modelliert ein Symbol, das die Informationen über eine Methode speichert, die in einem Neuron definiert ist. Dieses Symbol ist mit dem definierenden Knoten `ASTFunction` über die vererbte Assoziation verbunden. Des Weiteren ist das `MethodSymbol` mit einem `MethodKind` verbunden.

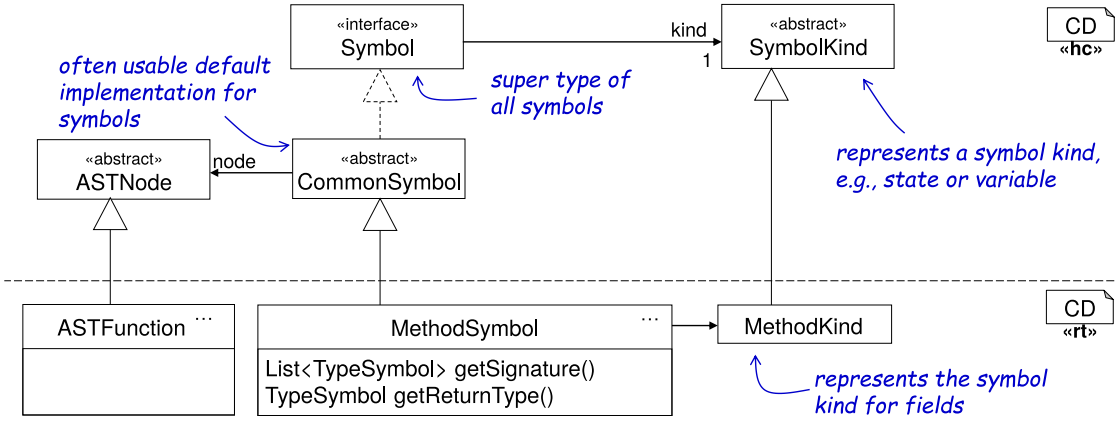


Abbildung 4.8: Ein Ausschnitt der Symbolinfrastruktur von NESTML.

Im Falle einer Methode speichert das Symbol die essentielle Information über das Modellelement wie seinen Namen, die Parameter und den Rückgabewert. In der Sprachimplementierung werden Symbole durch Erweiterung der abstrakten Klasse `CommonSymbol` definiert. Jedes Symbol verfügt über ein spezielles Attribut vom `SymbolKind`-Typ, das abhängig vom modellierten Modellelement definiert wird. Es ist zu beachten, dass unterschiedliche Symbole dieselbe `SymbolKind`-Klasse teilen können. Wenn unterschiedliche Symbolspezialisierungen dieselbe `SymbolKind`-Klasse teilen, werden diese von MontiCore als gleicher Symboltyp behandelt. Somit wird die transparente Sprachkomposition unterstützt, da MontiCore nicht unterscheidet, aus welcher Subsprache das Symbol stammt.

Typischerweise stellen Symbole Komfortfunktionen zur Verfügung. Die Methode `getSignature` aus der `MethodSymbol`-Klasse liefert eine Liste mit Symbolen, die anhand der textuellen Darstellung des jeweiligen Typs aus dem Neuronenmodell bestimmt wurden. Die Methode `getReturnType` löst den Rückgabetyp auf und liefert das entsprechende Symbol an den Aufrufer zurück.

Der Zugriff auf ein Symbol innerhalb eines Modells kann nicht von überall erfolgen. Die Sichtbarkeit eines Symbols könnte durch sprachspezifische Regeln eingeschränkt werden. In den Programmiersprachen Java und C++ stehen lokale Variablen einer Methode z.B. nur innerhalb der Methode selbst zur Verfügung. Des Weiteren können Symbole durch gleichnamige Symbole in hierarchisch geschachtelten Modellen verdeckt werden. In der Programmiersprache Java ist es beispielsweise möglich, dass eine lokale Variable in einer Methode eine Klassenvariable verdeckt.

Somit ergibt sich die Definition der Sichtbarkeit eines Symbols wie folgt:

Definition 4.4. *(Sichtbarkeit) Die Sichtbarkeit eines Symbols ist die logische Region, in der das Symbol durch seinen (einfachen) Namen potenziell zugänglich ist [MSN17].*

In NESTML kann eine Variable, die zum Beispiel im `state`-Block definiert wird, durch

eine gleichnamige Variable im `update`-Block verdeckt werden. Abbildung 4.9 demonstriert dieser Art der Verdeckung (engl:shadowing) an einem konkreten Beispiel. Die Variable `V_m` ist im `state`-Block in der Zeile 3 definiert. In Zeile 6 wird die ursprüngliche Definition durch die neue Definition verdeckt. In Zeile 9 wird aber die Variable `V_m` aus Zeile 3 benutzt.

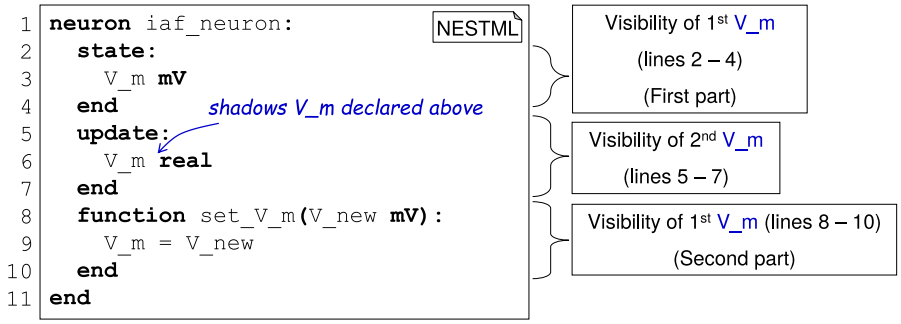


Abbildung 4.9: Beispiel der Verdeckung einer Variable im `update`-Block, hier `V_m`.

Um die Sichtbarkeitsregeln transparent zu verwalten, verwendet die MontiCore Workbench eine hierarchische *Scope*-Datenstruktur. Ein konkreter Scope kann als Symbolcontainer angesehen werden, der nach dem Composite-Muster [GHJV93] aufgebaut ist. Ein Scope enthält beliebig viele eingebettete Scopes, die ihrerseits weitere Scopes einbetten können.

Es ergibt sich folgende Definition eines Scope:

Definition 4.5. *(Scope) Scope ist eine logische Gruppierung der Symboldefinitionen mit den zugehörigen Sichtbarkeitsregeln.*

Abbildung 4.10 illustriert diesen Ansatz am Ausschnitt der entsprechenden NESTML-Umsetzung. Der `GlobalScope` verwaltet alle Scopes und spielt eine wesentliche Rolle beim modellübergreifenden Auflösen der Symbole. Um dies zu ermöglichen, werden `ArtifactScope`s erzeugt, die unterschiedliche Modelldateien repräsentieren. Anhand vollqualifizierter Namen kann der `GlobalScope` den richtigen `ArtifactScope` identifizieren und Anfragen an diesen Scope weiterleiten.

Eine weitere Aufgabe des `GlobalScope`s ist die Verwaltung der vordefinierten Datentypen (z.B. `integer`), Konstanten (z.B. `t`) und Methoden (z.B. `emit_spike`). Diese Symbole werden bei der Erstellung im `GlobalScope` registriert und stehen allen NESTML-Modellen stets zur Verfügung.

Jede NESTML-Datei wird als eine Instanz des `ArtifactScope` behandelt. Der Sprachentwickler kann weitere Scopes bei Bedarf definieren. Im vorliegenden Beispiel stellt `NeuronScope` einen benutzerdefinierten Scope dar. Der Zweck dieses Scopes ist es, beim Auflösen der Symbole Anfragen an andere Artefakte weiterzuleiten, falls das anfragende Neuron eine *Import*-Beziehung zu anderen NESTML-Modellen hat.

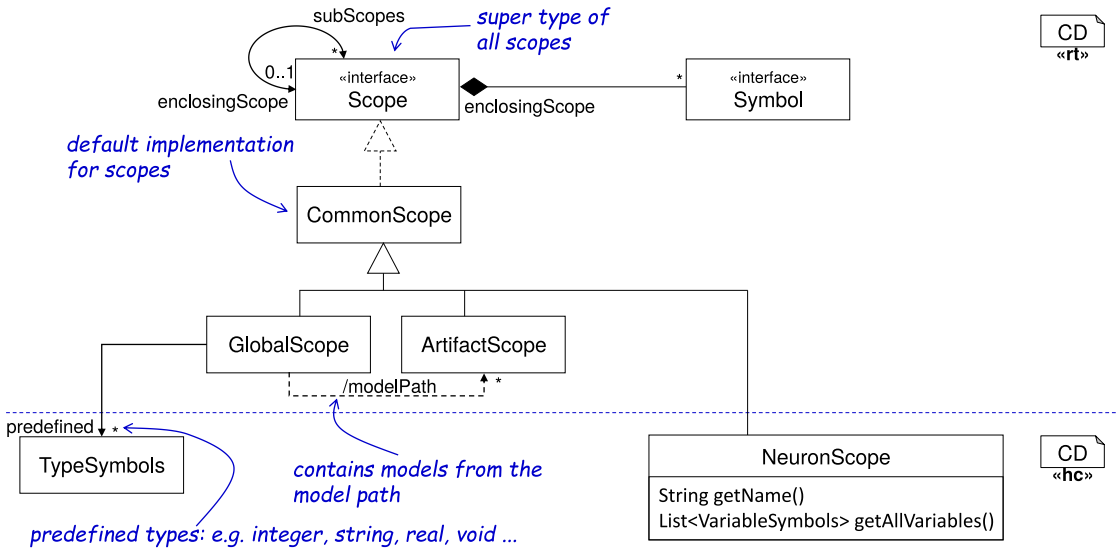


Abbildung 4.10: Ein Ausschnitt der NESTML-spezifischen Umsetzung der Scope-Hierarchie in der MontiCore Workbench.

Den Aufbau der Scope-Hierarchie führt die MontiCore Workbench semi-automatisiert durch. Dies geschieht durch das Implementieren eines vordefinierten Interfaces `ScopeSpanningSymbol` in den sprachspezifischen Symbolen. In diesem Fall erzeugt die MontiCore Workbench die passende Hierarchie der Scopes automatisch, sobald das entsprechende Symbol erzeugt und in der Symboltabelle registriert wird.

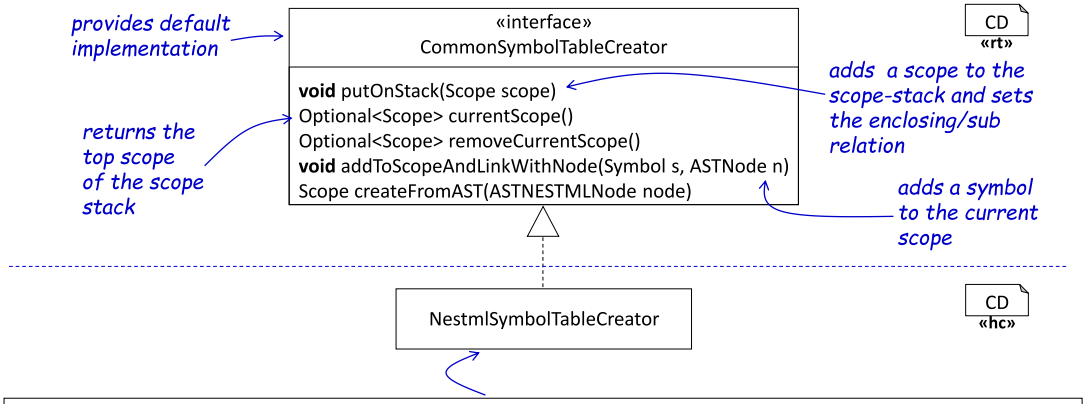
Eine wichtige Rolle beim Erzeugen der Symbole und der Scope-Hierarchie spielt der `SymbolTableCreator`. Der `SymbolTableCreator` ist ein vordefiniertes Interface, das sprachspezifisch implementiert wird und für jede Sprache in der Symboltabelleninfrastruktur registriert wird.

Um die Scope-Hierarchie korrekt aufzubauen, verwaltet MontiCore intern eine Stack-Datenstruktur [CSRL01]. Abbildung 4.11 visualisiert eine beispielhafte Implementierung des `NestmlSymbolTableCreators`. Dabei stehen der implementierenden Klasse einige Komfortfunktionen als Teil des Interfaces `SymbolTableCreator` zur Verfügung, um die Scope-Hierarchie zu verwalten:

`createFromAST`: Diese Methode wird benutzt, um die Erstellung der gesamten Symboltabelle zu initiieren. Dabei muss das geparste Modell der Methode als ein Argument übergeben werden.

`putOnStack`: Diese Methode wird benutzt, um einen neuen Scope auf dem Scope-Stack abzulegen.

`removeScope`: Diese Methode ist das Gegenstück der `putOnStack`-Methode. Mit dem




```
1  public class NestmlSymboltableCreator implements SymbolTableCreator {
2    Scope createFromAST(final ASTNESTMLNode rootNode) {
3      rootNode.accept(this);
4      return getFirstCreatedScope();
5    }
6    // ...
7    public void visit(final ASTNeuron ast) {
8      final NeuronSymbol neuronSymbol = new NeuronSymbol(astNeuron.getName(), NEURON);
9      addToScopeAndLinkWithNode(neuronSymbol, astNeuron);
10   }
11 }
```

Java «hc»

Abbildung 4.11: Ein Auszug des sprachspezifischen `SymbolTableCreators` der NESTML-Sprache.

Aufruf dieser Methode wird der aktuelle Scope aus dem Stack entfernt.

`currentScope:` Diese Methode wird benutzt, um beim Aufbau der Symboltablle auf den alktuellen Scope zuzugreifen.

`putInScopeAndLinkWithAst:` Diese Methode legt ein Symbol im aktuellen Scope ab und assoziiert das Symbol mit einem AST-Knoten aus dem Quellmodell. In Abbildung 4.11 wird beispielsweise das `NeuronSymbol` mit dem `ASTNeuron`-Knoten verbunden (vgl. Zeile 9).

Auf diese Weise verbindet der `NestmlSymbolTableCreator` die Erzeugung der Symbole aus NESTML-Modellen mit der Verwaltung der Sichtbarkeitsregeln dieser Symbole.

Die Symboltabelle wird auch zur Modellkomposition verwendet. Sie ermöglicht das aus der komponentenbasierten Softwareentwicklung bekannte Konzept der Modularität. Jedes Modell wird dabei als ein eigenständiges und abgeschlossenes Modul angesehen. Modelle können, solange sich die Schnittstelle nicht ändert, verändert und gleichzeitig von anderen Modellen ohne Anpassung weiter benutzt werden.

4.6 Visitoren zum Traversieren der ASTs

Wie zuvor beschrieben, werden Instanzen der MontiCore-Sprachen als komplexe Baumstrukturen (ASTs) instanziiert. Das Rückgrat aller Algorithmen, die auf dem AST ausgeführt werden, bildet der Visitor-Ansatz [HMSNRW16], der durch Anwendung des Double-Dispatch Musters [Vli04], eine transparente Trennung dieser Datenstrukturen von den darauf laufenden Berechnungen ermöglicht.

Ursprünglich wurde dieses Muster in [GHJV93] vorgestellt und liegt in MontiCore in einer erweiterter Form vor. Der Name des Musters resultiert aus seiner grundlegenden Funktionsweise. Dabei *besucht* (engl: visits) der Visitor alle Konten des ASTs und ruft dabei jeweils die Methode `visit` auf.

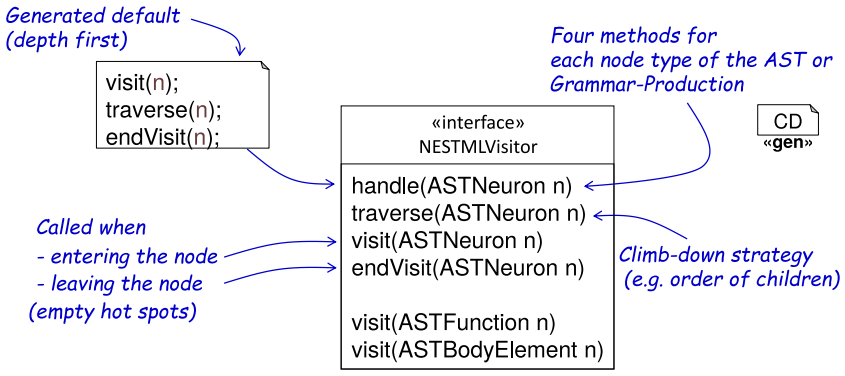


Abbildung 4.12: Ein Ausschnitt des Visitors, der auf der Grundlage der NESTML-Grammatik generiert wird.

Die MontiCore Workbench generiert dabei die Basisfunktionalität eines Visitors aus jeder Grammatik. Der Ausschnitt des NESTML-Visitors, der aus der NESTML-Grammatik generiert wird, ist in Abbildung 4.12 zu sehen. Für jedes Nichtterminal aus der entsprechenden Grammatik werden jeweils vier Methoden generiert, die in den konkreten Visitoren überschrieben werden können. Die folgende Liste erläutert diese Methoden:

`handle`: Diese Methode kontrolliert die Abstiegsstrategie des Visitors. Falls nicht anders spezifiziert, verwendet der Visitor eine Tiefensuche [CSRL01].

`traverse`: Diese Methode kontrolliert, in welcher Reihenfolge die direkten Nachkommen eines Knotens besucht werden. Normalerweise wird von der MontiCore Workbench keine bestimmte Reihenfolge garantiert. Diese Methode ist dafür zuständig, dass alle Nachkommen eines Knotens besucht werden.

`visit`: Diese Methode wird beim Vorkommen einer Knotens vom entsprechenden Typ im AST aufgerugen. Der Typ wird durch die Signatur der Methode festgelegt.

`endVisit`: Diese Methode wird aufgerufen, nachdem der Knoten selbst und alle seine Nachkommen besucht wurden.

Der Sprachentwickler kann den generierten Visitor benutzen, um eigene Berechnungen und Analysen auf dem AST durchzuführen. Dabei sieht die MontiCore Workbench vor, dass zusätzliche Funktionalität innerhalb der `visit`- bzw. `endVisit`-Methoden implementiert wird. Dafür überschreibt der Sprachentwickler die `visit`-Methoden bzw. `endVisit`-Methoden für die gewünschten Signatur und implementiert darin eigene Logik.

4.7 Kontextbedingungen

Wie bereits bei der Vorstellung der MontiCore-Grammatik erläutert, baut die Grammatik auf den kontextfreien Grammatiken auf. Das führt dazu, dass kontextabhängige Bedingungen durch den MontiCore-Parser nicht geprüft werden können. Aber selbst Bedingungen, die durch einen kontextfreien Parser theoretisch erkannt werden könnten, sind in einer kontextsensitiven Weise oft einfacher zu formulieren. Um diesem Nachteil zu entgegnen, bietet die MontiCore Workbench ein integriertes Framework zur Definition von Kontextbedingungen an, die sowohl den Modell-AST als auch die Symboltabelle verwenden [Sch12, Völ11].

Dabei werden Kontextbedingungen in unterschiedliche Kategorien eingeordnet. Martin Schindler [Sch12] unterscheidet zwischen den folgenden Kategorien von Kontextbedingungen:

Sprachinterne Intra-Modell-Bedingungen sind Bedingungen, die innerhalb eines Modells geprüft werden.

Sprachinterne Inter-Modell-Bedingungen sind Bedingungen, die zwischen mehreren Modellen derselben Sprache geprüft werden.

Sprachübergreifende Intra-Modell-Bedingungen sind Bedingungen, die innerhalb eines Modells geprüft werden. In diesem Fall werden Teile desselben Modells durch unterschiedliche Sprachen definiert oder Symbole verwendet, die in Modellen anderer Sprachen definiert worden sind.

Sprachübergreifende Inter-Modell-Bedingungen sind Bedingungen, die zwischen mehreren Modellen geprüft werden, wobei diese in Unterschiedlichen Sprachen definiert werden.

Ein weiterer Vorteil der MontiCore Workbench besteht darin, dass der Sprachentwickler sich nicht manuell um diese Unterscheidung kümmern muss. Die Symboltablleninfrastruktur ist in der Lage, die unterschiedlichen Beziehungen zwischen Modellen und

Sprachen automatisch zu erkennen. Bei Bedarf werden die ausstehenden Symbole aus anderen Modellen nachgeladen. Wenn die Symbole aus heterogenen Sprachen stammen, können diese durch passende Adapter korrekt umgewandelt werden.

Die Kontextprüfungen sind technisch gesehen eine spezielle Form des Visitor-Musters (vgl. Abschnitt 4.6). Für jedes Nichtterminal aus der Grammatik generiert die MontiCore Workbench ein Interface, das genau eine `check`-Methode enthält. Die konkrete Kontextbedingung implementiert dieses Interface und überschreibt die `check`-Methode mit der passenden Signatur.

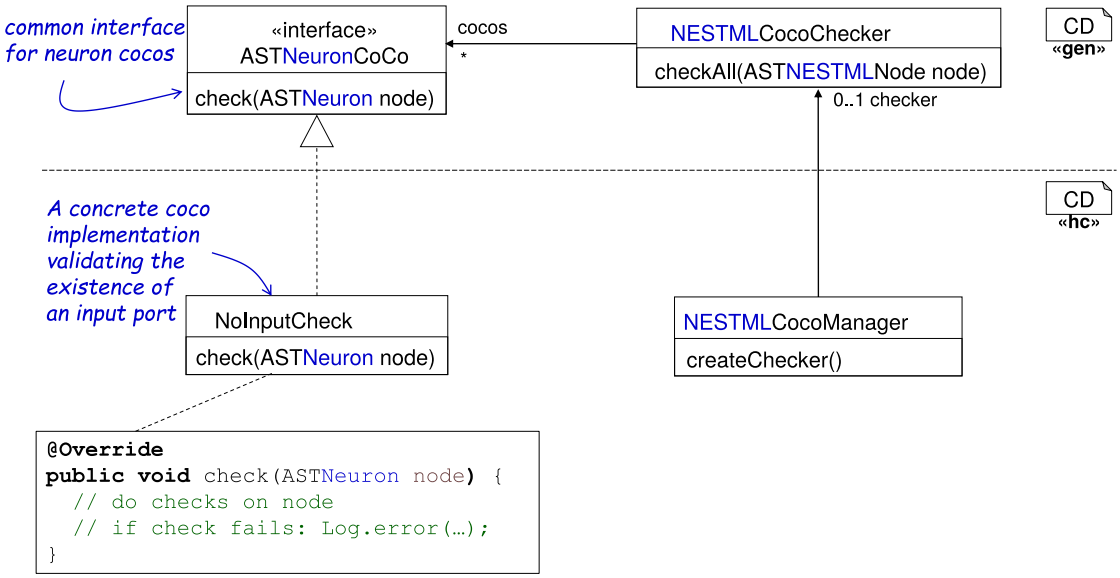


Abbildung 4.13: Sprachspezifische Implementierung der Registrierung von Kontextbedingungen.

Abbildung 4.13 demonstriert eine exemplarische Implementierung einer NESTML-Kontextbedingung. Sie prüft, dass jedes Neuron stets mindestens einen eingehenden Port besitzt (vgl. Abschnitt 7.3). Um diese Prüfung umzusetzen, implementiert die Klasse `NoInputCheck` das `ASTNeuronCoco`-Interface und überschreibt die `check`-Methode mit dem `ASTNeuron`-Parameter in der Methodensignatur. Da der AST-Knoten zur Verfügung steht, kann die Kontextbedingung einfach die Anzahl der Ports in der abstrakten Syntax zählen. Wenn deren Anzahl gleich 0 ist, wird ein Fehlereintrag mithilfe der `Log`-Methode erstellt. Bei diesem Fehler wäre es möglich, ihn bereits beim Aufbau des AST durch den Parser zu identifizieren. Dadurch, dass eine Kontextbedingung allerdings den kompletten AST sowie die Symboltabelle zur Verfügung hat, kann sie weitaus informativere Fehlernachrichten erstellen. Daher sollten möglichst viele Fehlerabfragen auf der Ebene der Kontextbedingungen umgesetzt werden und nicht in der Grammatik bzw. dem Parser spezifiziert werden.

Für die Verwaltung der Kontextbedingungen einer Sprache schreibt MontiCore eine Methodik für die Organisation der Kontextbedingungen vor. Für jede Sprache wird eine Manager-Klasse erstellt (vgl. `NESTMLCocoManager` in Abbildung 4.13). Der Manager ist für die Instanziierung aller Kontextbedingungen und deren Registrierung beim sprachspezifisch generierten Prüfer zuständig (vgl. `NESTMLCococChecker`). Schließlich kann der `NESTMLCocoChecker` alle bei ihm registrierten Kontextbedingungen auf einem NESTML-AST prüfen.

4.8 Codegenerierung

Nachdem alle Elemente zum Einlesen und Validieren der Modelle erläutert wurden, stellt dieser Abschnitt die von MontiCore angebotene Codegenerierungsinfrastruktur vor und legt die Grundlage für ein genaueres Verständnis des in der Arbeit entwickelten NEST-Generators.

Die MontiCore Workbench verwendet das *Freemarker*-Framework[4] [GBR04] zur Codegenerierung. MontiCore erweitert Freemarker um eine transparente Anbindung an Modell-ASTs und deren Symboltabelle. Freemarker stellt ein Model-to-Text (M2T)-Framework dar, das einen AST in einen Text transformiert. Die MontiCore Workbench unterstützt auch Visitor-basierte Model-to-Model (M2M)-Ansätze. Eine ausführliche Übersicht der möglichen Transformationsansätze mit deren Vor- und Nachteilen findet sich in [CH06b]. NESTML benutzt eine Kombination aus templatebasiertem M2T-Ansatz und einer Reihe von M2M-Transformationen, die Vereinfachungen und Optimierungen von NESTML-Modellen durchführen.

Im Rahmen dieser Arbeit wird ein Codegenerator entwickelt, der ausführbaren Python- und C++-Quellcode aus NESTML-Modellen generiert. Um die Codegenerierung zu starten, wird ein ausgezeichnetes Starttemplate mit einem initialen AST-Knoten initiiert. Innerhalb dieses Templates können weitere Subtemplates eingebunden werden, die Teile des Zielsystems generieren.

Freemarker selbst bietet eine eingebaute Kontrollsprache, die verschiedene Direktiven für die Ausführungskontrolle beinhaltet. So können einfache Schleifenoperationen, Bedingungen und Zeichenkettenoperation direkt als *Freemarker*-Direktiven umgesetzt werden. Freemarker ermöglicht es, auf Java-Objekte zuzugreifen und Methoden dieser Objekte auszuführen.

Das Codegenerierungsframework stellt ein ausgezeichnetes Objekt `tc` zur Verfügung, das innerhalb von Generierungstemplates benutzt werden kann. Dabei handelt es sich um eine Instanz der `TemplateController`-Klasse. Mithilfe dieses Objekts können Templates eingebunden werden oder Variablen instanziiert werden, die dem Datentausch zwischen Templates dienen können.

[4] `http://freemarker.org/`

Templates, die in MontiCore zur Codegenerierung verwendet werden, können mit Signaturen angereichert werden, um sie möglichst wiederverwendbar zu machen. Eine Signatur wird am Anfang des Templates deklariert. Die Signatur eines Templates definiert, welche Parameter beim Einbinden des Templates übergeben werden müssen. Die MontiCore Workbench kontrolliert dabei, dass alle geforderten Parameter beim Aufruf des Templates übergeben wurden. Dies ermöglicht es, Templates von der Struktur eines spezifischen ASTs zu entkoppeln und in unterschiedlichen Kontexten wiederzuverwenden.

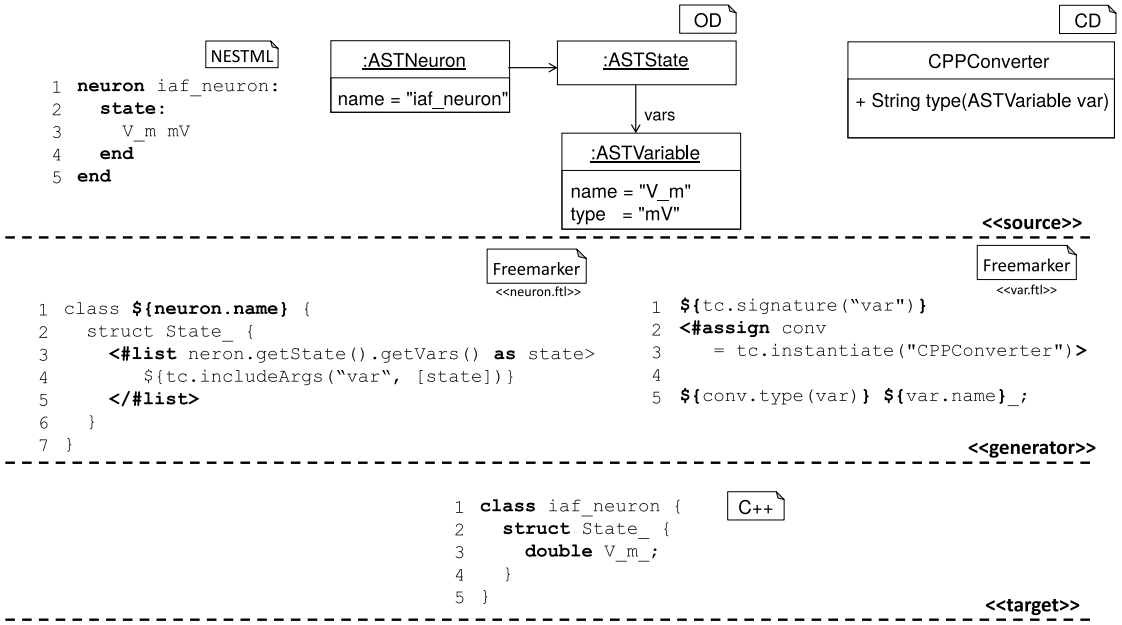


Abbildung 4.14: Verschiedene Codegenerierungsphasen, um aus einem Modell-AST mithilfe der Generierungstemplates eine C++-Implementierung zu generieren.

Abbildung 4.14 erläutert die vorgestellte Funktionalität von MontiCore an einem Beispiel. Die `source`-Schicht erzeugt lediglich den AST, der als Haupteingabe für die Codegenerierung dient. Die Generatortemplates können innerhalb eines `${...}`-Ausdrucks direkt auf die Attribute des AST zugreifen. Im vorliegenden Beispiel wird der `${neuron.name}`-Ausdruck durch den im `ASTNeuron`-Objekt gespeicherten Wert des Attributs ersetzt. Ein weiterer Teil der `source`-Schicht ist die Runtime-Klasse `CPPConverter`.

Die `generator`-Schicht enthält zwei Generierungstemplates, die mithilfe der Runtime-Klassen eine C++-Implementierung generieren. Dabei werden Ausdrücke, wie `${neuron.name}` in Zeile 1, durch den im jeweiligen Attribut gespeicherten Wert ersetzt. Im vorliegenden Beispiel wird der Ausdruck `${neuron.name}` durch den String `"iaf_neuron"` ersetzt. Die `#list`-Anweisung iteriert durch eine Liste aus Einträgen `{vm, thres}`,

die in einem Listenobjekt `vars` gespeichert sind. Zeile 4 bindet für alle Elemente der Liste `vars` den Text ein, der durch ein weiteres Template `var.ftl` erzeugt wird. Das Template `var.ftl` verwendet eine Signatur. Diese legt fest, dass beim Aufruf des Templates ein Parameter übergeben werden muss.

Um ein Template einzubinden, wird die Methode `includeArgs` des `TemplateControllers` benutzt. Als einziges Argument wird ihr der AST-Knoten übergeben, der die NESTML-Variable speichert. In den Zeilen 2-3 wird eine Instanz der Runtime-Klasse `CPPConverter` erzeugt. Der *Converter* ist dafür zuständig, NESTML-Typen (z.B. `real` oder `mV`) in passende C++-Datentypen zu konvertieren. Mit der `assign`-Anweisung kann diese Instanz an einen Variablennamen gebunden werden. In diesem Fall wird sie einer Variable *conv* zugeordnet. Anschließend wird in Zeile 5 diese Variable benutzt, um einen NESTML-Typ in einen C++-Typ zu transformieren.

Die `target`-Schicht enthält den generierten Code. In diesem Fall ist der Code eine C++-Klasse mit eingebettetem C++-`struct`.

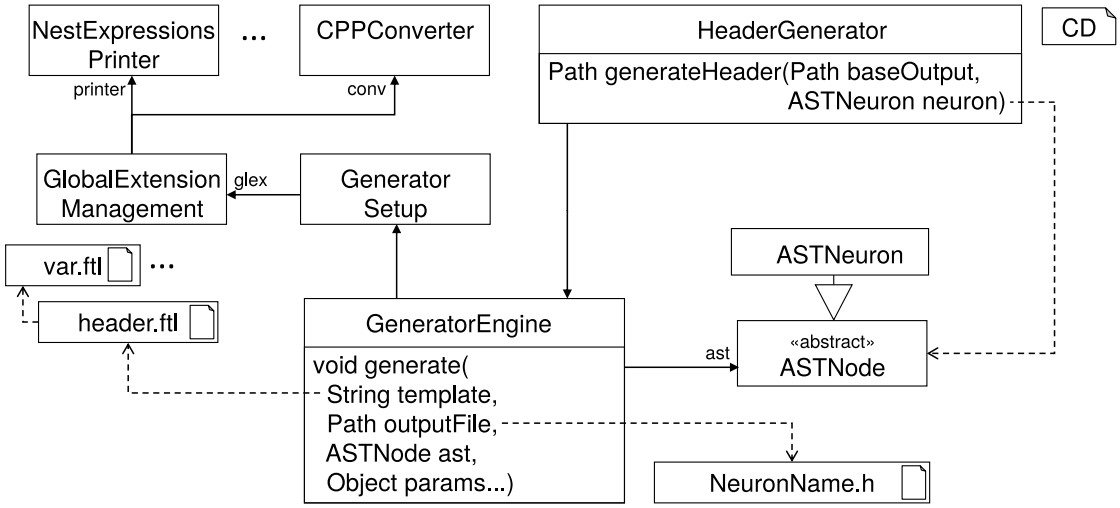


Abbildung 4.15: Ein Ausschnitt aus dem vereinfachten NEST-Codegenerator.

Abbildung 4.15 zwigt das Klassendiagramm eines exemplarischen MontiCore Codegenerators, der für die Generierung eines Neuronenheaders zuständig ist. Die zentrale Klasse ist die `GenerationEngine`. Mit der Methode `generate`, einem Starttemplate, z.B. `header.ftl`, und einem AST-Knoten wird die Generierung einer Zieldatei initiiert. Die technische Information wie beispielsweise das Zielverzeichnis wird in einem Objekt `GeneratorSetup` gespeichert. Die Klasse `GlobalExtensionManagement` verwaltet Abhängigkeiten und Erweiterungen des Generators. Anstatt die Hilfsklassen in dem Template selbst zu instanziieren (vgl. Abbildung 4.15), ist es auch möglich, diese im Objekt der Klasse `GlobalExtensionManagement` zu kapseln. Das entspricht dem *Inversion-of-*

Control-Entwurfsmuster [Fow04]. Auf diese Weise sind die Templates einfacher zu pflegen, zu testen und zu erweitern. Im vorliegenden Fall werden die Objekte `printer` und `conv` als Abhängigkeiten des Generators übergeben. Schließlich ist die Klasse `HeaderGenerator` für die korrekte Instanziierung und Registrierung der Hilfsobjekte in der Klasse `GlobalExtensionManagement` zuständig.

Insgesamt stellt die MontiCore Workbench ein gut integriertes und homogenes Werkzeug zur Codegenerierung dar. Es ermöglicht, Generatoren auf Basis der Symboltabelle und Modell-ASTs zu erstellen und unterstützt den Sprachentwickler auch bei der Erstellung von M2T- und M2M-Generierungskomponenten.

4.9 Zusammenfassung

Dieses Kapitel stellt zunächst domänenspezifische Sprachen als einen Ansatz zur Reduktion der Komplexität der Softwareentwicklung in den Neurowissenschaften vor. Darauf aufbauend werden die Elemente einer DSL erklärt und eine Kosten-Nutzen-Analyse der Verwendung von DSLs im Softwareentwicklungsprozess durchgeführt. Anschließend wird die MontiCore Language Workbench vorgestellt, mit deren Hilfe DSLs agil und kostengünstig erstellt werden können. Das Grammatikformat von MontiCore ermöglicht es, die abstrakte und konkrete Syntax in einem homogenen Artefakt zu definieren und Sprachverarbeitungswerkzeuge aus dieser Grammatik zu generieren. Auch die Möglichkeit der Komposition verschiedener Grammatiken zu einer DSL wird vorgestellt. Darüber hinaus wird die Funktionsweise von Symboltabellen, Visitoren und Kontextbedingungen erklärt. Anschließend werden die Möglichkeiten zur Codegenerierung mithilfe der MontiCore Workbench und Freemarker erläutert.

Die vorgestellten Konzepte zeigen nur einen Auszug der Funktionalität der MontiCore Workbench. Diese sind aber ausreichend, um die Anwendung von MontiCore im Rahmen dieser Arbeit zu verstehen. Weitere Details werden nachfolgend an den Stellen eingeführt, wo sie für das weitere Verständnis notwendig sind.

Im Rahmen dieser Arbeit wird die NESTML-Modellierungssprache erarbeitet, die der Spezifikation von Punktneuronen dienen soll. Diese DSL ist mithilfe der MontiCore Workbench entwickelt und durch geeignete Kontextbedingungen semantisch gesichert. Auf Basis dieser DSL werden dann Neuronenmodelle entwickelt und die Modellierungsmethodik zur Entwicklung neuer Neuronenmodelle definiert (vgl. Kapitel 7). Der Codegenerator für den NEST-Simulator wird auf Basis des hier vorgestellten Codegenerierungansatzes erstellt (vgl. Kapitel 8) und evaluiert (vgl. Kapitel 9).

Kapitel 5

NESTML: eine domänenspezifische Sprache für die Spezifikation von Punktneuronen

Modellierungssprachen spielen in der Neurowissenschaft eine wichtige Rolle für das Teilen und Wiederverwenden von Forschungsergebnissen. In Kapitel 3 wurde allerdings gezeigt, dass die bereits existierten Modellierungssprachen essenzielle Anforderungen an eine Modellierungssprache für den NEST-Simulator nicht im vollen Umfang erfüllen. Da Erweiterungen und Anpassungen dieser Modellierungssprachen aufgrund ihrer technischen Umsetzung nicht möglich sind, wird in diesem Kapitel eine neue domänenspezifische Sprache NESTML [PBI+16] für den NEST-Simulator erarbeitet.

Um die Anforderungen an das Metamodell (vgl. *RQ4*) und die Codegenerierung (vgl. *RQ5*) durch die Wahl der passenden Language Workbench zu erfüllen, wird NESTML und ihre Sprachverarbeitungswerkzeuge mithilfe der MontiCore Workbench entwickelt (vgl. Kapitel 4). Somit wird der sprachliche Unterbau von NESTML modular und erweiterbar aufgebaut. NESTML abstrahiert von der konkreten technischen Realisierung in einem Simulator (vgl. *RQ1*). Diese Sprache enthält eine beschränkte Menge der für die Modellierung von Punktneuronen notwendigen Modellierungselemente mit klarer Semantik (vgl. *RQ2*). Daher bleibt die Sprache leicht erlernbar und portabel. Sie verfügt über eine klare und kompakte Syntax. NESTML unterstützt dennoch eine Vielzahl von biologischen Neuronen, die aufgrund der Portabilität in unterschiedlichen Simulatoren ausgeführt werden könnten.

5.1 Exemplarisches NESTML-Neuron

Die konzeptionelle Arbeitsweise von Punktneuronen, die mithilfe von NESTML spezifiziert werden, wurde bereits in Kapitel 2 erläutert. Die Syntax von NESTML ist durch die der Programmiersprache Python inspiriert. Da diese in den Neurowissenschaften sehr stark verbreitet ist [MBD+09, DDG+13] und Neurowissenschaftlern somit bereits bekannt ist, senkt diese syntaktische Anlehnung die Einstiegshürde für NESTML und erhöht die Verständlichkeit der Modelle für neurowissenschaftliche Anwender.

Im Laufe des Kapitels wird die NESTML-Syntax anhand einiger konkreten Neuronenmodelle vorgestellt. In Abbildung 5.1 wird ein *Integrate-and-Fire*-Neuronenmodell

in der Syntax von NESTML definiert. Das Membranpotenzial des Neurons wird durch die Variable `V_m` modelliert, deren Dynamik mit einer Differenzialgleichung deklarativ beschrieben wird. Zur Laufzeit integriert das Neuron eingehende Ströme so lange, bis die vordefinierte Schranke überschritten ist und das Neuron feuert einen Spike. Nach dem Feuern eines Spikes versetzt sich das Neuron in einen Refraktärzustand, in dem alle eingehenden Ereignisse ignoriert werden.

NESTML

```
1 neuron rc_neuron:
2
3   state: # Captures the dynamic state of the neuron
4     V_m mV
5   end
6
7   equations: # Declarative description of equations
8     shape I_a = (e/tau_syn) * t * exp(-t/tau_syn)
9     function I pA = (I_sum(I_a, spikes) + currents)
10    V_m' = -1/Tau * (V_m - E_L) + 1/C_m * I
11  end
12
13  parameters:  # can be set from outside
14    C_m     pF = 250pF [[C_m > 0]]   # Capacity
15    Tau     ms = 10ms     # Membrane time constant
16    tau_syn ms = 2ms      # Synaptic current
17    t_ref   ms = 2ms      # Refractory period
18    E_L     mV = -70mV    # Resting potential
19    V_reset mV = -70mV - E_L # Reset potential
20    Theta   mV = -55mV - E_L # Spiking threshold
21  end
22
23  internals:  # auxiliary variables
24    ref_coutns integer = steps(t_ref)
25    r  integer
26  end
27  input: # input events
28    spikes   <- spike
29    currents <- current
30  end
31
32  output: spike # output events
33
34  update: # neuron's state update
35    if r == 0: # not refractory
36        integrate_odes()
37        # threshold crossing
38        if V_m >= Theta:
39          r = ref_counts
40          V_m = V_reset
41          emit_spike()
42        end
43    else:
44        r = r - 1
45    end
46
47  end
48
49 end
```

Abbildung 5.1: Ein *Integrate-and-Fire*-Neuronenmodell als NESTML-Modell.

Die Definition eines Neurons beginnt mit dem Schlüsselwort `neuron` gefolgt vom Namen des Neurons. Im vorliegenden Fall wird das Neuron `rc_neuron` spezifiziert (vgl. Zeile 1).

Jedes NESTML-Neuron besteht aus verschiedenen Blöcken. Unter anderem sind dies Blöcke, in denen Variablen definiert werden können wie der `state`-Block (vgl. Zeilen 3-5), der `parameters`-Block (vgl. Zeilen 13-21) und der `internals`-Block (vgl. Zeilen 23-26). Variablendeklarationen innerhalb dieser Blöcke erlauben die Verwendung sowohl von primitiven Datentypen (z.B. `integer` oder `boolean`) als auch physikalischer Einheiten (z.B. Picoamper: `pA`) und zusammengesetzter physikalischer Einheiten (z.B. $\frac{mV}{s}$).

Für Variablen aus dem `state`-Block können zusätzlich Gleichungen im `equations`-Block spezifiziert werden (vgl. Zeilen 7-11). Differenzialgleichungen werden mithilfe von Aufrufen der `integrate_odes`-Methode im `update`-Block schrittweise entwickelt (vgl. Zeile 36).

Variablen, deren Werte sich ausschließlich aus anderen Variablen zusammensetzen, können mithilfe der `function`-Deklarationen modelliert werden (vgl. Zeile 9). Im vorlie-

genden Beispiel akkumuliert die Variable `I` alle eingehenden Ströme.

Die eingehenden bzw. ausgehenden Ports werden in einem `input`- bzw. `output`-Block definiert. Eingehende Ports können unterschiedliche Typen von Ereignissen erhalten. Der genaue Typ des Ports wird durch ein Schlüsselwort `spike` oder `current` festgelegt. Das Neuron `rc_neuron` deklariert zwei eingehende Ports: den `spikes`-Port in Zeile 28 und den `currents`-Port in Zeile 29. Diese werden innerhalb des `input`-Blockes definiert. Schließlich legt der ausgehende Port in Zeile 32 fest, dass das Neuron `rc_neuron` in der Lage ist, Spikes zu feuern.

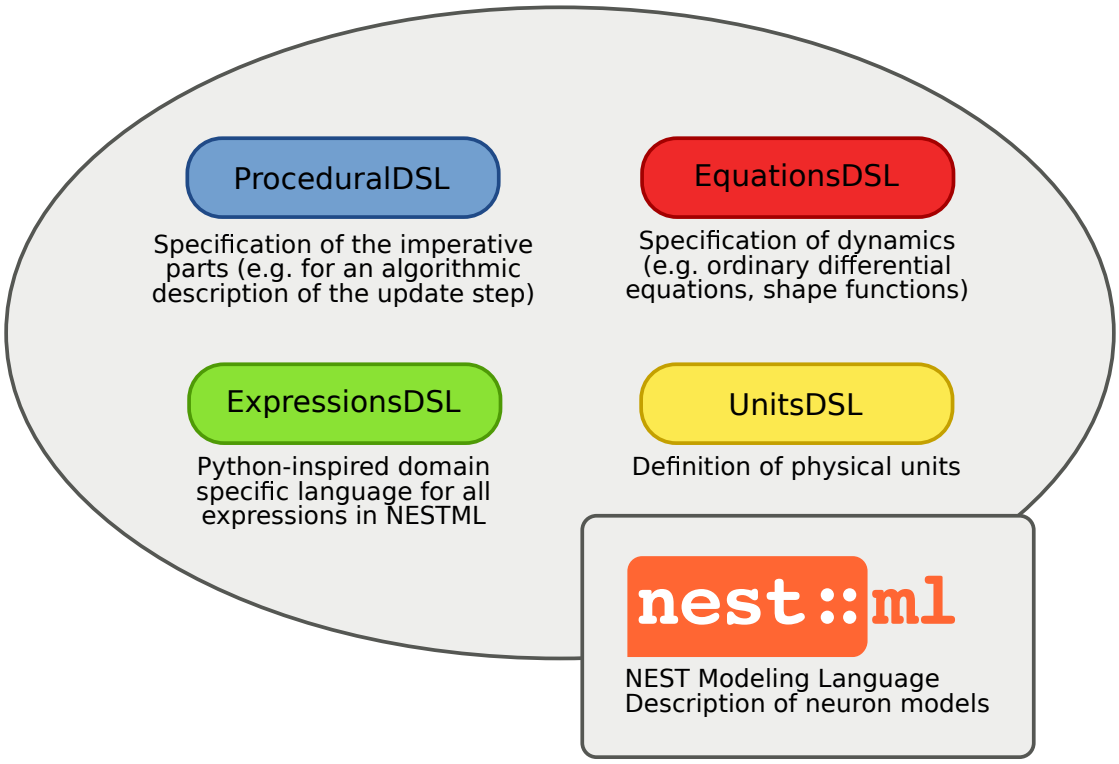


Abbildung 5.2: Struktureller Aufbau der NESTML-Sprachen.

Auf Grammatikebene setzt sich die NESTML-Sprache aus verschiedenen Subsprachen zusammen. Dabei werden die Spracheinbettung und Sprachvererbung aus der MontiCore Language Workbench benutzt, um die einzelnen Subsprachen zu einer monolithischen Sprache zusammenzufügen.

Abbildung 5.2 visualisiert den schematischen Aufbau von NESTML inklusive aller Subsprachen. NESTML ist dabei eine übergeordnete Sprache, die andere Subsprachen einbettet und erweitert (vgl Abschnitt 5.2). Der Zweck von NESTML ist es, die grundlegenden Modellierungselemente zu definieren, mit denen Struktur Verhalten eines Neu-

rons spezifiziert werden können. Die Differenzialgleichungen können mithilfe der *EquationsDSL* spezifiziert werden (vgl. Abschnitt 5.4). Die in einem Neuronenmodell definierten Variablen können physikalische Einheiten verwenden, die mithilfe der *UnitsDSL* modelliert werden (vgl. Abschnitt 5.6). Die *ExpressionsDSL* definiert die Struktur und Form der in NESTML zulässigen Ausdrücke (vgl. Abschnitt 5.5). Schließlich ermöglicht die *ProceduralDSL* die Definition von Modellabschnitten, die nicht durch andere deklarative Konstrukte ausgedrückt werden können und stattdessen in einem imperativen Stil erstellt werden sollen (vgl. Abschnitt 5.3). Im Weiteren werden die einzelnen Subsprachen erläutert.

5.2 NEST Modeling Language

NESTML bietet eine einfache und präzise Syntax für die Definition von biologischen Neuronen. Dennoch verfügt die Sprache aufgrund der Erweiterungsmöglichkeiten und der eingebauten prozeduralen Subsprache über große Flexibilität, wodurch die meisten Punktneuronenmodelle spezifiziert werden können.

Die NESTML-Syntax wurde an die Python-Syntax angelehnt, um die Lernhürde für NESTML-Nutzer zu senken. Unter anderem wird auf die in vielen anderen Programmiersprachen üblichen Semikolons (`;`) am Ende einer Anweisung verzichtet. Auch die Syntax der eingebauten Aktionssprache bedient sich vieler Sprachkonzepte aus der Programmiersprache Python.

Grundsätzlich werden Neuronen und Neuronenkomponenten in Dateien mit der Dateiendung `.nestml` gespeichert. Eine NESTML-Datei kann eine beliebige Anzahl von Neuronen bzw. Komponenten enthalten. Daher ist keine Übereinstimmung zwischen Dateinamen und Inhalt der Datei gefordert.

5.2.1 Neuronen- und Komponentendefinition

Die Definition eines Neurons beginnt mit dem Schlüsselwort `neuron`, gefolgt von seinem Namen. Ein Neuron in NESTML wird anhand seines Namens dateiübergreifend eindeutig identifiziert. Abbildung 5.3 zeigt die Definition eines exemplarischen Neurons mit dem Namen `iaf_neuron`.

NESTML

```
1 package example
2
3 # example neuron definition
4 neuron iaf_neuron:
5   ...
6 end
```

Abbildung 5.3: Eine exemplarische Definition des Neurons `iaf_neuron` in der NESTML-Syntax.

Der vollqualifizierte Name des Neurons setzt sich automatisch aus dem Namen der Datei, dem relativen Pfad der Datei und dem Namen des Neurons bzw. der Komponente zusammen. Dadurch bleibt es dem Modellierer erspart, manuell die Konsistenz zwischen Modellen mit deren Ablageort zu sichern. Potenzielle Fehler werden dennoch von den NESTML-Sprachwerkzeugen erkannt und berichtet.

Wäre das vorliegende Neuron ist in einer Datei `examples.nestml` gespeichert, wäre der qualifizierte Name des Neurons `examples.iaf_neuron`. Die Deklaration des Paketnamens (`package`, vgl. Zeile 1) ist optional und wird, falls ausgelassen, automatisch anhand des Dateinamens bestimmt und nach dem Parsen der Datei gesetzt.

Die erste Art der Wiederverwendung wird in NESTML durch die Vererbungsbeziehung auf Modellebene realisiert (vgl. *RQ2.1*). Syntaktisch wird die Vererbung zwischen zwei Neuronen durch das Schlüsselwort `extends` realisiert, das vom Namen des zu erweiternden Neurons gefolgt wird. Durch die Erweiterung eines Neurons werden alle Elemente des vererbenden Neurons in das erbende Neuron integriert. Attribute, Gleichungen und `shape`-Funktionen können anschließend im erbenden Neuron überschieben werden.

Abbildung 5.4 demonstriert diese Art der Erweiterung. Dabei wird ein Basisneuron `iaf_base` mit Zustandsvariablen für die Modellierung des Membranpotenzials und der Schranke durch das Neuron `iaf_neuron` erweitert. Das Neuron `iaf_neuron` erhält dadurch Zugriff auf die im Neuron `base_iaf` definierten Variablen `V_m`, `theta` und `V_reset`.

```
1 neuron iaf_neuron extends base_iaf:
2   update:
3     if V_m >= theta:
4        V_m = V_reset
5     end
6   end
7 end
```

```
1 neuron base_iaf:
2   state:
3     V_m   mV = -60mV
4     theta mV = -45mV
5     V_reset mV = -60mV
6   end
7
8 end
```

NESTML

Abbildung 5.4: Definition eines Neurons `iaf_neuron` als Erweiterung des Neurons `base_neuron`. Dabei stehen die im Neuron `base_iaf` definierten Zustandsvariablen im Neuron `iaf_neuron` zur Verfügung.

Der hier vorgestellte Vererbungsmechanismus hat einige Vorteile. Die in der Neurowissenschaft üblichen Variablennamen mit etablierter Semantik (z.B. `V_m` für das Membranpotenzial) können so in einem Basismodell definiert werden um die implizite Konvention explizit festzuhalten. Die Vererbung löst auch das Problem der Redundanz, das in Kapitel 1 ausführlich diskutiert wurde. Durch das gezielte Überschreiben der `shape`-Funktionen in Erweiterungsneuronen können die Basisimplementierungen für Differenzialgleichungen und Dynamik vollständig wiederverwendet werden.

Eine andere Art der Wiederverwendung ist durch die Definition von Komponenten gegeben. NESTML erlaubt es, wiederverwendbare und modulare Teile der Spezifikation in Form einer Komponente zu definierten (vgl. *RQ3.2*). Die Definition einer Komponente

beginnt mit dem Schlüsselwort `component`, gefolgt von einem Namen. Im Rumpf einer Komponente können dieselben Elemente wie im Rumpf eines Neurons verwendet werden.

Abbildung 5.5 (A) zeigt die Definition einer wiederverwendbaren Komponente `RefComponent`. `RefComponent` kapselt dabei die Logik der Refraktärphasenberechnung.

(A)

```
component RefComponent:
  function is_refractory() boolean:
    ...
  end

  function decrease() void:
    ...
  end

end
```

(B) NESTML

```
import refractory.RefComponent

neuron iaf_neuron:
  use RefComponent as refr
  ...
  update:
    if refr.is_refractory():
      refr.decrease()
      ...
    end
  end
end
```

Abbildung 5.5: Der Komponentenmechanismus von NESTML. (A) zeigt die Definition einer wiederverwendbaren Komponente, die die Logik der Refraktärphasenberechnung kapselt. (B) zeigt die Verwendung der Komponente aus (A) im Neuron `iaf_neuron`.

Um auf Komponenten zuzugreifen, die in anderen Dateien gespeichert sind, werden in NESTML explizite `import`-Anweisungen verwendet (vgl. Zeile 1 in Abbildung 5.5 (B)). Im vorliegenden Beispiel stellt die Schnittstelle von `RefComponent` zwei Methoden zur Verfügung. Nach dem Importieren kann `RefComponent` mithilfe einer `use`-Anweisung im importierenden Neuron als lokale Variable verwendet werden (vgl. Zeile 4 in Abbildung 5.5 (B)). Der lokal nutzbare Name wird nach dem Schlüsselwort `as` definiert. Beispielsweise werden die Methoden `is_refractory()` und `decrease()` auf diese Weise im `update`-Block des Neurons `iaf_neuron` wiederverwendet.

5.2.2 Variablenblöcke

Variablen die biologische Eigenschaften eines Neurons modellieren spielen eine wichtige Rolle bei der Neuronenspezifikation. Im Folgenden werden drei NESTML-Variablenblöcke vorgestellt, die innerhalb eines Neurons unterstützt werden. Die präzise Semantik der Variablen aus den einzelnen Blöcken wird in Kapitel 8 eingeführt.

State Der Zustandsblock enthält Variablen, die für den zeitlich veränderlichen Zustand des Neurons stehen. In der NESTML-Syntax beginnt der Zustandsblock mit dem Schlüsselwort `state`, gefolgt von einem Doppelpunkt, und endet mit dem Schlüsselwort `end`. Innerhalb des Blockes kann eine beliebige Anzahl von Variablen deklariert werden.

```
1 state:
2   V_m mV = 60mV [[V_m >= -90mV]]
3
4   function V_rel mV = V_m -70mV
5 end
```

NESTML

Requires that V_m value is always above -90mV

Abbildung 5.6: Exemplarischer `state`-Block bestehend aus zwei Variablen, die das Membranpotenzial und ein zu einer Konstante relatives Membranpotenzial modellieren.

Ein exemplarischer `state`-Block wird in Abbildung 5.6 veranschaulicht. Das kanonische Beispiel für eine Zustandsvariable ist die Variable `V_m`, die das Membranpotenzial darstellt (vgl. Zeile 2). Der Aufbau einer Deklaration wird in Abschnitt 5.3 genauer erläutert.

Abgeleitete Variablen sind Variablen, deren Werte sich ausschließlich aus Werten anderer Variablen ableiten lassen. In NESTML werden sie mithilfe einer `function`-Deklaration modelliert (vgl. `V_rel` in Zeile 4). Diese Deklaration besteht aus einem Schlüsselwort `function`, dem Namen, dem Typ und einem definierenden Term. Dabei muss der Term, mit dem man den Wert der Variable bestimmt, stets angegeben werden.

Schließlich können Invarianten für eine Variablendeklaration in doppelten eckigen Klammern definiert werden. Dabei kann eine Invariante ein beliebiger boolescher Ausdruck sein. Die genaue Interpretation der Invarianten aus dem `state`-Block wird in Kapitel 8 gegeben.

Parameters Dieser Block enthält Variablen, die während einer Simulation konstant sind und bei der Instanziierung eines Neurons eingestellt werden. Dieser Block beginnt mit dem Schlüsselwort `parameters`, gefolgt von einem Doppelpunkt, und endet mit dem Schlüsselwort `end`. Innerhalb des Blockes kann eine beliebige Anzahl von Variablen deklariert werden.

```
1 parameters:
2   C_m pF = 250pF [[C_m > 0pF]] # Membrane capacity constant
3   E_L mV = -70mV               # Resting potential
4 end
```

NESTML

Abbildung 5.7: Exemplarischer `parameters`-Block bestehend aus zwei Variablen. Die Variable `C_m` modelliert die Membrankapazität, `E_L` das Ruhepotenzial des Neurons.

Beispiele für Variablen aus dem `parameters`-Block sind die Länge der Refraktärperiode oder die Größe der Membrankapazität. Abbildung 5.7 zeigt exemplarisch einen `parameters`-Block. Die Variable `C_m` in Zeile 3 modelliert die Membrankapazität, `E_L` in Zeile 4 das Ruhepotenzial des Neurons.

Um zu gewährleisten, dass die an das Neuron übergebenen Werte im vom Modellierer vorgesehenen Bereich sind, können Plausibilitätsprüfungen spezifiziert werden. Die Plausibilitätsprüfungen werden am Ende der Variablendeklaration in doppelten eckigen Klammern definiert. Sie werden immer dann geprüft, wenn die Parameter vm Benutzer des Neuronenmodells verändert werden. Die genaue Interpretation der Invarianten aus dem `parameters`-Block wird in Kapitel 8 gegeben.

Internals Variablen, die keiner der beiden vorher genanten Kategorien zuzuordnen sind, können im `internals`-Block definiert werden. Typischerweise hängen diese Variablen von den `parameters`-Variablen ab und werden einmalig vor jeder Simulation berechnet. In der NESTML-Syntax beginnt dieser Block mit dem Schlüsselwort `internals`, gefolgt von einem Doppelpunkt, und endet mit dem Schlüsselwort `end`. Innerhalb des `internals`-Blockes kann eine beliebige Anzahl von Variablen deklariert werden.

NESTML

```
1 internals:
2   # Number of simulation steps in order to simulate 2 milliseconds
3   ref_counts integer = steps(2 ms)
4 end
```

Abbildung 5.8: Exemplarischer `internals`-Block, der eine Variablendeklaration enthält.

Abbildung 5.8 zeigt einen exemplarischen `internals`-Block, der eine Variable `ref_counts` in Zeile 3 enthält. Die Variable `ref_counts` modelliert die Anzahl der Schritte, die in einer zeitdiskreten Simulation notwendig sind, um 2 Millisekunden Realzeit zu simulieren.

5.2.3 Ein- und ausgehende Ports

Damit Neuronen mit anderen Neuronen kommunizieren können, stellt NESTML unterschiedliche Möglichkeiten bereit, um ein- und ausgehende Ports eines Neurons zu spezifizieren. Die eingehenden Ports werden innerhalb eines `input`-Blockes spezifiziert. Der eindeutig spezifizierte ausgehende Port wird innerhalb des `output`-Blockes spezifiziert.

Um Ereignisse von anderen Neuronen oder von der Umgebung zu empfangen, spezifizieren NESTML-Neuronen den `input`-Block. Dieser Block beginnt mit dem Schlüsselwort `input`, gefolgt von einem Doppelpunkt, und endet mit dem Schlüsselwort `end`. Innerhalb dieses Blockes können mehrere benannte eingehende Ports vom Typ `spike` oder `current` definiert werden. Die `spike`-Ports können zusätzlich als `inhibitatory` oder `excitatory` qualifiziert werden, was durch das entsprechende Schlüsselwort definiert wird. Diese Schlüsselwörter beziehen sich jeweils auf die Selektion der eingehenden Signale entsprechend ihrem Gewicht. Falls kein spezifischer Typ bei einem `spike`-Port angegeben wird, empfängt der Port beide Arten von Ereignissen.

Bei der Spezifikation des `input`-Blockes stehen dem Modellierer einige Freiheiten zur

Verfügung. Abbildung 5.9 listet exemplarisch gültige Kombinationen der Ports innerhalb des `input`-Blockes.

(A)

```
input:
  spikes   <- spike # Receives positive and negative weighted spikes

  currents <- current
end
```

(B)

```
input:
  spikes_in <- spike inhibitory # Receives negative weighted spikes
  spikes_ex <- spike excitatory # Receives positive weighted spikes
end
```

(C)

```
input:
  receptor_A <- spike # Receptor with the index 1
  receptor_B <- spike # Receptor with the index 2
end
```

(D)

```
output: spike # 'current' can be used instead
```

Abbildung 5.9: Einige exemplarische `input`-Blöcke. (A) Hier wird ein Block mit zwei Ports definiert. (B) Hier werde zwei `spike`-Ports, die jeweils unterschiedlich gewichtete Spikes verarbeiten, definiert. (C) Hier werden mehrere Ports vom gleichen Typ definiert. (D) Hier wird ein Ausgangsport vom Typ `spike` definiert.

Fall `(A)` zeigt einen Block bestehend aus zwei Ports von unterschiedlichem Typ. Dieser Fall definiert einen `spikes`-Port, der sowohl positiv als auch negativ gewichtete Spikes empfangen kann. Des Weiteren wird ein `currents`-Port definiert, der Ströme, also Ereignisse vom Typ `current`, empfangen kann.

Fall `(B)` definiert zwei unterschiedlich benannte Ports, die `inhibitatory`- bzw. `excitatory`-Ereignisse vom Typ `spike` verarbeiten.

Fall `(C)` zeigt, wie mehrere Ports vom selben Typ im Neuron modelliert werden können. Hier wird eine Menge von `spike`-Ports definiert, die sich lediglich durch ihre Namen unterscheiden. Dabei können alle Ports sowohl `inhibitatory`- als auch `excitatory`-Spikes verarbeiten. Das biologische Äquivalent ist ein Neuron mit mehreren Rezeptoren [PL95].

Unzulässige Kombinationen der Ports im `input`-Block werden durch Kontextbedingungen geprüft. Beispielsweise würde die Konstellation mit genau einem `inhibitory`-`spike`-Port, ohne einen weiteren `excitatory`-Port als Fehler interpretiert, da in dieser Konstellation alle `excitatory`-Ereignisse verloren gehen würden, da kein Port im Modell

diese Spikes verarbeiten könnte.

Damit Neuronen Ereignisse an andere Neuronen oder die Umgebung senden können, wird der `output`-Block verwendet. Dieser Block beginnt mit dem Schlüsselwort `output`, gefolgt von einem Doppelpunkt. Anschließend kann entweder das `spike`- oder das `current`-Schlüsselwort spezifiziert werden. Beispielsweise spezifiziert Abbildung 5.9 (D) einen `output`-Block vom Typ `spike`.

5.2.4 Spezifikation von Differenzialgleichungen

Differenzialgleichungen stellen das wesentliche Instrument für die Modellierung des Verhaltens eines biologischen Neurons dar (vgl. Abschnitt 2.2). Daher bietet NESTML die Möglichkeit, Differenzialgleichungen in mathematischer Notation innerhalb des Neuronenmodells zu spezifizieren.

Gleichungen werden innerhalb des `equations`-Blockes spezifiziert. Dieser Block beginnt mit dem Schlüsselwort `equations`, gefolgt von einem Doppelpunkt, und endet mit dem Schlüsselwort `end`. Der `equations`-Block kann eine beliebige Anzahl von Gleichungen, `shape`-Funktionen und `function`-Variablen enthalten. `function`-Variablen werden in diesem Block genutzt, um die Spezifikation einer Differenzialgleichung leserlicher zu machen. Die genaue Form der einzelnen Differenzialgleichungen wird durch eine dedizierte *EquationsDSL* definiert, die in die NESTML-Sprache eingebettet wird (vgl. Abschnitt 5.4).

NESTML

```
1  equations:
2    shape I_a = (e/tau_syn) * t * exp(-t/tau_syn)
3    function I pA = (I_sum(I_a, spikes) + currents)
4    V_m' = -1/Tau * (V_m - E_L) + 1/C_m * I
5  end
```

Abbildung 5.10: Ein exemplarischer `equations`-Block. Die Funktion `I_a` definiert die Form der postsynaptischen Antwort, die in der darauffolgenden Gleichung zusammen mit einem `spike`-Port im Aufruf der vordefinierten Methode `conv` verknüpft wird.

Abbildung 5.10 demonstriert die Verwendung des `equations`-Blockes an einem konkreten Beispiel. Das Membranpotenzial wird durch eine `state`-Variable `V_m` modelliert. Der Verlauf des Membranpotenzials wird durch eine Differenzialgleichung beschrieben (vgl. Zeile 4). Diese Gleichung verwebdet eine `shape`-Funktion `I_a`, die die postsynaptische Antwort des Neurons beschreibt und eine abgeleitete Variable in Zeile 3. Die Variable `I` bündelt alle ankommenden Ströme. Somit kann die eigentliche Differenzialgleichung kompakt definiert werden.

5.2.5 Spezifikation des Verhaltens

Das Laufzeitverhalten der Neuronen wird innerhalb des `update`-Blockes definiert. Dieser Block beginnt mit dem Schlüsselwort `update`, gefolgt von einem Doppelpunkt, und endet mit dem Schlüsselwort `end`. Für die Implementierung des Blockrumpfes kann der Modellierer auf die imperative eingebettete Sprache *ProceduralDSL* zurückgreifen (vgl. Abschnitt 5.3). Die *ProceduralDSL* verfügt über die gängigen Kontrollstrukturen einer imperativen Programmiersprache. Um die im Gleichungsblock definierten Variablen schrittweise zu lösen, steht die `integrate_odes`-Funktion zur Verfügung. Jedes Neuronenmodell muss genau einen `update`-Block besitzen. Dies wird durch eine Kontextbedingung geprüft (vgl. Abschnitt 7.3).

Eine exemplarische Implementierung der unterschwelligen Dynamik (vgl. Kapitel 2) zeigt Abbildung 5.11. In diesem Beispiel wird die Differenzialgleichung so lange schrittweise gelöst, bis der Wert des Membranpotenzials eine vordefinierte Schranke überschreitet. Danach feuert das Neuron einen Spike und der Wert des Membranpotenzials wird auf den vordefinierten Wert `V_reset` zurückgesetzt.

NESTML

```
update:
  integrate_odes()
  if V_m >= Theta: # threshold crossing
    V_m = V_reset
    emit_spike()
  end
end
```

Abbildung 5.11: Beispiel für eine unterschwellige Neuronendynamik ohne Refraktärphase.

Wiederverwendbare Funktionalität kann in NESTML in Methoden gekapselt sein. Methoden können entweder direkt im definierenden Neuron oder aus einer Komponente referenziert werden. Der Mechanismus für den Aufruf einer Methode aus einer Komponente ist in Abbildung 5.5 erläutert. Eine Methode in einem Neuron bzw. einer Komponente beginnt mit dem Schlüsselwort `function` und endet mit dem Schlüsselwort `end`. Nach dem Schlüsselwort `function` folgt der Methodenname, gefolgt von einer eventuell leeren Liste der Funktionsparameter. Die Parameterliste wird innerhalb von runden Klammern definiert. Jeder Parameter besteht aus einem Namen und seinem Typ. Die Parameter werden durch Kommas voneinander getrennt. Danach kommt ein optionaler Rückgabewert, gefolgt von einem Doppelpunkt. In der Implementierung der Methode steht dem Modellierer die eingebettete *ProceduralDSL* zur Verfügung (vgl. Abschnitt 5.3), um die Logik der Methode umzusetzen.

Abbildung 5.12 zeigt zwei exemplarische Methodendefinitionen, die jeweils für das Setzen und Auslesen einer Zustandsvariablen verantwortlich sind. Die `set_V_m`-Methode weist der Zustandsvariable `V_m` den Wert des Funktionsparameters `V_value` zu. Da der

NESTML

```
1 function set_V_m(V_value mV):
2   V_m = V_value - E_L  # E_L is a variable from the parameters block
3 end
4
5 function get_V_m() mV:
6   return V_m + E_L
7 end
8
9 update:
10   ...
11   V_m_old mV = get_V_m()
12   ...
13 end
```

Abbildung 5.12: Eine exemplarische Definition der `get_V_m()`- und `set_V_m`-Methoden.

Rückgabetyp der Methode ausgelassen ist, wird als Typ `void` angenommenen, der die übliche Semantik hat. Die `get_V_m`-Methode berechnet den Wert der Zustandsvariable `V_m` relativ zur Variable `E_L` und gibt diesen anschließend zurück. die Benutzung der `get_V_m`-Methode im `update`-Block ist im Beispiel in Zeile 11 zu sehen.

Sowohl der `update`-Block als auch der `function`-Block benutzen die *ProceduralDSL*, um die Implementierung des Blockes zu spezifizieren. Der folgende Abschnitt erläutert die Funktionsweise der *ProceduralDSL*.

5.3 ProceduralDSL: Aktionssprache für die imperative Spezifikation

Im Rahmen dieser Arbeit wird eine Aktionssprache [KT08] entwickelt, die in Neuronen benutzt werden kann, um das Verhalten von Funktionen und Neuronendynamiken zu spezifizieren. Vor allem für die Spezifikation der Neuronendynamik existieren einige Modellierungsansätze, die Variationen eines Mealy-Automaten [Bra84, HUM02, Rum96] darstellen. Auch wenn der Automatenansatz sich in der Theorie für die Spezifikation der Dynamik gut eignet, verwendet NESTML eine textuelle Aktionssprache [GKR+07], die den gleichen Funktionsumfang bietet.

Ein Problem des Mealy-Ansatzes ist, dass Neuronenentwickler sich einerseits mit der formalen Theorie der Mealy-Automaten in der Regel nicht gut auskennen und anderseits das Erstellen eines solchen Automaten ohne ein passendes grafisches Werkzeug schwierig ist. Dennoch bietet NESTML die Möglichkeit, mithilfe der Spracherweiterung später eine Automatenspezifikation zu integrieren. Dafür müsste eine entsprechende Automatengrammatik [RRW14, Wor16, RRW16] in die NESTML-Grammatik eingebettet werden.

Da die deklarativen Elemente von NESTML aufgrund ihres Fokus auf das Lösen eines bestimmten Problems, nicht immer ausreichen um die vielfältigen Anforderungen an

die Spezifizierung des Neuronenverhaltens zu erfüllen, stellt NESTML mit der *ProceduralDSL* eine plattformunabhängige prozedurale imperative Sprache zur Verfügung. Die Plattformunabhängigkeit der *ProceduralDSL* wird dadurch gewährleistet, dass sie nur modellierte Elemente referenzieren darf. Die Übersetzung der *ProceduralDSL* in lauffähigen Code findet durch einen Generator statt. Dieser Ansatz wird in der Literatur als *ActionLanguage* bezeichnet [GHK+15a, GHK+15b, KT08].

Der Generator entkoppelt das modellierte System und die Laufzeitumgebung, in der die Simulation der Neuronen stattfindet, vom Neuronenmodell.

Eine Aktionssprache hat im Vergleich zur direkten Einbettung einer GPL in die Modellspezifikation [Sch12] viele Vorteile. Der wesentliche Vorteil besteht darin, dass das modellierte System vom Systemcode getrennt wird. Insgesamt hat die Trennung folgende Vorteile:

Konsistente Modellierung: Aufgrund des Bezugs der *ProceduralDSL* auf die Modellelemente, die in einem Neuron- oder Komponentenmodell definiert sind, entstehen konsistente Modelle, die bereits zur Modellierungszeit semantisch geprüft werden können. In der *ProceduralDSL* können keine Ausdrücke verwendet werden, die die Kenntnis des Generators oder Zielsystems voraussetzen und sich nicht aus Modellen alleine ableiten lassen.

Modulare Weiterentwicklung: Aufgrund dieser Trennung kann ein Generator oder das Zielsystem ausgetauscht werden, ohne dass Änderungen an bestehenden Modellen vorgenommen werden müssen [LRSS]. Die Modelle sind dadurch modular und simulatoragnostisch definiert.

Trennung der Zuständigkeiten [HL95]: Der Modellierer benötigt keine Kenntnisse über das Zielsystem oder die Details des Generators. Somit ist die Entwicklung des Generators und der Neuronenmodelle getrennt und parallel möglich.

Zielplattformunabhängiger Code: Da sich der modellierte Code im Neuronenmodell nicht auf das Zielsystem bezieht und somit keine plattformspezifischen Aspekte des Zielsystemcodegenerators voraussetzt, können gleichzeitig verschiedene Zielsysteme über unterschiedliche Codegeneratoren adressiert werden.

Auch in der konkreten Syntax der *ProceduralDSL* setzt sich die Anlehnung an die Programmiersprache Python fort. Ein wesentlicher Aspekt ist die Art und Weise wie Anweisungen im Programmtext voneinander getrennt werden. Im Unterschied zu Programmiersprachen wie C und Java, in denen Anweisungen durch Semikolons getrennt werden, werden in NESTML und insbesondere in der *ProceduralDSL* stattdessen Zeilenumbrüche benutzt. Auch bei der Umsetzung der bedingten Anweisungen und Schleifen wurde der Python-Stil eingehalten. Beispielsweise müssen Bedingungen nicht innerhalb von runden Klammern definiert werden.

Generell besteht ein *ProceduralDSL*-Programm aus einfachen und zusammengesetzten Anweisungen. Zu den einfachen Anweisungen zählen Deklarationen, Funktionsaufrufe und Zuweisungen. Die zusammengesetzten Anweisungen sind entweder Verzweigungen oder Schleifen. Dabei dürfen die zusammengesetzten Anweisungen beliebige Anweisungen enthalten.

Alle Variablen in der *ProceduralDSL* müssen vor der ersten Benutzung deklariert sein. Variablen sind generell stark typisiert. In Anlehnung an die physikalische Notation bzw. die UML [RJB04] besteht die Deklaration aus einer nicht leeren Liste von Variablennamen gefolgt von deren Typ. Als Datentypen werden in der *ProceduralDSL* primitive Datentypen (z.B. `integer` oder `real`) und die für die Anwendungsdomäne notwendigen physikalischen Einheiten unterstützt (z.B. `mV`, vgl. Abschnitt 5.6 für eine detaillierte Vorstellung der physikalischen Typen). Nach einem optionalen Gleichheitszeichen kann anschließend ein optionaler Initialisierungsausdruck folgen.

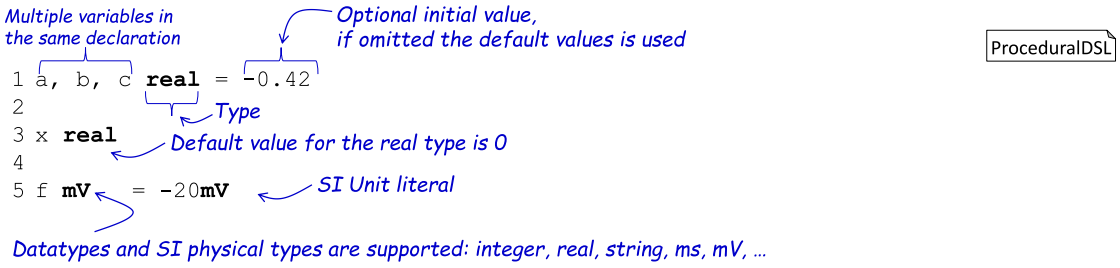


Abbildung 5.13: Unterschiedliche Arten von in NESTML unterstützten Variablendeklarationen.

Abbildung 5.13 zeigt unterschiedliche Varianten von Variablendeklarationen. In Zeile 1 werden drei Variablen `a`, `b`, und `c` vom Typ `real` deklariert und mit demselben skalaren Wert initialisiert. Zeile 3 demonstriert die Möglichkeit, einen wohldefinierten Standardwert für die Initialisierung einer Variablen zu benutzen. Wenn der Initialisierungsausdruck für eine Variable vom Datentyp `real` ausgelassen wird, wird die Standardbelegung verwendet (in diesem Fall der Wert 0). Schließlich zeigt Zeile 5 die Verwendung von physikalischen Datentypen, die in Abschnitt 5.6 näher beschrieben werden. Im vorliegenden Fall wird eine Variable `f` vom Typ *Millivolt* deklariert. Um die Variable `f` korrekt zu initialisieren, muss ein Einheitenliteral verwendet werden (im Beispiel `-20mV`). Einheitenliterale sind im Prinzip numerische Gleitkommazahlen mit einem Postfix, in dem die Einheit spezifiziert wird. Die Details über Einheitendatentypen und Einheitenliteralen werden in Abschnitt 5.6 erläutert.

void: Dieser Datentyp dient als Rückgabewert für Methoden, die keinen Wert zurückgeben. Der Typ entspricht dem Typ `void` aus Programmiersprachen wie *Java* bzw. *C*. `void` darf nicht als Typ einer Deklaration benutzt werden.

boolean: Dieser Datentyp entspricht dem binären Typ, der aus der booleschen Logik bekannt ist [Boo48]. Die möglichen und ausschließlichen Ausprägungen sind: `true` und `false`. Die Standardbelegung ist der Wert `false`.

string: Dieser Datentyp modelliert den Zeichenkettendatentyp (engl: string). Beispiele für seine Ausprägungen sind `"Bob"`, `"Hello World"` und `""`. Die Standardbelegung ist der leere String: `""`.

integer: Dieser Datentyp modelliert Ganzzahlen. Exemplarische Ausprägungen sind `0`, `1` und `42`. Der Standardwert ist `0`.

real: Dieser Datentyp modelliert die rationalen Zahlen mit doppelter Genauigkeit (engl: double precision). Exemplarische Ausprägungen sind `0.0` oder `3.14`. Der Standardwert ist `0.0`.

SI Einheiten (z.B. `mV`, `nS`, `[nS/s]`: Diese können mit einfachen arithmetischen Operationen zu komplexen Einheiten kombiniert werden, z.B. `[nS/s]`. Mögliche Ausprägungen sind: `1.0mV`, `20[nS]`, `5.0[nS/s]`. Der Standardwert ist `0 [type]`. Die Details werden in Abschnitt 5.6 erläutert.

Die Liste oben fasst die verfügbaren Datentypen zusammen und erläutern deren Wertebereich. Zeichenketten (engl: strings) werden mit dem Datentyp `string` definiert. Mit dem Typ `boolean` können Variablen modelliert werden, die die zwei Wahrheitswerte `true` und `false` annehmen können. Es werden zwei numerische Datentypen unterstützt. Ganzzahlige Variablen werden mit dem Datentyp `integer`, Gleitkommazahlen mit dem Datentyp `real` modelliert. Physikalische Einheiten können in einer domänennahen Notation spezifiziert werden.

Die *ProceduralDSL* unterstützt eine implizite Konvertierung einer Variablen vom Typ `integer` zu einer Variablen vom Typ `real`. Als Fehler werden Konvertierungen von Variablen vom Typ `real` und Variablen von beliebigem Einheitentyp behandelt. Auch Konvertierungen anderer Typen sind in NESTML generell verboten.

Eine Zuweisung in der *ProceduralDSL* besteht aus einer linken Seite, auf der eine Variable referenziert wird, einem Zuweisungsoperator (`=`) und einem Term auf der rechten Seite. Abbildung 5.14 fasst die unterstützen Arten von Zuweisungen zusammen. Neben der einfachen Zuweisung, die den Wert der rechten Seite der Variable auf der linken Seite direkt zuweist, bietet die *ProceduralDSL* auch zusammengesetzte Zuweisungen.

Eine zusammengesetzte Zuweisung `a $OP= b` ist eine Abkürzung für einen Ausdruck der Form `a = a $OP b`, wobei `$OP` einer der folgenden Operatoren sein kann: `+`, `-`, `*`, `/`. So ist die Zuweisung `a *= b` zum Beispiel äquivalent zu folgendem Term: `a = a * b`.

Bei allen Zuweisungen muss der Typ der linken Seite kompatibel mit dem Typ der rechten Seite sein. Da es keine implizite Typkonvertierung gibt, müssen diese Typen entweder übereinstimmen oder es muss eine Zuweisung eine Wertes vom Typ `integer` zu einer Variable vom Typ `double` sein.

ProceduralDSL

```
1 a, b integer
2
3 a = b
```

```
1 a, b integer
2
3 a += b
```

```
1 a, b integer
2
3 a -= b
```

```
1 a, b integer
2
3 a *= b
```

```
1 a, b real = 1.0
2
3 a /= b
```

Abbildung 5.14: Exemplarische Beispiele für einfache und zusammengesetzte Zuweisungen.

Die letzte Art der einfachen *ProceduralDSL*-Anweisungen sind Funktionsaufrufe. Ein Funktionsaufruf besteht aus dem Namen der aufzurufenden Funktion und einer Liste der Argumente. Diese werden innerhalb runder Klammern spezifiziert. Dabei muss jeder Term, der als Argument übergeben wird, ein valider *ExpressionsDSL*-Ausdruck sein (vgl. Abschnitt 5.5). Beispielsweise würde der Funktionsaufruf `pow(2.,2.)` die Potenzfunktion für die vorliegenden Argumente berechnen. Auch bei einem Funktionsaufruf müssen die Typen der Argumente mit den erwarteten Typen der Funktionsparameter kompatibel sein.

Die *PrceduralDSL* stellt einige vordefinierte Funktionen bereit. Neben den reinen mathematischen Funktionen (z.B. `log`, `pow`) und Logging-Funktionen (z.B. `info`, `warning`) bietet die *ProceduralDSL* einige für NESTML spezialisierten Funktionen an. Auch die in einem Neuron bzw. einer Komponente definierten Methoden können für einen Funktionsaufruf verwendet werden. Nachfolgend werden alle in der `PrceduralDSL` verfügbaren Funktionen erläutert.

`conv`: Diese Funktion kann ausschließlich in Differenzialgleichungen verwendet werden, um einen Port, der im `input`-Block definiert ist, mit einer `shape`-Funktion zu falten.

`emit_spike`: Der Aufruf dieser Methode modelliert das Feuern eines Spikes. Diese Funktion kann ausschließlich im `update`-Block verwendet werden.

`integrate_odes`: Mithilfe dieser Funktion werden alle Differenzialgleichungen aus dem `equations`-Block einen Simulationsschritt propagiert. Diese Funktion kann ausschließlich im `update`-Block verwendet werden.

`resolution`: Mit dem Aufruf dieser Funktion wird die aktuelle Auflösung der Simulation als Wert in `ms` bestimmt. Das Zeitmodell der Simulation ist in Abschnitt 8.1 diskutiert. Die Funktion darf im `internals`- oder `update`-Block benutzt werden.

`steps`: Mithilfe dieser Funktion wird eine Zeit in Millisekunden in die Anzahl von Simulationsschritten umgerechnet. Die Anzahl der Schritte entspricht der Dauer des Zeitintervalls unter der vorliegenden Simulationsauflösung. Das Zeitmodell der Simulation ist in Abschnitt 8.1 diskutiert.

`delta`: Eine spezielle `shape`-Funktion, die eine Deltaverteilung [Pet12] modelliert. Die Funktion darf ausschließlich als Argument der `conv`-Methode verwendet werden.

Darüber hinaus unterstützt die *ProceduralDSL* Logging mithilfe der Methoden `info` und `warning` und das Ziehen von Zufallszahlen vom Typ `real` und `integer` durch die Methoden `random` und `randomInt`.

Die *ProceduralDSL* unterstützt unterschiedliche Anweisungen für die bedingte Ausführung bzw. Verzweigung. Abbildung 5.15 fasst diese Möglichkeiten zusammen.

(A)

```
1 if 2 < 3:
2  [statements]+
3 end
```

(B) ProceduralDSL

```
1 if 2 < 3:
2   [statements]+
3 else:
4    [statements]+
5 end
```

(C)

```
1 if 2 < 3:
2   [statements]+
3 elif 4>6:
4   [statements]+
5 else:
6   [statements]+
7 end
```

(D)

```
1 if 1 < 4:
2   if 2 < 3:
3     [statements]+
4   end
5 end
```

Abbildung 5.15: Zusammenfassung der sprachlichen Konstrukte für bedingte Ausführung von Anweisungen.

Fall `(A)` demonstriert den einfachsten Fall, in dem die Ausführung von Anweisungen von einer einzelnen Bedingung abhängt. Syntaktisch beginnt die bedingte Anweisung mit einem Schlüsselwort `if`, gefolgt von einem booleschen Ausdruck. Anschließend wird ein Doppelpunkt (`:`) gesetzt. Der Rumpf der `if`-Verzweigung kann beliebige Anweisungen enthalten. Das Ende der Verzweigung wird durch das Schlüsselwort `end` gekennzeichnet.

Fall `(B)` demonstriert ein Beispiel der Verzweigung mit einem zusätzlichen `else`-Zweig. Der `else`-Zweig ist immer der letzte Zweig einer Verzweigung, falls er vorkommt. Er wird ausgeführt, wenn keine der vorherigen Bedingungen efüllt ist. Syntaktisch wird dieser Zweig mit einem Schlüsselwort `else`, gefolgt von einem Doppelpunkt eingeleitet. Der Rumpf des `else`-Zweigs kann beliebige Anweisungen enthalten.

Fall `(C)` demonstriert die Verwendung des `elseif`-Zweigs. Der `elseif`-Zweig ermöglicht es, mehrere Bedingungen in einem Anweisungsblock zu definieren. Syntaktisch gleicht dieser Block dem einfachen `if`-Zweig. Der Rumpf des `elseif`-Zweigs kann beliebige Anweisungen enthalten.

`if`-Anweisungen können beliebig tief geschachtelt werden. Fall `(D)` zeigt eine exemplarische zweifache Schachtelung einer `if`-Anweisung. Dabei wird die Möglichkeit genutzt,

dass der Rumpf der `if`-Anweisung sowohl einfache als auch zusammengesetzte Anweisungen enthalten darf.

Die letzte Art der zusammengesetzten Anweisungen sind Schleifen. Um eine sich wiederholende Folge von Anweisungen in der *ProceduralDSL* auszudrücken, können zwei Arten von Schleifenkonstrukten verwendet werden. Zum einen steht eine universelle `while`-Schleife zur Verfügung. Syntaktisch beginnt die `while`-Schleife mit einem Schlüsselwort `while`, gefolgt von einer booleschen Bedingung. Am Ende der Bedingung wird ein Doppelpunkt (`:`) gesetzt. Das Ende der `while`-Schleife wird durch das Schlüsselwort `end` gekennzeichnet. Der Rumpf der `while`-Schleife kann beliebige Anweisungen enthalten.

Eine stets terminierende `while`-Schleife zu erstellen, ist im Allgemeinen eine fehleranfällige Aufgabe, da sich der Benutzer selbst um deren Terminierung kümmern muss. Daher bietet die *ProceduralDSL* eine `for`-Schleife mit einer expliziten Zählervariablen an. Diese Art der Schleife erlaubt es, über eine endliche Menge von Zahlen zu iterieren. Die Menge wird dabei durch Angabe der unteren bzw. oberen Grenze und der Schrittweite definiert. Dadurch wird gewährleistet, dass die Iteration stets terminiert.

ProceduralDSL

(A)
```
a integer

while a <= 10:
  [statements]+
end
```

(B)
```
a integer

for a in 1 ... 5:
  [statements]+
end
```

(C)
```
a integer

for a in 1 ... 5 step 2:
  [statements]+
end
```

(D)
```
a integer

for a in 0.1 ... 0.5 step 0.1:
  [statements]+
end
```

Abbildung 5.16: Unterschiedliche Arten von Schleifen in der *ProceduralDSL*. Fall (A) zeigt eine `while`-Schleife, Fälle (B), (C) und (D) unterschiedliche Arten von `for`-Schleifen.

Abbildung 5.16 gibt einen Überblick über die verfügbaren Schleifenkonstrukte. Fall (A) zeigt ein einfaches Beispiel einer `while`-Schleife. Bei dieser `while`-Schleife müsste der Modellierer sich im Rumpf selbstständig darum kümmern, dass die Schleife terminiert.

Die anderen Fälle demonstrieren unterschiedliche Arten von `for`-Schleifen. In Fall `(B)` iteriert die Variable `a` über eine Menge aus Ganzzahlen (`{1, 2, 3, 4, 5}`). Die Schrittweite kann beim Generieren des Zahlenbereichs mithilfe des Schlüsselworts `step` kontrolliert werden. Im Beispiel `(C)` wird die Schrittweite auf den Wert 2 gesetzt. Das hat zur Folge, dass die Variable `a` über die Menge `{1, 3, 5}` iteriert. Schließlich können Intervalle aus rationalen Zahlen für die Iteration verwendet werden. Fall `(D)` zeigt das Intervall bestehend aus den rationalen Zahlen `{0.1, 0.2, 0.3, 0.4, 0.5}`.

5.4 EquationsDSL für die Beschreibung der Differenzialgleichungen

Differenzialgleichungen spielen eine wichtige Rolle bei der Spezifikation von Neuronenmodellen (vgl. Kapitel 2). NESTML unterstützt deshalb die Spezifikation von Differenzialgleichungen in einer anwenderfreundlichen Notation. Dabei werden Differenzialgleichungen durch die in der Mathematik übliche Apostrophennotation definiert (z.B. V'), die im Kontext von NESTML einer Ableitung nach Zeit (z.B. $\frac{\partial V}{\partial t}$) entspricht.

(A) EquationsDSL

```
equations:
  shape I_a = (e/tau) * t * exp(-t/tau)
  V_m' = -V_m/Tau + conv(I_a, spikes)/C_m
end
```

(B)

```
state:
  I_a  [pA]     = 0pA
  I_a' [pA/ms] = (e * 1pA) / tau_syn
end

equations:
  I_a'' = -g_I' / tau
  I_a' = I_a' - I_a /tau
  V_m' = -V_m/Tau + conv(I_a, spikes)/C_m
end
```

Defines explicit initial values for each order

(C)

```
equations:
  shape I_a = (e/tau) * t * exp(-1/tau*t)
  I_syn pA = conv(I_a, spikes)
  V_m' = -V_m/Tau + I_syn/C_m
end
```

Factors out a complex term

Abbildung 5.17: Verschiedene Modellierungselemente im `equations`-Block. (A) zeigt eine `shape`-basierte Notation für die Definition der Faltung der postsynaptischen Ströme (B) zeigt eine zu (A) äquivalente Darstellung als Gleichungssystem mit Anfangswerten (C) zeigt die Definition eines Synonyms für die klarere Definition von Differenzialgleichung.

Abbildung 5.17 zeigt drei wesentliche Elemente dieser Sprache:

Gleichungen: Differenzialgleichungen werden syntaktisch durch eine Referenz zu einer Zustandsvariablen und einem Postfix, bestehend aus einer nicht leeren Liste aus `'`-Symbolen deklariert. Die Anzahl von `'`-Symbolen gibt die Ordnung der jeweiligen Differenzialgleichung an.

Shapes: `shape`-Funktionen sind explizite Funktionen, die in Abhängigkeit der Zeitva-

riable `t`, den Verlauf der postsynaptischen Ströme beschreiben. Nach der Definition der analytischen Darstellung können `shape`-Funktionen ausschließlich in einem `conv`-Aufruf verwendet werden (vgl. die Zeilen 2 und 3 in Abbildung 5.17 `(A)`).

Funktionen: Da Gleichungsdefinitionen im Allgemeinen komplex und unübersichtlich werden können, erlaubt die `EquationsDSL` Teile der Gleichung in separierte benannte Ausdrücke auszulagern (vgl. I_{syn} in Abbildung 5.17 `(C)`).

Fall `(B)` in Abbildung 5.17 weist eine Besonderheit auf. Im Unterschied zu Fall `(A)` müssen die Anfangswerte für jede Differenzialgleichung explizit angeben werden (vgl. Zeile 2 und 3). Dabei müssen stets alle Anfangswerte angegeben werden, da die Spezifikation der Differenzialgleichungen sonst nicht eindeutig wäre.

5.5 Ausdruckssprache ExpressionsDSL

In der vorliegenden Arbeit spielt die Kompatibilität der *ExpressinsDSL* zu Python-Ausdrücken eine wichtige Rolle. Neben der besseren Vertrautheit für Python-affine Modellierer hat dies den Vorteil, dass diese Ausdrücke direkt in validen Python-Code überführt werden können. Die Kompatibilität zu Python erlaubt eine nahtlose Integration mit generierten Python-Scripten, die zum Lösen der im Modell definierten Differenzialgleichungen verwendet werden. Die Verwendung von Python zur Modellanalyse ist in Unterabschnitt 8.1.6 diskutiert.

Die *ExpressionsDSL* kann auf eine rekursive Weise spezifiziert werden. Dabei sind die folgenden, als Terme bezeichneten Ausdrücke valide `ExpressionsDSL`-Modelle:

Variablen: Damit sind alle Variablen gemeint, die anhand ihrer Namen referenziert werden können.

Funktionsaufrufe: Das sind Referenzen zu den vordefinierten Funktionen (z.B. `log`) oder Funktionen, die innerhalb der Komponenten bzw. Neuronen definiert sind.

Boolesche Literale: `true, false`.

Numerische Literale: 1, 20.0, 5.0.

Physikalische Literale: 1.0mV, 20nS 5.0[nS/s].

Unendlichkeit-Symbol: Das Symbol `inf` repräsentiert positive Unendlichkeit. Mit dem Symbol `-inf` wird negative Unendlichkeit dargestellt.

String-Literale: Alle Stringliterale wie zum Beispiel `"Bob"` und `""`.

Operator	Beschreibung	Beispiel
`()`	Klammerausdrücke	`(a)`
`**`	Potenzierungsausdrücke, d.h. $base^{power}$, wobei *base* und *power* beides valide Ausdrücke sind. Dieser Operator ist rechtsassoziativ, d.h `a**b**c` $\leftrightarrow$ `(a**(b**c))`	`a ** b`
`+`, `-`, $\sim$	Unäres Plus, unäres Minus und bitweise Negation	`-a`, `-b` $\sim$`c`
`*`, `/`, `%`	Multiplikations-, Divisions- und Modulooperation	`a * b`, `a % b`
`+`, `-`	Additions- und Subtraktionsoperation	`a + b`, `a - b`
`<<`, `>>`	Bit-Shift nach links und rechts. Der rechte Operand des Operators bestimmt die Anzahl der zu verschieben Bits	`a << n`, `a >> n`
`&`, `^`,`|`	Bitweise-Operationen: bitweises UND, bitweises ODER und bitweises XOR	`a&b`, `a^b`
`<`, `<=`, `==`, `!=`, `>=`, `>`	Relationsoperationen wie bspw. kleiner, größer, gleich oder ungleich.	`a <= b`, `a != b`
`not`, `and`, `or`	Logische Konjunktion, logische Disjunktion und logische Negation	`not a`, `a or b`
`?:`	Der ternäre logische Operator, der abhängig von der Bedingung entweder das erste oder zweite Argument des Operators zurückgibt	`c?a:b`

Tabelle 5.1: Zusammenfassung der verfügbaren Operatoren, um neue *ExpressionsDSL*-Ausdrücke aus anderen validen *ExpressionsDSL* Ausdrücken zu bilden.

Vordefinierte Operatoren werden verwendet, um neue Ausdrücke zu erstellen. Zwei beliebige valide Ausdrücke `a` und `b`, ein boolescher Ausdruck `c` und ein `integer`-Ausdruck `n` können mithilfe der vordefinierten Operationen (vgl. Tabelle 5.1), zu validen Ausdrücken verknüpft werden.

5.6 UnitsDSL: Sprache für der physikalischen Einheiten

Das *International System of Units* (SI) [Tay95] ist ein international festgelegtes Einheitensystem. Unter die SI-Einheiten fallen die meisten physikalischen bzw. naturwissenschaftlichen Einheiten [GMW06]. Um das System möglichst flexibel und einheitlich zu gestalten, definiert das SI-System Basiseinheiten und abgeleitete Einheiten. Die Basisein-

heiten werden als atomar angesehen. Die abgeleiteten Einheiten werden in Abhängigkeit von Basiseinheiten definiert. Tabelle 5.2 fasst die sieben Basiseinheiten zusammen.

Quantität	Symbol	SI Name	UnitsDSL Name
Länge	L	Meter	m
Masse	M	Kilogramm	kg
Zeit	T	Sekunde	s
Elektrischer Strom	I	Ampere	A
Temperatur	Θ	Kelvin	K
Menge einer Substanz	N	Mol	mol
Helligkeitsintensität	J	Candela	cd

Tabelle 5.2: Spezifikation der sieben festgelegten SI-Basiseinheiten [Tay95].

Alle Einheiten im SI-System können um einen multiplikativen Faktor erweitert werden. Dabei wird ein festgelegtes Präfix dem SI-Einheitennamen hinzugefügt. Tabelle 5.3 fasst alle zulässigen Präfixe zusammen.

Faktor	SI Name	UnitsDSL Präfix	Faktor	SI Name	UnitsDSL Präfix
10^{-1}	deci	d	10^{1}	deca	da
10^{-2}	centi	c	10^{2}	hecto	h
10^{-3}	milli	m	10^{3}	kilo	k
10^{-6}	micro	μ	10^{6}	mega	M
10^{-9}	nano	n	10^{9}	giga	G
10^{-12}	pico	p	10^{12}	tera	T
10^{-15}	femto	f	10^{15}	peta	P
10^{-18}	atto	a	10^{18}	exa	E
10^{-21}	zepto	z	10^{21}	zetta	Z
10^{-24}	yocto	y	10^{24}	yotta	Y

Tabelle 5.3: Eine Zusammenfassung der festgelegten multiplikativen Faktoren [Tay95].

Im SI-System werden alle zusätzlichen Einheiten als Kombination der Basiseinheiten definiert. Dies geschieht auf folgende Weise: Jede der sieben Einheiten wird ein rationaler Koeffizient zugeordnet. Zusammen mit der Magnitude entsteht eine achtstellige Signatur, die eine beliebige Einheit beschreiben kann. Diese Signatur kann als Formel nach dem folgenden Schema aufgefasst werden:

$$Q = 10^{x} \times L^{\alpha} \times M^{\beta} \times T^{\gamma} \times I^{\delta} \times \Theta^{\epsilon} \times N^{\zeta} \times J^{\eta}$$

Diese Signatur wird hier exemplarisch verwendet, um die Einheit *Millifarad* darzustellen:

$$mF = 10^{-3} \times m^{-2} \times kg^{-1} \times s^4 \times A^2 \times K^0 \times mol^0 \times cd^0$$

Auch in der *UnitsDSL* werden die Einheiten in zwei Kategorien eingeteilt: Die erste Kategorie besteht aus einfachen Basiseinheiten, die anhand des Namens und eines Präfixes referenziert werden. Um abgeleitete Einheiten zu definieren, können die Basiseinheiten zu zusammengesetzten Einheiten verknüpft werden.

Alle einfachen Basiseinheiten sind valide Einheiten. Zwei valide Einheitenausdrücke `a`, `b` und eine Ganzzahl `n` können dann mit folgenden Operatoren neue Einheitenausdrücke bilden:

`a**n` **(Die Potenzfunktion):** Da in Neuronenmodelle viele Einheiten der Form `a**-1` vorkommen, kann diese Notation durch Ausdrücke der Form `1/a` ersetzt werden. Das Exponentialargument kann nur eine ganzzahlige Zahl sein.

`a*b, a/b`**:** Multiplikations- und Divisions-Operationen.

`(a)`: Klammerung, z.B. `(1/b)**-2`

Nach der initialen Analyse der bereits existierenden Neuronenmodelle wurden einige abgeleitete Einheiten identifiziert, die immer wiederkehren. Für diese Einheiten wurden einfache Einheitensymbole definiert, so dass diese mit ihren Namen direkt verwendet werden können. Tabelle 5.4 fasst diese neuen Symbole zusammen.

Bq	C	F	Gy
Hz	J	N	Ohm
Pa	S	Sv	TH
V	W	Wb	kat

Tabelle 5.4: Zusammenfassung der abgeleiteten physikalischen Einheiten, die in der NESTML-Notation unter einfachen Namen zur Verfügung stehen.

Statt der Entwicklung eines neuen Einheitensystems, wäre auch eine Integration bzw. Erweiterung eines bereits existierten Ansatzes denkbar. Abschließend werden hier exemplarisch zwei solcher Systeme diskutiert:

Modelica [Fri10] ist eine Modellierungssprache mit einem starken Fokus auf die Modellierung von Differenzialgleichungen. Diese kann zur Modellierung verschiedener mechanischer, elektrischer oder physikalischer Prozesse verwendet werden. Teil der Modellierungssprache ist auch ein System für physikalische Einheiten. Dennoch sieht die Ausführungsumgebung von *Modelica* keine Anknüpfungspunkte vor, mit denen eine Integration mit der restlichen NESTML Infrastruktur realisierbar wäre.

SymPy [MSP+16]: Es wäre möglich das Einheitensystem des SymPy-Frameworks zu adoptieren, da NESTML dieses Framework bereits für die Analyse von Gleichungen während der Codegenerierung verwendet (vgl. Unterabschnitt 8.1.6). Diese Alternative wurde jedoch aus zwei Gründen verworfen: Zum einen wäre NESTML dann an die syntaktische Form der Einheiten in *SymPy* angewiesen, zum anderen ist eine nahtlose Integration ai die restlichen Abläufe der Sprachverarbeitungsinfrastruktur von NESTML nicht möglich, da dieses Analyse- und Sprachmodul als externes Modul keine passenden Anknüpfungspunkte vorsieht.

5.7 Zusammenfassung

Dieses Kapitel hat die NESTML-Syntax vorgestellt. Dabei wurde der sprachliche Aufbau von NESTML erklärt und die beteiligten Subsprachen mit ihren Modellierungselementen ausführlich und vollständig erläutert. Nachfolgend wird die Implementierung von NESTML mithilfe der MontiCore Language Workbench erklärt. F

Kapitel 6

Umsetzung von NESTML mit der MontiCore Workbench

Nach der Vorstellung der konkreten Syntax von NESTML im vorherigen Kapitel, werden nun die Besonderheiten der Implementierung aller NESTML-Subsprachen mit der MontiCore Workbench erläutert. Der Fokus dieses Kapitel liegt auf der grammatikalischen Umsetzung und der Konzeption der Symboltabelle. Dabei werden die Grundlagen aus Kapitel 4 als bekannt vorausgesetzt.

Die hier diskutierten Grammatiken sind im Vergleich zu Grammatiken für das Erstellen des NESTML-Frontends weniger detailiert. Dennoch demonstrieren sie die wesentlichen Eigenschaften der zugrundeliegenden Grammatiken, so dass die grammatikalische Umsetzung der in NESTML verwendeten Sprachen mit der MontiCore Workbench nachvollzogen werden kann. Die vollständigen Grammatikdefinitionen sind in Anhang D zu finden.

6.1 Grammatiküberblick

Die MontiCore Workbench unterstützt verschiedene Mechanismen zur Erweiterung von Grammatiken. Beispielsweise sind die Grammatikvererbung und die Grammatikeinbettung wichtige Methoden, um die Komplexität der grammatikalischen Spezifikation einer DSL zu reduzieren. Die Grammatikerweiterung erlaubt es, eine Menge von bereits existierenden Grammatiken wiederzuverwenden. In der aktuellen Version der MontiCore Workbench wird die Grammatikvererbung auch dazu benutzt, die Grammatikeinbettung zu realisieren.

Die sprachliche Realisierung von NESTML ist in mehrere Subsprachen aufgegliedert, um eine modulare und unabhängige Weiterentwicklung der Subsprachen zu ermöglichen. Abbildung 6.1 visualisiert diese Dekomposition anhand der definierenden Grammatiken der einzelnen Subsprachen. Dabei wird jede Subsprache als eigenständige Grammatik aufgefasst. Somit können die einzelnen Subsprachen modular und unabhängig weiterentwickelt werden. Desweiteren werden die Grammatiken mit Produktionen für Literale (z.B. `Name` oder Gleitkommazahlen) vollständig aus der MontiCore-Grammatiksammlung wiederverwendet.

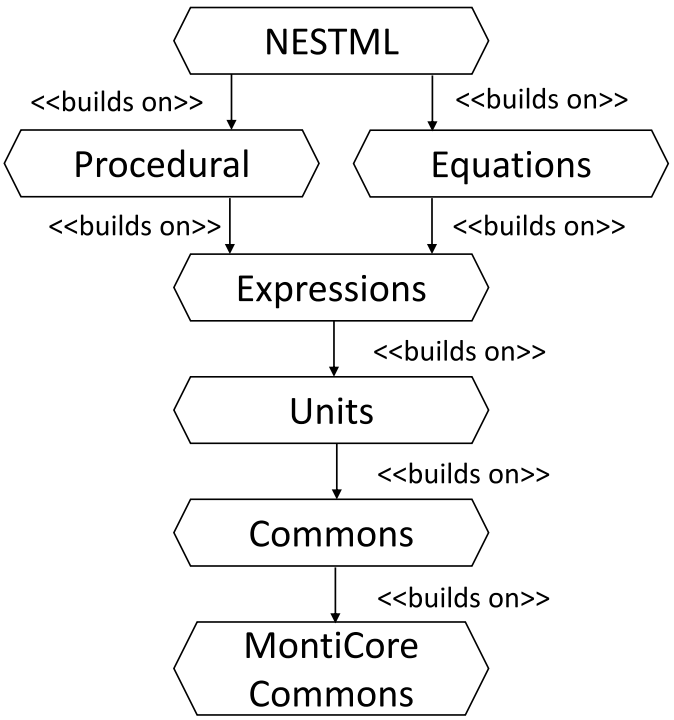


Abbildung 6.1: Die Grammatikhierarchie der Modellierungssprache NESTML. NESTML setzt sich aus verschiedenen Grammatiken zusammen. Jede Grammatik ist für die Spezifikation eines Teilaspekts des Neurons verantwortlich.

MontiCore Commons: Diese Grammatiken sind Teil der Distribution von MontiCore. In diesen Grammatiken werden unter anderem Lexer- und Parser-Produktionen für Namen, Leerzeichen (engl: whitespaces) und verschiedene Literale definiert. Ausführliche Erläuterungen zu diesen Grammatiken ist in [Sch12] zu finden.

Commons: In dieser Grammatik werden der Stil von NESTML-Kommentaren und die Handhabung von Zeilenumbrüchen definiert. Zum einen werden Zeilenumbrüche nicht ignoriert, wie es in den herkömmlichen Programmiersprachen üblich ist, zum anderen werden Kommentare entsprechend der Form von Kommentaren in der Programmiersprache Python festgelegt.

Units: Diese Grammatik definiert Produktionen für die Modellierung von physikalischen Einheiten (vgl. Abschnitt 5.6).

Expressions: Diese Grammatik definiert Produktionen für die gültigen Ausdrücke (vgl. Abschnitt 5.5).

Procedural: Diese Grammatik definiert die imperative Sprache *ProceduralDSL* (vgl. Abschnitt 5.3). Dabei werden neben den Produktionen für Literale auch die Ausdrücke aus der `Expressions`-Grammatik eingebettet.

Equations: Diese Grammatik definiert die in Abschnitt 5.4 erläutertete deklarative Sprache für die Spezifikation von Differenzialgleichungen. Dabei werden auch Ausdrücke aus der `Expressions`-Grammatik eingebettet.

NESTML: Schließlich erweitert diese Grammatik die `Procedual`- und `Equations`-Grammatiken, um mit deren Hilfe die Struktur von Neuronen und Komponenten festzulegen.

Die `Commons`-Grammatik erfüllt zwei Aufgaben. Zum einen wird der Stil der Kommentare in dieser Grammatik konfiguriert, zum anderen wird der von MotiCore generierte Parser angepasst, damit Leerzeichen nicht ignoriert werden. Auch die `NEWLINE`-, `BLOCK_OPEN`- und `BLOCK_CLOSE`-Produktionen sind Teil der `Commons`-Grammatik. Weitere Hintergrundinformationen zur technischen Umsetzung der vollständigen `Commons`-Grammatik (vgl. Listing D.2) sind in [Par13] zu finden.

Im Weiteren werden die wesentlichen Eigenschaften der in Abbildung 6.1 eingeführten Grammatiken an auf das wesentliche reduzierten Versionen dieser Grammatiken erläutert. Die vollständigen Grammatikdefinitionen sind in Anhang D zu finden.

6.1.1 NESTML-Grammatik

Neuronen werden in Dateien gespeichert. Die gesamte Datei wird auf der Grammatikebene durch eine `NESTMLCompilationUnit`-Produktion repräsentiert (vgl. Abbildung 6.2). Die `NESTMLCompilationUnit` beginnt mit einer beliebigen Anzahl von `import`-Anweisungen (vgl. Zeile 2) und kann eine beliebige Anzahl von Neuronen enthalten (vgl. Zeile 3). Eine `import`-Anweisung (vgl. Zeile 5) ist dabei ein vollqualifizierter Name des zu importierenden Elements oder eine Stern-Importanweisung, die alle Elemente aus der referenzierten Datei importiert (vgl. Abschnitt 6.2).

MCG

```
1 NESTMLCompilationUnit =
2     (Import | NEWLINE)*
3     (Neuron | NEWLINE)*;
4
5 Import = "import" QualifiedName ([star:".*"])?;
6 Neuron = "neuron" Name ("extends" base:Name)? Body;
```

Abbildung 6.2: Definition der Nichtterminale für die Definition einer Datei mit Neuronenmodellen.

Die Produktion für die Spezifikation eines Neurons besteht aus dem Schlüsselwort `neuron`, einem Namen, einer optionalen Referenz zu einem Basisneuron und dem Rumpf

(vgl. Zeile 6). Eine Besonderheit der `NESTMLCompilationUnit`-Produktion ist die Handhabung von Zeilenumbrüchen. Im Unterschied zur herkömmlichen Interpretation in Programmiersprachen, wie Java oder C++, werden Zeilenumbrüche in NESTML nicht ignoriert, sondern dienen dazu, Anweisungen voneinander zu trennen. Auf der Grammatikebene wird dies durch das Vorkommen des `NEWLINE`-Nichtterminals festgelegt (vgl. die Zeilen 2-3).

Die MontiCore Workbench erlaubt es, eine Grammatik durch Alternativen in Subsprachen zu erweitern, indem ein `Interface`-Nichtterminal der ersten Grammatik in der Subsprache durch neue Produktionen implementiert wird. In Abbildung 6.3 wird diese Funktionalität von MontiCore benutzt, um die Struktur der Neuronen-Definitionen flexibel zu gestalten.

```
1 interface BodyElement;
2
3 Body = BLOCK_OPEN
4          (NEWLINE | BodyElement)*
5        BLOCK_CLOSE;
6
7 Var_Block implements BodyElement = ...
8 Equations implements BodyElement = ...
9 Dynamics  implements BodyElement = ...
```

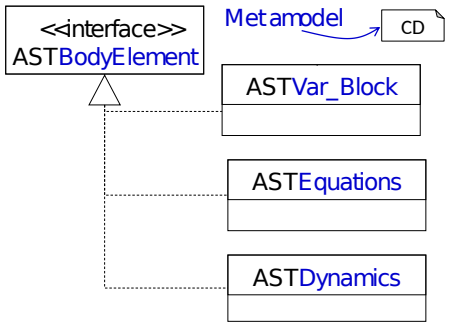


Abbildung 6.3: Definition der Nichtterminale für die Spezifikation des Neuronenrumpfes. Das Klassendiagramm auf der rechten Seit zeigt das generierte Metamodell.

Wenn Parser-Produktionen ein `Interface`-Nichtterminal auf der Grammatikebene implementieren, implementieren auch die Klassen des generierten Metamodells das entsprechende Interface. Im vorliegenden Fall wird dadurch eine konsistente Verarbeitung von allen möglichen `ASTBodyElement`en ermöglicht. Das `BodyElement`-Interface erlaubt es, NESTML nachträglich durch neue Blocktypen in Subsprachen zu erweitern. Dafür müsste die `NESTML`-Grammatik erweitert werden. In der Grammatik der Subsprache wäre es möglich, das `BodyElement`-Interface durch neue Produktionen zu implementieren. Unter anderem könnte auf diese Weise eine Automatengrammatik [Wor16] eingebettet werden, um die Dynamik des Neurons als Mealy Automaten [Rum96] spezifizieren zu können.

Alle Variablenblöcke des Neurons (z.B. die `state`-, `parameters`- und `internals`-Blöcke) werden durch dieselbe Produktion `Var_Block` modelliert. Abbildung 6.4 demonstriert diese Produktionen. Der Rumpf aller Variablenblöcke befindet sich zwischen den Schlüsselwörtern `:` und `end`. Die genaue Unterscheidung des Blockes findet durch das Erkennen eines lexikalischen Tokens statt (vgl. den `["state"]`-Ausdruck in Zeile 2). Auf der Ebene des Metamodells werden drei Methoden (e. g. `isState()`) in der

```
1 Var_Block implements BodyElement =                                    MCG
2     (["state"]|["parameter"]|["internal"])
3     BLOCK_OPEN
4       (AliasDecl | NEWLINE)*          This production is defined
5     BLOCK_CLOSE;                      in the Procedural-grammar
6
7  AliasDecl = (["recordable"])? Declaration;
```

Abbildung 6.4: Definition des Nichtterminals für die Spezifikation der unterschiedlichen Variablenblöcken

generierten `ASTVar_Block`-Klasse erzeugt. Die jeweilige Methode gibt genau dann den Wert `true` zurück, wenn das entsprechende Schlüsselwort in der konkreten Syntax eines NESTML-Modells erkannt wurde.

`AliasDecl`-Produktionen legen die innere Struktur der Variablenblöcke fest. Die `AliasDecl`-Produktion bettet eine Produktion aus der Grammatik der *ProceduralDSL* ein (vgl. Zeile 7). Die eingebettete `Declaration`-Produktion wird lediglich um ein optionales Schlüsselwort `recordable` ergänzt.

Abbildung 6.5 zeigt die grammatikalische Umsetzung des `equations`-Blockes. Dieser Block wird durch ein Schlüsselwort `equations` in der konkreten Syntax eines NESTML-Modells eingeleitet. Der Rumpf des Blockes befindet sich zwischen den Schlüsselwörtern `:` und `end`. Die innere Struktur des Blockes wird vollständig mithilfe der eingebetteten Produktion `OdeDeclaration` aus der `Equations`-Grammatik definiert. Diese Produktion wird in Unterabschnitt 6.1.3 näher erläutert.

```
1 Equations implements BodyElement =                                    MCG
2     "equations"
3     BLOCK_OPEN                       This production is defined
4       OdeDeclaration                 in the Equations-grammar
5     BLOCK_CLOSE;
```

Abbildung 6.5: Definition des Nichtterminals für die Spezifikation des `equations`-Blockes

Als Letztes wird die grammatikalische Umsetzung des `updates`-Blockes vorgestellt (vgl. Abbildung 6.6). Dieser Block wird durch ein Schlüsselwort `update` in der konkreten Syntax eines NESTML-Modells eingeleitet. Der Rumpf des Blockes befindet sich zwischen den Schlüsselwörtern `:` `end`. Die innere Struktur des Blockes wird vollständig mithilfe der eingebetteten `Block`-Produktion aus der `Procedural`-Grammatik definiert. Die `Block`-Produktion wird in Unterabschnitt 6.1.2 näher erläutert.

Die NESTML-Grammatik ist so konzipiert, dass in der Grammatik nur die Struktur des Neurons spezifiziert wird. Der Inhalt der unterschiedlichen Blöcke kommt erst aufgrund der Einbettung anderer Grammatiken zustande. Somit bleibt die NESTML Modellierungssprache sehr modular und flexibel. Einerseits können die einzelnen Gram-

```
1 Dynamics implements BodyElement =
2     "update"
3     BLOCK_OPEN
4       Block
5     BLOCK_CLOSE;
```


Abbildung 6.6: Definition des Nichtterminals für die Spezifikation des `update`-Blockes

matiken so unabhängig von der NESTML-Grammatik entwickelt werden, andererseits kann die NESTML-Grammatik dadurch an die Einbettung anderer bzw. neuer Sprachen angepasst werden. Dadurch ergeben sich kleinere und modulare Grammatiken, die besser wiederverwendet werden können. Die vollständige `NESTML`-Grammatik ist in Listing D.5 zu finden.

6.1.2 Procedural-Grammatik

In diesem Abschnitt wird die Grammatikstruktur der *ProceduralDSL* vorgestellt. Dabei wird eine vereinfachte Version der Grammatik benutzt, die die wesentlichen Eigenschaften der zugrunde liegenden vollständigen Grammatik erklärt. Die vollständige Grammatikdefinition ist in Listing D.4 zu finden.

Die Hauptaufgabe der *ProceduralDSL* ist es, eine Möglichkeit zur Verfügung zu stellen, eine Folge von imperativen Anweisungen an vorgesehenen Stellen in einem Neuronenmodell zu spezifizieren. Um diese Anforderung zu erfüllen, wird die `Block`-Produktion auf der Grammatikebene bereitgestellt (vgl. Abbildung 6.7). Die `Block`-Produktion wird beispielsweise in der `NESTML`-Grammatik für die Spezifikation des Rumpfes von Methoden und der Neuronendynamik im `update`-Block benutzt.

```
1 Block = ( Stmt | NEWLINE )*;
2
3 Stmt = Small_Stmt | Compound_Stmt;
```


Abbildung 6.7: Produktionen für einen Block mit beliebigen Anweisungen.

Ähnlich zur `NESTML`-Grammatik werden auch in der `Procedural`-Grammatik Anweisungen mithilfe von Zeilenumbrüchen voneinander getrennt. Dies wird durch ein explizites Vorkommen der `NEWLINE`-Produktion erreicht (vgl. Zeile 1). Die Anweisungen sind in einfache Anweisungen (vgl. `Small_Stmt`) und zusammengesetzte Anweisungen (vgl. `Compound_Stmt`) unterteilt. Diese Unterscheidung ist notwendig, um den Aufbau der Scope-Hierarchie eleganter implementieren zu können. Alle `Compound_Stmt`-Elemente öffnen stets neue Gültigkeitsbereiche. Durch die Unterscheidung der Anweisungen kann der `NESTMLSymbolTableCreator` diesen jeweils in der entsprechenden `visit`-Methode erstellen.

Die `Small_Stmt`- und `Compound_Stmt`-Produktionen sind als `interface`-Produktionen

umgesetzt (vgl. Zeilen 1 und 7 in Abbildung 6.8), um eine Anpassung und Erweiterung in Subsprachen zu ermöglichen. Auf diese Weise können beide Typen von Anweisungen in Subsprachen erweitert werden.

MCG

```
1 interface Small_Stmt;
2 Assignment implements Small_Stmt = ... ;
3 FunctionCall implements Small_Stmt = ... ;
4 Declaration implements Small_Stmt = ... ;
5 ReturnStmt implements Small_Stmt = ... ;
6
7 interface Compound_Stmt;
8 IF_Stmt implements Compound_Stmt = ... ;
9 FOR_Stmt implements Compound_Stmt = ... ;
10 WHILE_Stmt implements Compound_Stmt = ...;
```

Abbildung 6.8: Zusammenfassung der Produktionen für alle Anweisungsarten.

Die Zuweisungsvarianten werden durch verschieden Alternativen innerhalb derselben Produktion modelliert. Abbildung 6.9 demonstriert diese Produktion. Auf der Ebene des Metamodells wird für jede Alternative ein `boolean`-Getter generiert (bspw. `isCompoundSum` für die `+=`-Zuweisung). Diese Vorgehensweise bietet eine bessere Handhabung der Klasse im Codegenerator. Im Vergleich zu einer `interface`-Produktion für die Modellierung einer Zuweisung gibt es einen Nachteil. Eine transparente Integration weiterer Arten von zusammengesetzten Zuweisungen (beispielsweise für andere Operatoren) ist in Subsprachen schwierig. Man kann dennoch die ganze Produktion in einer Subsprache überschrieben und auf diese Weise weitere Operatoren integrieren.

MCG

```
1 Assignment = lhsVarialbe:Variable
2     (assignment:["="]        |
3      compoundSum:["+="]      |
4      compoundMinus:["-="]    |
5      compoundProduct:["*="]  |
6      compoundQuotient:["/="]) Expr;
```

Abbildung 6.9: Produktion für einfache und zusammengesetzte Zuweisungen.

Die zusammengesetzten Anweisungen werden in Abbildung 6.10 exemplarisch am Beispiel der Produktion für `for`-Schleifen erläutert. Die Struktur der `FOR_Stmt`-Produktion ist darauf ausgelegt, im Falle eines Fehlers möglichst gute diagnostische Nachrichten generieren zu können. Die Zählervariable (vgl. `var:Name` in Zeile 1) wird durch eine einfache Namensreferenz eingeleitet. Die Korrektheit der Verwendung der referenzierten Variablen wird erst nach dem Parsen des Modells durch eine Kontextbedingung geprüft. Im Unterschied zu den Parser-Fehlern, erlaubt diese Prüfung es, aufschlussreichere Informationsnachrichten zu erzeugen, da der Kontext des Fehlers zu diesem Zeitpunkt bereits bekannt ist. Der Kontext kann dann als Teil der Fehlerbeschreibung ausgegeben werden.

Auch bei den beiden Intervallgrenzen (vgl. `from`- und `to`-Ausdrücke) werden beim Parsen beliebige Ausdrücke akzeptiert. Die Typkorrektheit der Ausdrücke wird erst nach dem Parsen geprüft.

```
1 FOR_Stmt = "for" var:Name "in" from:Expr "..." to:Expr                    MCG
2            ("step" step:SignedNumericLiteral)?
3            BLOCK_OPEN
4              Block
5            BLOCK_CLOSE;
```

Abbildung 6.10: Produktion für die Spezifikation von `for`-Schleifen.

Der Rumpf der `for`-Schleife befindet sich zwischen den Schlüsselwörtern `:` und `end`. Der Rumpf wird in Zeile 4 mithilfe der `Block`-Produktion spezifiziert. Auf der Ebene des Metamodells resultiert das im Composit-Muster [GHJV93]. Somit kann die `for`-Schleife beliebige *ProceduralDSL*-Anweisungen enthalten.

6.1.3 Equations-Grammatik

Abbildung 6.11 demonstriert die grammatikalische Umsetzung der *EquationsDSL*. Dabei fasst die `OdeDeclaration`-Produktion (vgl. Zeile 1) alle für die *EquationsDSL* relevanten Produktionen zusammen. Desweiteren werden Zeilenumbrüche in dieser Produktion explizit behandelt, da sie in NESTML nicht ignoriert werden.

```
1 OdeDeclaration  = (Equation | Shape | OdeFunction | NEWLINE)*;            MCG
2
3 OdeFunction = (["recordable"])? "function" Name Datatype "=" Expr;
4
5 Equation = lhs:Variable "=" rhs:Expr;
6                                          This production is defined
7 Shape = "shape" lhs:Variable "=" rhs:Expr;   in the Expressions-grammar
```

Abbildung 6.11: Produktionen für die Spezifikation von Funktionen, Differenzialgleichungen und `shape`-Funktionen im `equations`-Block.

Die übrigen Produktionen setzen die Sprachkonzepte der Funktionen (vgl. Zeile 3) Gleichungen (vgl. Zeile 5) und `shape`-Funktion (vgl. Zeile 7) um. Die `Equations`-Grammatik greift auf die eingebettete `Expr`-Produktion zurück, um die rechten Seiten der modellierten Elemente zu spezifizieren. Die `OdeDeclarations`-Produktion definiert den Anknüpfungspunkt für die Einbettung der *EquationsDSL* in andere Subsprachen. Die vollständige Grammatikdefinition ist in Listing D.3 zu finden.

6.1.4 Expressions-Grammatik

Das Ziel der *ExpressionsDSL* ist es, eine modulare und universelle Ausdruckssprache zu schaffen. Die *ExpressionsDSL* wird an den Stellen in der Neuronenspezifikation benutzt,

wo ein Ausdruck notwendig ist. Sie wird deshalb in die *EquationsDSL* und *ProzeduralDSL* eingebettet.

Abbildung 6.12 demonstriert einen Ausschnitt der `Expressions`-Grammatik. Die Grammatik beginnt mit der Definition von Termen, die als atomare Ausdrücke angesehen werden. Dazu zählen Literale aller Datentypen, Variablen und Funktionsaufrufe. Es werden sowohl Literale aus der MontiCore-Bibliothek benutzt als auch neue in der Grammatik definiert (vgl. die Zeilen 2 und 3). Alle Terme werden unter der `Term`-Produktion zusammengefasst (vgl. Zeile 5).

```
1 SILiteral = NumericLiteral ("["type:UnitType"]" | plainType:Name)?;
2 Variable = name:QualifiedName (order:"\'")*;
3 FunctionCall = name:QualifiedName "(" args:(Expr& || ",")* ")";
4
5 Term = FunctionCall | SILiteral | ["inf"] | Variable | StringLiteral;
6
7 Expr = ["("] p:Expr [")"]
8       | <rightassoc> base:Expr ["**"] exponent:Expr
9       | left:Expr (["*"] | ["/"] | ["%"]) right:Expr
10      | left:Expr (["+"] | ["-"]) right:Expr
11      | (uPlus:["+"] | uMinus:["-"] | uTilde:["~"]) u:Expr
12      | Term;
```

MCG

Abbildung 6.12: Produktion für die Terme und Ausdrücke der *ExpressionsDSL*.

Die rekursive Natur der Sprachdefinition der Ausdrücke wird grammatikalisch durch eine direkte Linksrekursion umgesetzt [ASU86]. Eine Grammatikregel ist direkt linksrekursiv, wenn sie der folgenden Form entspricht: $A \rightarrow A\alpha$. Dabei entspricht A einem beliebigen Nichtterminal gefolgt von einer beliebigen Folge von Terminalen und Nichtterminalen. Im vorliegenden Beispiel demonstriert die `Expr`-Produktion diese Eigenschaft. Beispielsweise sind die Alternativen in den Zeilen 8, 9 und 10 genau nach diesem Muster aufgebaut.

Hierbei spielt die Reihenfolge der Alternativen innerhalb derselben Produktion bei der Modellierung von Prioritäten der zugrundeliegenden Operatoren eine Rolle. Da die Produktion für den Multiplikationsoperator (vgl. Zeile 9) vor der Produktion für den Additionsoperator (vgl. Zeile 10) in der Produktion `Expr` vorkommt, bindet der Multiplikationsoperator stärker als der Additionsoperator. Dies hat zur Folge, dass der nicht geklammerte Ausdruck `a + b * c` wie folgt interpretiert wird: `a + (b * c)`.

Es ist zusätzlich möglich, die Assoziativität einer Regel explizit zu kontrollieren. Beispielsweise wird in der Alternative für den Operator der Potenzfunktion (vgl. Zeile 8) die Standardassoziativität verändert. Diese Alternative wird mit dem Schlüsselwort `<asso-cright>` markiert. Dadurch werden Ausdrücke der Form `a**b**c` wie folgt interpretiert: `a ** (b ** c)`. Ohne diese Markierung wäre der Ausdruck wie folgt interpretiert: `(a ** b) ** c`.

6.1.5 Units-Grammatik

Die in Abschnitt 5.6 beschriebene Aufteilung der Datentypen in primitive und physikalische Datentypen besteht auch in der grammatikalischen Umsetzung der Sprache. Alle primitiven Datentypen werden durch entsprechende Schlüsselwörter in der `Datatype`-Produktion aufgelistet. Diese Schlüsselwörter werden in eckige Klammern gesetzt, sodass für jedes der Schlüsselwörter eine boolesche `Getter`-Funktion erzeugt wird. Beispielsweise wird aufgrund des `string`-Schlüsselwortes die `isString`-Methode generiert. Die physikalischen Einheiten werden durch die `UnitType`-Produktion spezifiziert. Die rekursive Natur der Sprachdefinition von zusammengesetzten Einheiten wird grammatikalisch durch eine direkte Linksrekursion umgesetzt [ASU86].

```
1 Datatype = ["integer"] | ["real"] | ["string"] |
2            ["boolean"] | ["void"] |
3            UnitType;
4
5 UnitType = "(" UnitType ")"
7            | base:UnitType pow:["**"] exponent:IntLiteral
8            | left:UnitType (timesOp:["*"] | divOp:["/"]) right:UnitType
9            | unitlessLiteral:IntLiteral divOp:["/"] right:UnitType
10           | unit:Name;
```

Primitive datatypes

MCG

Abbildung 6.13: Produktionen für die Spezifikation von unterstützen primitiven und physikalischen Datentypen.

In den Zeilen 7 bis 8 werden der Multiplikationsoperator, der Divisionoperator und der Operator für die Potenzfunktion definiert. Nur diese Operationen dürfen zur Verknüpfung von einfachen Einheiten zu zusammengesetzten Einheiten verwendet werden. Die Grammatik stellt sicher, dass Argumente der Exponentialfuntion vorzeichenbehaftete Festkommazahlen sind. Die Einheiten können geklammert werden (vgl. Zeile 5). Zeile 9 führt die Möglichkeit ein, Typen der Art `unit**(-1)` in der Form `1/unit` zu spezifizieren. Die Typnamen werden auf der Grammatikebene durch Namen modelliert. Die semantische Korrektheit der verwendeten Datentypen wird wieder durch passende Kontextbedingungen sichergestellt. Beispielsweise wäre die folgende zusammengesetzte Einheit `1/string` zwar syntaktisch zulässig, sie wird an dem Modellierer dennoch als Fehler gemeldet. Zwar wäre es auch hier möglich, den Fehler bereits beim Aufbau des AST durch den Parser zu erkennen, die vom Parser produzierte Fehlerbeschreibung wäre jedoch schwer nachzuvollziehen. Aus diesem Grund wurde diese Prüfung ebenfalls als Kontextbedingung umgesetzt.

6.2 Symboltabelle von NESTML

Nach der Vorstellung der Grammatiken aller NESTML-Subsprachen wird in diesem Abschnitt der Aufbau der Symboltabelle von NESTML diskutiert. Wie in Kapitel 4 er-

läutert, spielen Bezeichner bzw. deren Repräsentation als Symbole in der MontiCore Workbench eine wichtige Rolle für das Prüfen von Kontextbedingungen und der Modellkomposition.

NESTML benutzt insbesondere die Möglichkeit der MontiCore Workbench, Sprachen aufzuteilen und mithilfe der Sprachvererbung und Spracheinbettung zu einer vereinten Sprache zusammenzufügen. Die semantische Verzahnung der einzelnen Subsprachen geschieht dann hauptsächlich mithilfe der Symboltabelle. Die Symboltabelleninfrastruktur, die das Auflösen der Bezeichner transparent übernimmt, ist in der Lage die Dekomposition von Subsprachen vor dem Sprachentwickler zu verbergen. Dabei regelt die Symboltabelleninfrastruktur eigenständig die Suche und Adaption von Symbolen, die aus unterschiedlichen Sprachen stammen können.

Die Symboltabelle von NESTML wird in drei sequenziell ablaufenden Phasen aufgebaut. Vor bzw. nach jeder dieser Phasen wird eine Menge von Kontextbedingungen geprüft, die die notwendige Konsistenz des vorliegenden Modells für den darauffolgenden Schritt sicherstellen.

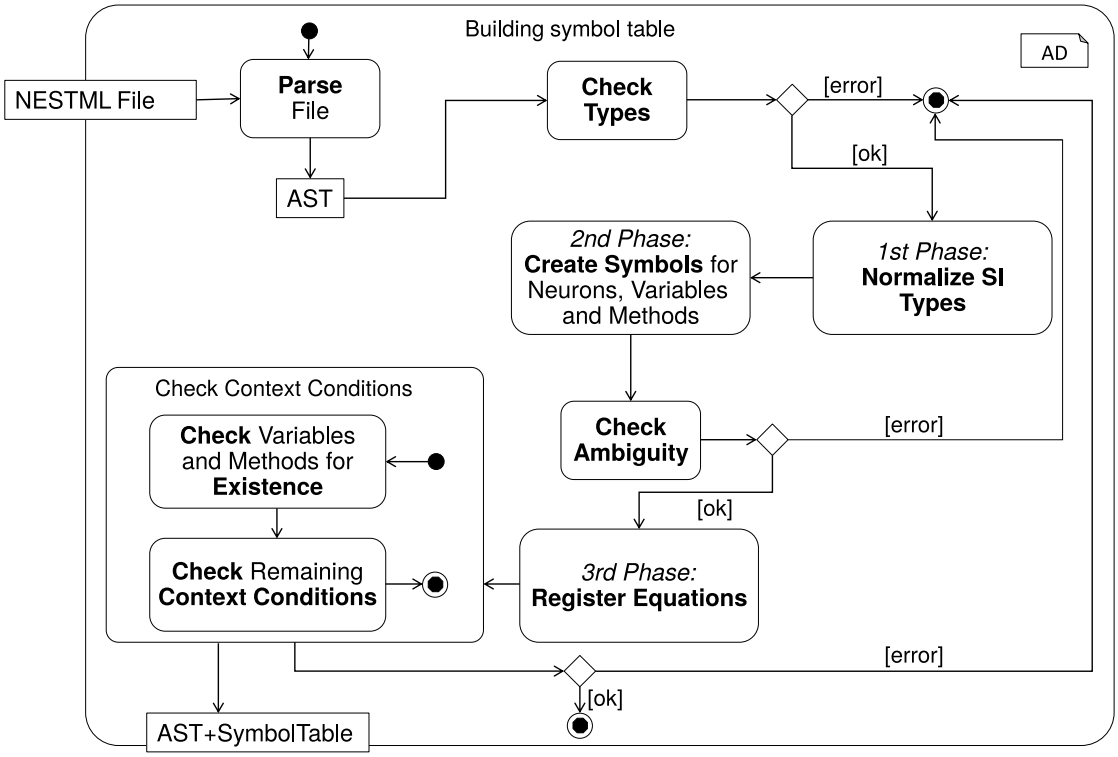


Abbildung 6.14: Schritte zum Aufbau der Symboltabelle aus einer NESTML-Datei.

Der Prozess ist in Abbildung 6.14 als Aktivitätsdiagramm abgebildet. Die einzelnen Phasen sind:

1. **Parse:** Eine im Filesystem gespeicherte NESTML-Datei wird durch den generierten NESTML-Parser verarbeitet. Der Parser instanziiert die Klassen des generierten Metamodells zu einem AST. Falls die NESTML-Datei syntaktische Fehler enthält, wird der Prozess mit einer Fehlermeldung abgebrochen.

2. **Check Types:** Alle Datentypen werden mithilfe eines speziellen Visitors besucht (vgl. Abschnitt 4.6). Für jedem Datentyp im Modell wird geprüft, ob dieser Typ ein valider Datentyp ist. Da die Korrektheit der benutzten Typen eine zwingende Voraussetzung für die darauffolgenden Schritte ist, wird der Prozess mit einer Fehlernachricht terminiert, falls ein undefinierter Typ im Modell vorkommt. Abschnitt 5.3 und Abschnitt 5.6 listen die unterstützten Datentypen auf.

3. **Normalize SI Types:** Alle im Modell benutzten physikalischen Einheiten werden in eine Normalform transformiert. Dabei wird das Modell wieder mit dem entsprechenden Visitor traversiert, der alle relevanten Knoten besucht, die eine Typreferenz enthalten (d.h. Deklarationen, Funktionsparameter und Einheitenliterale). Um die Konvertierung vorzunehmen, wird das in [BMP+16] vorgestellte Verfahren benutzt. Dabei werden die einfachen Einheiten direkt in die entsprechenden Signaturen konvertiert (vgl. Abbildung 6.16). Die zusammengesetzten Einheiten werden mit einem weiteren Visitor traversiert. Dabei werden die Operationen ausgewertet, die mit Einheiten verknüpft sind und eine resultierende Signatur berechnet.

4. **Register Symbols:** Wenn die vorherigen Schritte erfolgreich abgeschlossen wurden, werden für alle im Modell definierten Variablen und Methoden Symbole innerhalb der Symboltabelle erstellt und registriert.

5. **Check Unambiguity:** Die Eindeutigkeit und Existenz der referenzierten Variablen und Methoden ist eine zwingende Voraussetzung für die letzte Aufbauphase. Daher werden in diesem Arbeitsschritt Kontextbedingungen geprüft, die diese Konsistenz gewährleisten. Im Fehlerfall wird der Prozess des Symboltabellenaufbaus mit einer Fehlernachricht terminiert.

6. **Register Equations:** In dieser Aufbauphase der Symboltabelle findet eine Zuweisung der Differenzialgleichungen zu den entsprechenden Zustandsvariablen statt.

7. **Check Context Conditions:** Nach dem vollständigen und erfolgreichen Aufbau der Symboltabelle werden die Kontextbedingungen geprüft. Die Prüfungen werden in zwei sequenziellen Schritten ausgeführt. Zuerst werden alle Sprachkonstrukte geprüft, die eine Variable oder Methode referenzieren. Dabei wird eine Existenzprüfung durchgeführt, die gewährleistet, dass alle referenzierten Variablen und Methoden entweder im Modell selbst definiert sind oder es sich um eine vordefinierte Variable bzw. Methode handelt. Anschließend werden die restlichen

Prüfungen durchgeführt, die die Existenz dieser Modellelemente voraussetzen. Die vollständige Spezifikation der Kontextbedingungen ist in Abschnitt 7.3 zu finden.

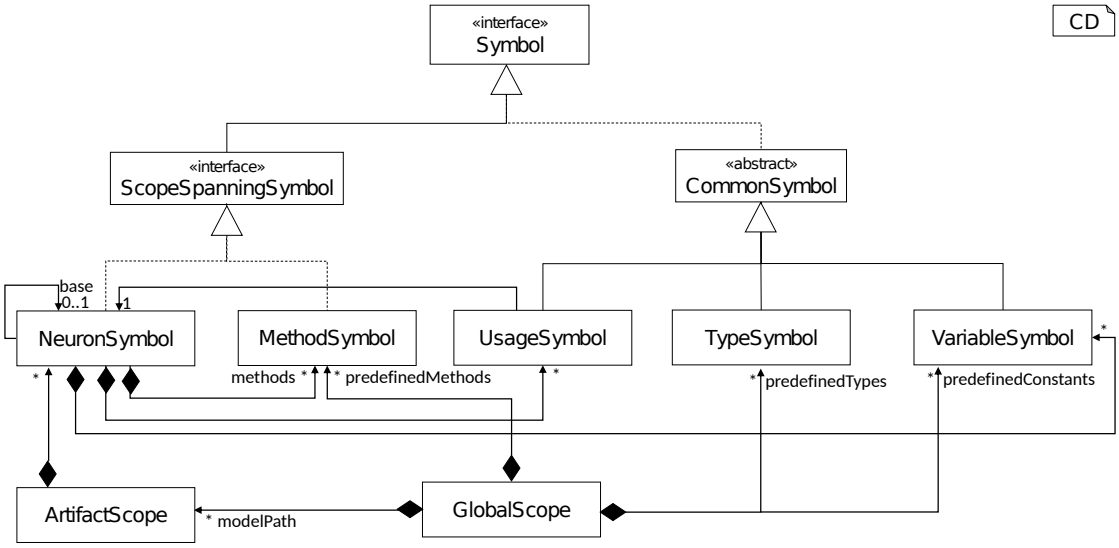


Abbildung 6.15: Überblick aller NESTML-Symbole und Scopes.

Abbildung 6.15 fasst die Symbole grafisch zusammen und visualisiert die Beziehungen der Symbole zueinander. Der `GlobalScope` verwaltet sowohl die NESTML-Dateien als auch alle vordefinierten Konstanten, Methoden und Datentypen. Der `GlobalScope` baut auch die Symboltabelle bei Bedarf automatisch auf. Das `NeuronSymbol` repräsentiert die Essenz des Neurons bzw. einer Komponente. Es besteht aus Symbolreferenzen zu Variablen, Methoden und anderen Neuronen. Die einzelnen Symbole hierbei sind:

TypeSymbol wird für alle vordefinierten Typen erstellt (vgl. Abschnitt 5.3 und Abschnitt 5.6). Ein `TypeSymbol` enthält neben dem Namen des modellierten Datentyps, auch die Information darüber, ob es sich um einen `UNIT`-, `PRIMITIVE`- oder `BUFFER`-Datentyp handelt. Ein `TypeSymbol` wird ausschließlich implizit während des Aufbaues der Symboltabelle instanziiert und im `GlobalScope` registriert.

VariableSymbol wird für jede Blockvariable, lokale Variable und jeden `input`-Port registriert. Während der Erzeugung der Symboltabelle werden die vordefinierten Variablen (beispielsweise `t` und `e`) als `VariableSymbol` im `GlobalScope` registriert. Ein `VariableSymbol` speichert auch die Information, in welchem Block die entsprechende Modellvariable definiert wurde. Dabei wird zwischen den folgenden Alternativen unterschieden: `STATE`, `PARAMETERS`, `INTERNALS`, `LOCALS`, `INPUT`. `LOCALS`-Symbole werden für Variablen erzeugt, die innerhalb einer Methode oder des `update`-Blockes definiert sind.

MethodSymbol wird für jede in einem Neuron oder in einer Komponente definierte Methode erstellt. Während der Erzeugung der Symboltabelle werden die vordefinierten Funktionen (beispielsweise `min` und `log`) als `MethodSymbol` im `GlobalScope` registriert. Neben dem Variablennamen wird die Signatur der Methode gespeichert. Die Signatur der Methode besteht aus den Namen und Typen der Argumente und dem Rückgabetyp.

NeuronSymbol wird für jedes Neuron bzw. jede Komponente erstellt. Es speichert die Information, ob es sich um ein Neuron oder eine Komponente handelt. Das `NeuronSymbol` stellt eine Schnittstelle bereit, um auf die im Neuron bzw. der Komponente modellierten Informationen zuzugreifen. Es verwaltet den Zugang zu den Blockvariablen, Methoden, `input`- und `output`-Ports.

UsageSymbol Dieses Symbol wird für jeden `use`-Block erzeugt. Mithilfe dieses Symbols wird eine Referenz zu einem anderen `NeuronSymbol` modelliert, das eine Komponente repräsentiert. Die Implementierung des Symbols ist nach dem Delegation-Muster aufgebaut [GHJV93, Gra03]. Das `UsageSymbol` delegiert alle Aufrufe an das Delegate-Symbol, das im `use`-Block referenziert ist.

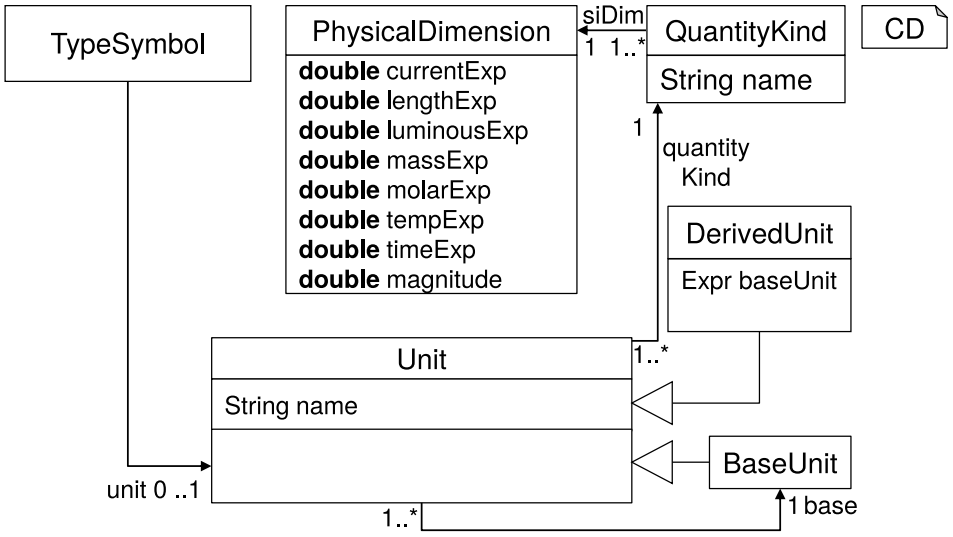


Abbildung 6.16: Überblick des Datenmodells für die Repräsentation der physikalischen Einheiten. Die Abbildung ist nach [BMP+16] adaptiert.

Das in Abbildung 6.16 dargestellte Datenmodell wurde entwickelt, um die Informationen über die physikalischen Einheiten zu speichern. Das Datenmodell unterscheidet zwischen einfachen und zusammengesetzten Einheiten. Die `PhysicalDimensions`-Klasse speichert die Signatur des physikalischen Typs als ein 8-Tupel von Gleitkommazahlen.

Damit werden alle einfachen Einheiten modelliert. Für zusammengesetzte Einheiten wird zusätzlich der definierende Ausdruck gespeichert damit die Nachvollziehbarkeit der Einheitendefinition erhalten bleibt.

Die `QuanityKind`-Klasse wird benutzt, um die Einheit eindeutig zu spezifizieren. Sie ist notwendig, da im SI-Einheitensystem unterschiedliche Einheiten existieren, die dieselbe Signatur teilen (z.B. das Drehmoment und die Energie). Anhand der Signatur und des Namens können Einheiten eindeutig identifiziert werden.

Die MontiCore Workbench bietet ein transparentes Verfahren, um die Geltungsbereiche von Variablen als eine hierarchische Scope-Struktur zu verwalten. Für jede NESTML-Datei wird ein `AritfaktScope` erstellt. Neue Scopes innerhalb des `ArtifactScope` werden automatisch aufgespannt, sobald ein `NeuronSymbol` oder `MethodSymbol` beim Aufbau der Symboltabelle erstellt wird. Dies wird dadurch erreicht, dass die beiden Symbole ein vordefiniertes Interface `ScopeSpanningSymbol` implementieren.

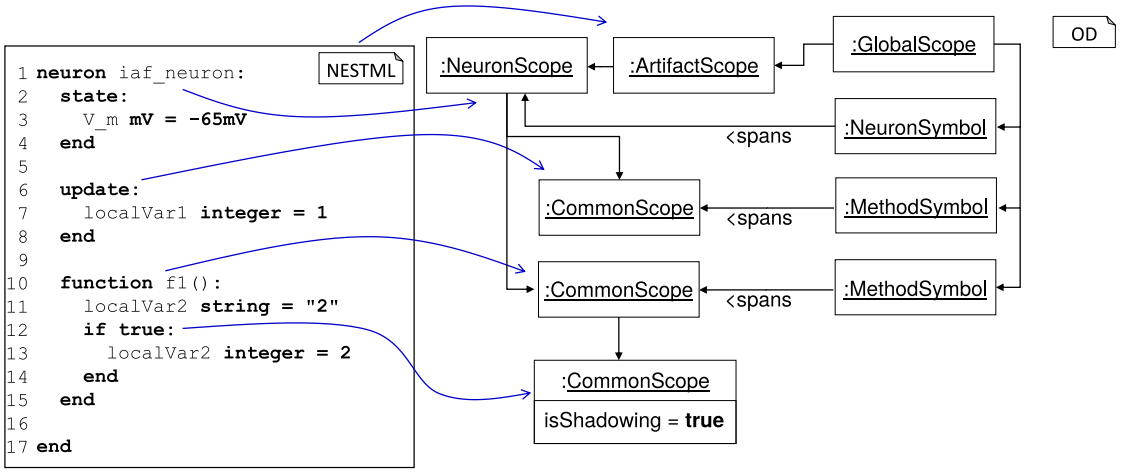
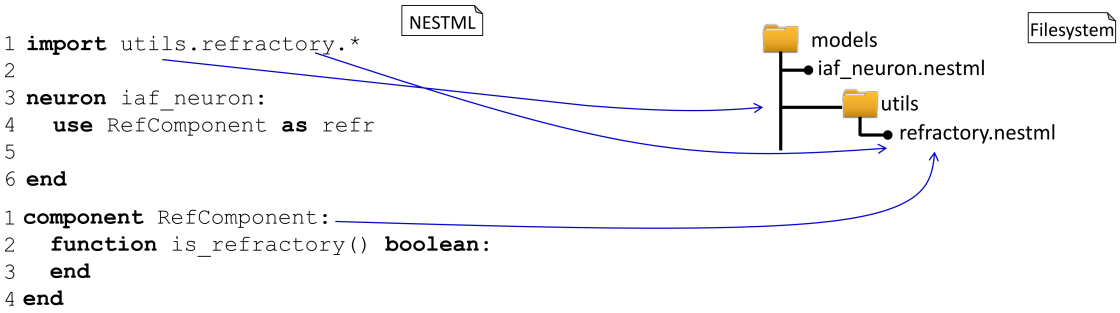


Abbildung 6.17: Abbildung der Geltungsbereiche von Variablen auf die Scope-Hierarchie.

Desweiteren spielen die Scopes bei Methoden und im `update`-Block eine wichtige Rolle. Dabei wird ein Scope immer dann aufgespannt, wenn eine zusammengesetzte Anweisung der *ProceduralDSL* vorkommt. Innerhalb dieses Scopes können Variablen aus übergeordneten Scopes überschrieben werden. Diese Funktionalität wird von der MontiCore Workbench bereitgestellt, wenn beim Erzeugen des Scopes der `isShadowing`-Parameter auf den Wert `true` gesetzt wird. Abbildung 6.17 demonstriert das erläuterte Verfahren an einem Beispiel. Alle Methoden und der `update`-Block spannen stets einen Scope auf, der im Scope des definierenden Neuron enthalten ist. Innerhalb dieser Scopes können Variablen verdeckt werden. Beispielsweise wird die Variable `localVar2` innerhalb des Scopes definiert, der aufgrund der `if`-Anweisung aufgespannt wird und verdeckt die

entsprechende Variable aus dem übergeordneten Scope der `f1`-Methode.




```
1 import utils.refractory.*
2
3 neuron iaf_neuron:
4   use RefComponent as refr
5
6 end
```

```
1 component RefComponent:
2   function is_refractory() boolean:
3   end
4 end
```

Abbildung 6.18: Funktionsweise des Import-Mechanismus ausgehend vom Modellpfad `models`.

Um Komponenten aus anderen Dateien zu importieren, wird für das Auflösen der Importanweisungen der Ansatz aus Python adaptiert. Für das Auflösen der importierten Elemente spielen die Namen der Dateien und Verzeichnisse, in denen das zu importierende Element gespeichert ist, eine entscheidende Rolle. Das Basisverzeichnis, in dem NESTML-Dateien gespeichert sind, wird als Modellpfad (engl. Modelpath) bezeichnet. Alle Import-Anweisungen, die innerhalb einer Datei spezifiziert sind, werden relativ zm Modellpfad interpretiert.

Abbildung 6.18 demonstriert den Mechanismus am konkreten Beispiel. Dabei ist der Modellpfad auf ein Verzeichnis `models` gesetzt. Mit der `import`-Anweisung in Zeile 1 werden alle Elemente importiet, die in der Datei `refractory.nestml` im Unterverzeichnis `utils` liegt. Durch den Import steht `RefComponent` in der importierenden Datei zur Verfügung.

Auf Modellebene ist es nicht nötig, die Paketinformation explizit zu deklarieren. Stattdessen wird die Paketspezifikation im Laufe einer Vortransformation erzänzt, die direkt nach dem Parsen des Modells ausgeführt wird. Im Allgemeinen wird die Information wie folgt berechnet: Ausgehend vom Modellpfad wird der relative Pfad zur Datei bestimmt indem der relative Pfad wird mit dem Dateinamen verknüpft wird.

Abbildung 6.19 demonstriert die Bestimmung der Paketdeklaration an einem Beispiel. Ausgehend vom Modellpfad `models` ist der relative Pfad der `neurons`-Datei leer. Daher besteht der Paketname nur aus dem Dateinamen.

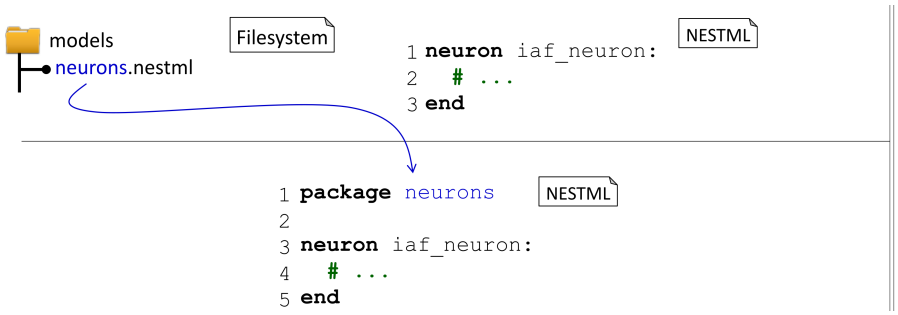


Abbildung 6.19: Exemplarische Ergänzung der Paketinformation anhand des Dateinamens der NESTML-Datei mithilfe der Modelltransformation.

Kapitel 7

Methodik für die Entwicklung neuer Neuronen mit NESTML

In diesem Kapitel wird die Verwendung von NESTML in Form einer detaillierten Anleitung vorgestellt. Die Anleitung demonstriert den Prozess der Neuronenmodellbildung mit NESTML. Die entwickelten Modelle können anschließend in NEST für eine neuronale Simulation verwendet werden. Während der Vorstellung von verschiedenen Neuronenmodellen werden die wichtigen Eigenschaften von NESTML an konkreten Beispielen demonstriert. Die Verwendung des NESTML Sprachwerkzeugs wird anhand der Beispiele schrittweise erläutert. Anschließend werden alle Kontextbedingungen von NESTML ausführlich diskutiert. Sie gewährleisten die semantische Korrektheit von Neuronenmodellen in NESTML.

Diese Anleitung wird in englischer Sprache vorgestellt, da sie in dieser Form auch für die Evaluierung des NESTML-Ansatzes verwendet wurde.

7.1 Developing biological neuron models with NESTML

Throughout this tutorial, a set of increasingly more complex neuron models will be developed. Each of these example models serves to motivate and explain the different language concepts for modeling different aspects of point neurons in NESTML.

7.1.1 Derivation of a mathematical model for biological neurons

Over time researchers in computational neuroscience created a multitude of biological neuron models. This variety ranges from precise 3d reconstructions of real neurons over multi-compartment neurons to simplified point neurons in which the morphology is collapsed into a single point. Figure 7.1 shows representative examples of models with different levels of detail.

Multi-compartment models promise better insights into the biophysical processes in a single biological neuron, but there are only a few effects on the network scale, which cannot be reproduced using simple point neurons. Therefore, the primary focus of NESTML is to provide a solid basis for modeling point neurons. The simulation of a point neuron

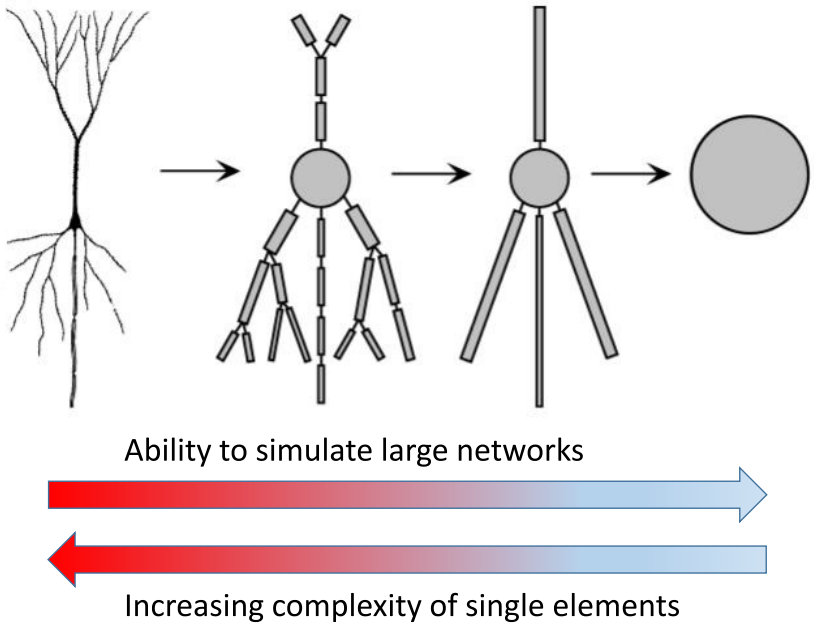


Figure 7.1: Different level of details for neuron models. Starting from a cortical pyramidal neuron, the level of detail reduces towards a point neuron model. Adapted from [Epp10] and [GKNP14].

requires less computational resources and memory compared to more detailed types of neuron models. Using point neurons thus enables researchers to simulate bigger networks, albeit at a lower level of detail.

Before proceeding to the introduction of the NESTML language itself, a simple derivation of a biological neuron model will be given first in order to introduce the basic terms and concepts. Figure 7.2 explains the process of building a biological neuron model based on an equivalent electrical circuit. The membrane of the nerve cell consists of a bi-lipid layer and can be regarded as an isolator separaring the inside from the outside. It is largely impermeable for ions and larger molecules. Active ion pumps and passive channels are built into the membrane and allow the selective passage of certain ions. Through the active transport of ions the neuron creates and maintains an electrical potential across the membrane. This potential is called the neuron's resting potential (cf. Figure 7.2 (A) and (B)).

The canonical approach for creating a computational model of a neuron is to build an electrical circuit (cf. panel (C)). Due to the separation of charges, the membrane can be represented by a capacitor, the membrane potential corresponds to the voltage over the capacitor. Different channels in the membrane transport ions into and out of the cell. These channels can be seen as resistors, which withstand the flow of current. External

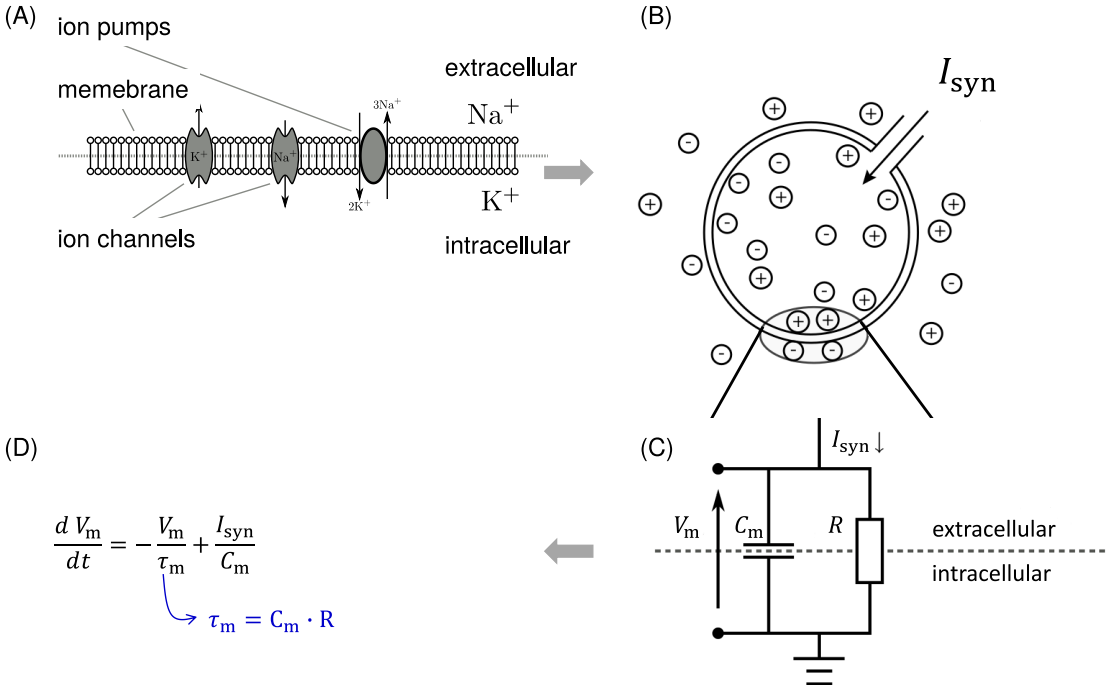


Figure 7.2: (A) Cross section of the cell membrane. (B) An abstract model of the membrane cross section. (C) An equivalent electrical circuit. (D) Differential equation that specifies the evolution of the membrane potential in the electrical circuit in (C).

stimulation of the neuron can be modelled as a source of current in the circuit. The synaptic current I_{syn} thus may add further charge into the capacitor. Additional leak currents can be added to model the flow of ions out of the cell through channels in the membrane.

The temporal dynamics of the RC circuit and thus of the membrane potential can be described by the following differential equation:

$$\frac{d}{dt}V_{\text{m}} = -\frac{V_{\text{m}}}{C_{\text{m}} \cdot R} + \frac{I_{\text{syn}}}{C_{\text{m}}} \tag{7.1}$$

By substituting $C_{\text{m}} \cdot R$ by the time constant τ_{m} of the RC chain and the membrane, the equation is turned into its canonical form:

$$\frac{d}{dt}V_{\text{m}} = -\frac{V_{\text{m}}}{\tau_{\text{m}}} + \frac{I_{\text{syn}}}{C_{\text{m}}} \tag{7.2}$$

Parameters of the neuron model (e.g. resistance, resting potential, etc.) can be extracted from real neurons in neurobiological experiments. The synaptic current I_{syn}

can be chosen to be more or less complex to reflect certain properties of real biological neurons. Depending on their exact definition, the equation can be either solved exactly or by means of a numeric solver.

7.1.2 NEST Modeling Language

The mathematical foundation in the previous section allows to model biologically realistic point neurons. Based on this methodology, this section derives the corresponding representation for a model in NESTML.

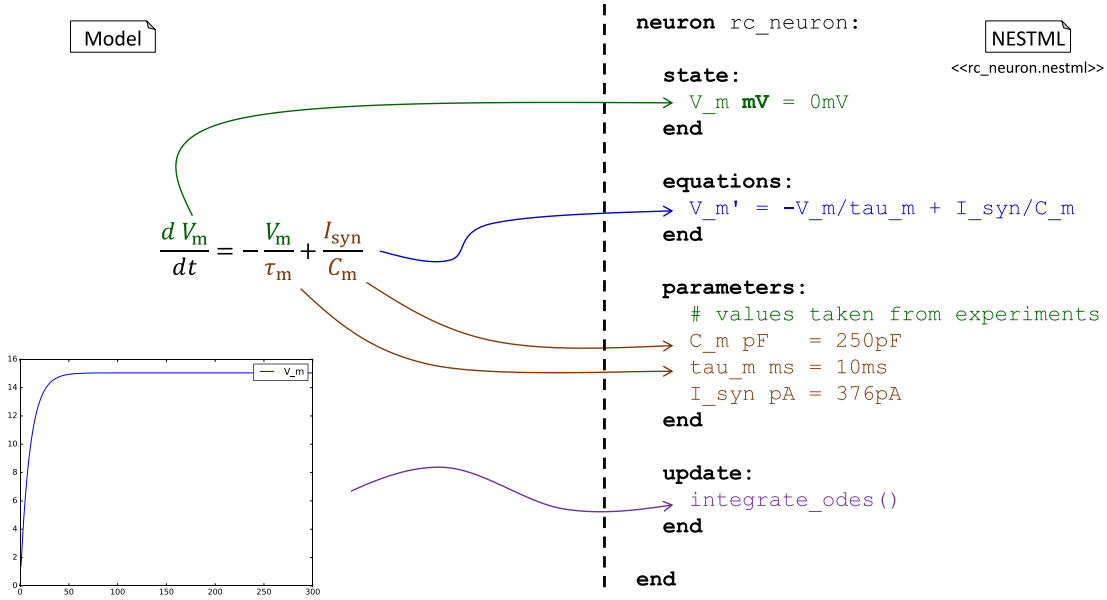


Figure 7.3: NESTML neuron model based on the mathematical formalism established in Figure 7.2 (D).

Figure 7.3 shows a direct mapping of the mathematical model to the corresponding model in NESTML. The depicted neuron named `rc_neuron` has a body which consists of different blocks where variables, equations, and the dynamic behavior are defined. The body of every block (including the `neuron`-block) starts with a colon character and ends with the `end` keyword.

Different blocks with variables exist to reflect the different semantics of the corresponding variables. The `state`-block contains variables which describe the temporally changing state of the neuron. `state`-variables can be additionally specified through differential equations which are stated in the `equations`-block. This is not possible for variables in other blocks. A differential equation is marked by a variable name followed by a non-empty list of `'`-characters. The `parameters`-block contains variables which

remain constant during the simulation but can vary among different neuron instances or simulations. Different parametrizations can be used to create a different spiking behavior using the same neuron model [JLG04]. The support for parametrizing neurons is thus an important feature of NESTML.

In general, all variable blocks are composed of variable declarations. A variable declaration consists of a non-empty list of variable names followed by the type of the variables. A type can be either a primitive datatype or a physical unit. Physical units can be complex units which are composed of units using multiplications, divisions, and powers: e.g. `1/ms` or `nS/ms**2`, which corresponds to $\frac{nS}{ms^2}$. Supported primitive types and physical units are listed in section 5.3 and section 5.6. Declarations end with an optional initialization expression.

The `update`-block enables a fine-grained specification of the neuron's temporal behavior. In the shown example, it only states the propagation of the equation in the `equation`-block by calling NESTML's predefined function `integrate_odes`. NESTML doesn't require that statements are concluded with a semicolon character (`;`). The NESTML parser instead uses the `newline`-whitespace character for this purpose similar to what is done in Python. Finally, NESTML models can be documented using comments which start after the `#`-character.

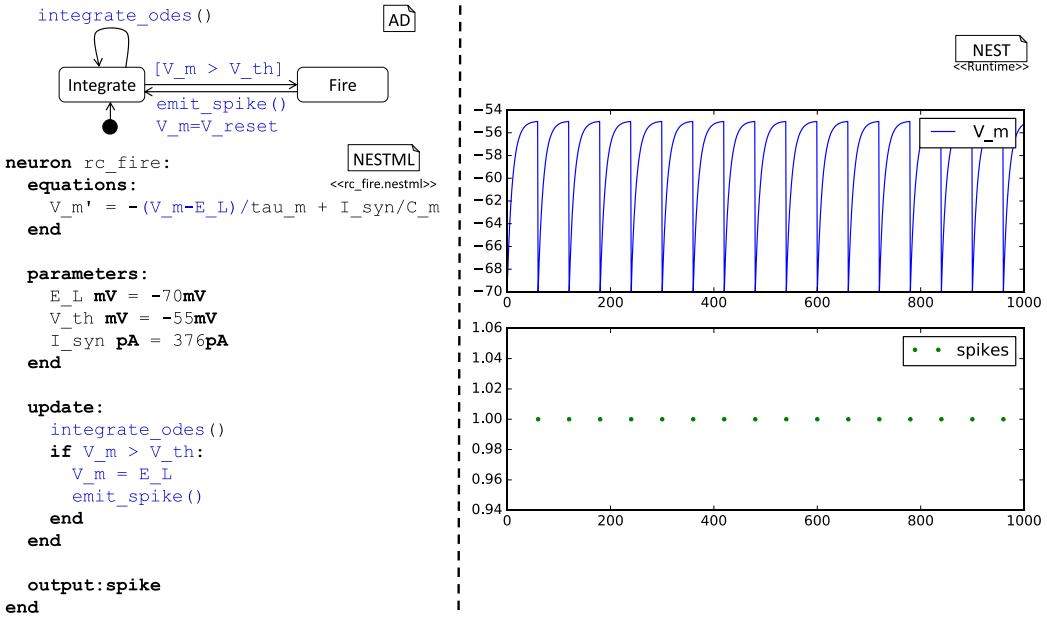


Figure 7.4: The depicted neuron defines the variable `V_m` for the membrane potential to always be relative to the resting membrane potential `E_L`. The `update`-block implements a simple subthreshold dynamics which is specified as an activity diagram.

The neuron model defined in Figure 7.3 shows two major differences to a real biological neuron: first, it always has a strictly positive membrane potential, which converges to `0mV` without external input. Second, the neuron isn't able to fire spikes, since this functionality is not implemented in the `update`-block.

In order to shift the membrane potential to the physiological range, the `V_m` variable is replaced by the term `V_m - E_L` in the differential equation. The variable `E_L` models the resting potential of the neuron, e.g. the value the membrane potential relaxes to in absence of external input. Figure 7.4 exemplifiex this approach by setting the membrane potential initially to the value of `E_L`. The differential equation is now expressed in terms of `V_m - E_L`. This means that negative values of `E_L` shift the membrane potential into the desired range. The example plot of the membrane potential on the right demonstrates this for `E_L=-70mV`.

The `update`-block implements a sub-threshold dynamics which integrates the membrane potential until the value of the threshold is reached. At this point in time a spike is fired and the membrane potential is reset to the resting potential. The predefined function `integrate_odes` propagates all differential equations in the `equations`-block for one time step. The fact that the example neuron should be able to fire spikes is encoded in the `output`-block. By calling the `emit_spike` method inside the `update`-block, the spike is actally fired and sent to all connected neurons.

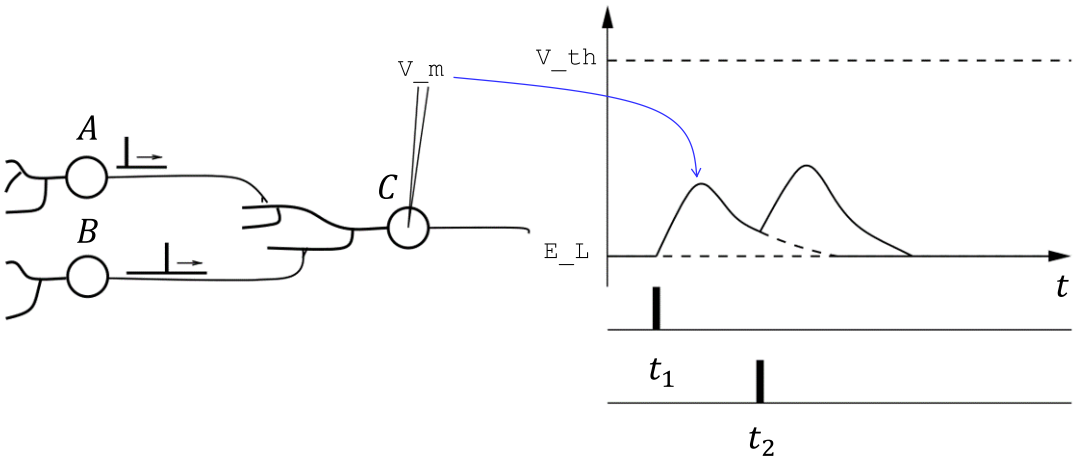


Figure 7.5: A post-synaptic neuron `C` receives input from two pre-synaptic neurons `A` and `B`. Each pre-synaptic spike evokes an excitatory postsynaptic potential in `A`. Adapted from [GKNP14].

Figure 7.5 shows a small neuronal network composed of two neurons `A` and `B` both connected to a third neuron `C`. In this constellation neurons `A` and `B` are called pre-synaptic neurons relative to the neuron `C`, which is called the post-synaptic neuron for `A` and `B`. Spikes which are fired by neurons `A` and `B` invoke an excitatory response in the

neuron `C`. In this example, neurons `A` and `B` are firing a spike at two different points of time, which leads to a positive excursion of the membrane potential of neuron `C`.

In order to support the neural network depicted in Figure 7.5, the neurons need to be able to process incoming spikes. This is enabled in the model by using an `input`-block. The `input`-block defines a list of named input ports with additional optional modifiers. In the case which is shown in Figure 7.6 the `input`-block contains only one port named `spikes`. The `spikes`-port is set up to receive inhibitory and excitatory inputs. In the context of NEST, this has the interpretation that incoming spikes with both negative and positive are routed to the port `spikes`. Alternatively two separate ports could be defined, e.g. `I_inh <- inhibitory spike` and `I_exc <- excitatory spike`. In the latter case, spikes with positive weight would be routed to the port `I_exc`, while spikes with negative weight would end up in the port `I_inh`.

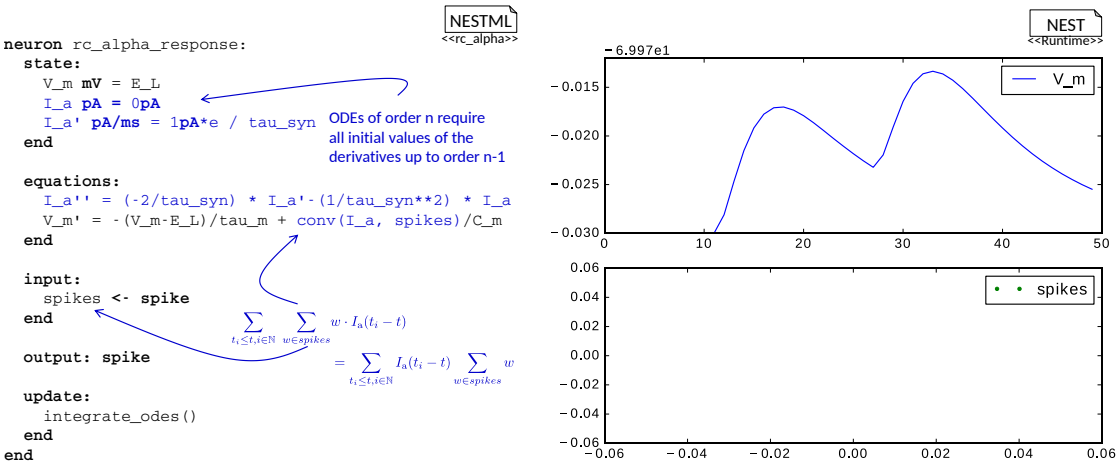


Figure 7.6: Modeling the post-synaptic response by convolving an α kernel with incoming spikes arriving at the named port `spikes`. The α kernel is modeled as a set of differential equations and an initial value.

The `conv`-function is used to create a temporally varying post-synaptic response. It expects the name of the port and the shape of the kernel as arguments. Mathematically, the `conv`-function performs the following computation:

$$conv(I\_a, spikes) = \sum_{t_i \leq t, i \in \mathbb{N}} \sum_{w \in spikes} w * I\_a(t_i - t) \tag{7.3}$$

One disadvantage of the previous approach for specifying the kernel is that the user must state both the differential equation and all initial values. To ease this often-needed operation, NESTML offers a more convenient way to model kernel functions. Figure 7.7 demonstrates this functionality with the example of the same α-shaped kernel that was

defined in the previous example as a system of differential equations. This time the kernel is specified as a function of t. It is defined with the `shape` keyword after which the kernel function is explicitly defined as an analytical function.

```
neuron rc_alpha_response_shape:

  equations:
    shape I_a=(e/tau_syn) * t * exp(-t/tau_syn)
    V_m' = -(V_m-E_L)/tau_m + conv(I_a, spikes)/C_m
  end

end
```

NESTML
<<rc_shape>>

Figure 7.7: Convolution of incoming spikes with an α kernel with incoming spikes which arrive via the named port `spikes`. The α kernel is modeled as a function of the time variable t.

`shape` functions describe the shape of a post-synaptic response. They are functions of `t`, where `t` is an implicitly defined variable that represents the current time of the simulation. The function `I_a(t)` in the example multiplies incoming spike weights `w` from the `spikes` port to compose the synaptic input. The advantage of this notation is the increased expressiveness of the model since the shape function is modeled explicitly. It is now also possible to derive all initial values automatically.

7.2 Installation and usage of the NESTML environment

The NESTML runtime environment currently requires a complex software setup composed of different software modules in specific versions. Since some of these modules are not purely Java-based, they don't run in a Java Runtime Environment. In addition, the Java Runtime Environment must be available in the latest version, which is not always the case on the host system.

To ease usage, NESTML is packaged as a command line script backed by a Docker container which encapsulates the software stack. The Docker container and bundled management script provides a concise console API to work with NESTML. Users thus only have to download the sources from the GitHub repository and install a Docker environment in order to be able to execute NESTML on their local computer.

The first step for installing NESTML on the local computer is to clone the GitHub repository [PBE+17a]. For all further explanations, the folder `<nestml_clone>` is assumed to be the local copy of this repository. Second, the latest version of Docker[1] must be installed, which is often available from the package management system of current Linux distributions. In order to execute the Docker commands without root permissions

[1] https://www.docker.com/

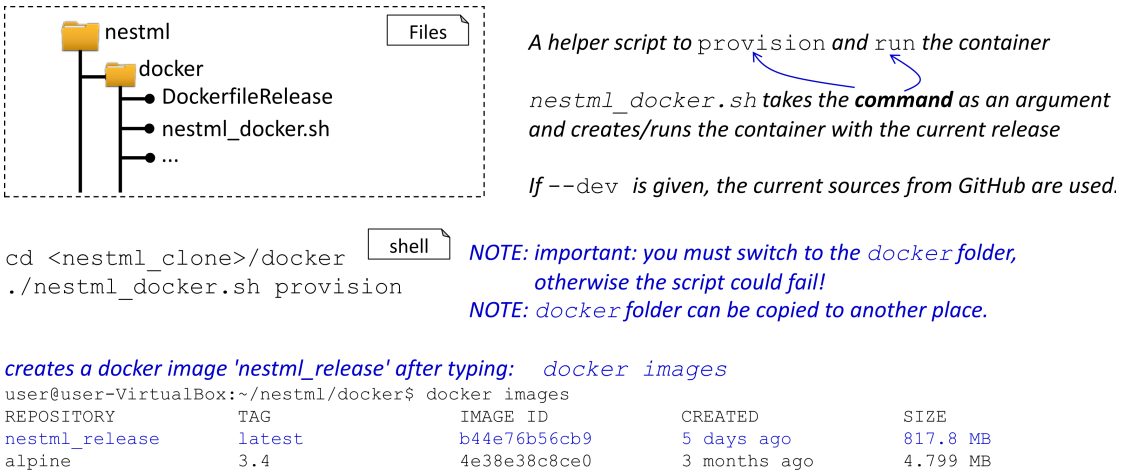


Figure 7.8: Installing the NESTML environment on the local machine.

and access the files produced by NESTML, the user must be added to the docker group[2].

To conveniently install and run NESTML, a management script called `docker_nestml.sh` is provided. It can be found in the `<nestml_clone>/docker` folder. To set up the local Docker container, the following command has to be executed in the console:

```
1 docker_nestml.sh provision
```

This step creates the container including all modules needed for NESTML's runtime. In the default case the latest released NESTML version is downloaded. To build the current development version of NESTML from sources, an additional argument `--dev` has to be provided to the installation command:

```
1 docker_nestml.sh provision --dev
```

The `docker images` command can be used to check if the Docker container was successfully created. If everything went well it should list the newly created `nestml_release` container (or `nestml_development` in the case the development option was set). The lower part of Figure 7.8 which summarizes all steps required for the NESTML-installation.

After a successful installation of the NESTML Docker container, the `docker_nestml.sh` script can be used to execute NESTML. The following command runs NESTML for all models in the given folder:

```
1 docker_nestml.sh run $path_to_models
```

[2] https://docs.docker.com/engine/installation/linux/ubuntulinux/#/manage-docker-as-a-non-root-user

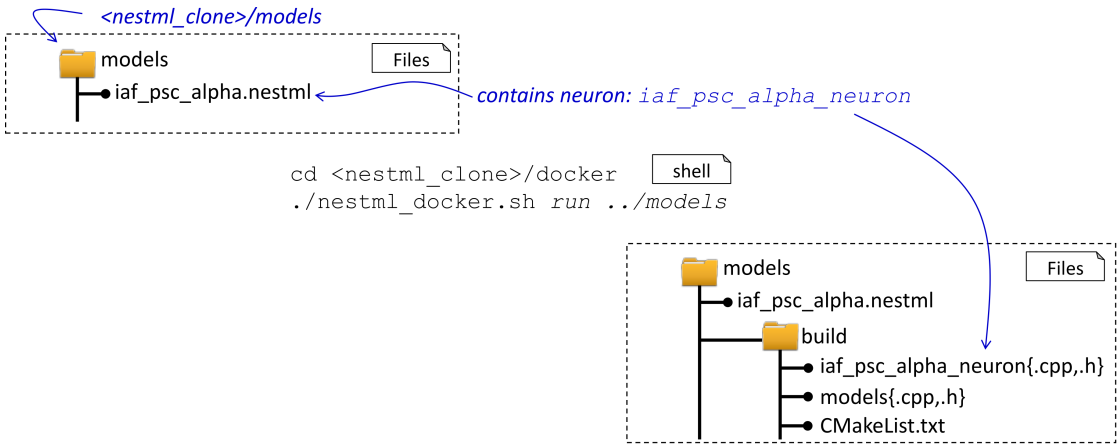


Figure 7.9: Execution of the NESTML tool on the `models`-folder. A successful run produces a set of C++ and CMake files in the subfolder `build`.

NESTML assumes that all neuron models are located in a dedicated folder `$path_to_models`. The name of this folder also becomes the name of the resulting NEST module. The output of the script invocation is a set of C++ and CMake files which are written to the subfolder `build` under the input folder. Figure 7.9 shows the whole process with the example of the models which are provided as a part of the NESTML distribution.

The code generated in the previous step can be integrated into NEST. The NESTML code generation framework is decoupled from the particular NEST installation and runs entirely independently. This means that the neuron and module code can be created on a different system than the system where the module will be built and integrated into NEST. This is an important feature, since NEST is often used on high-performance computer clusters without the possibility for a regular user to install Java or Docker. In such a situation, the generated code is simply transferred to the target computer and compiled and used there.

NEST offers an extension mechanism that allows to dynamically load new modules into the already installed simulator[3]. Figure 7.10 summarizes the steps for how to integrate NESTML models into NEST using this extension mechanism. First, the C++ and CMake code is generated from a set of NESTML models as described in the previous steps. In order to compile and install the module, a NEST installation and its C++ header files are required. The path to the installation is set with the environment variable `NEST_INSTALL_DIR`. After switching to the `build`-folder that contains the generated files from the previous step, the NEST extension module can be configured, compiled and installed with the following commands:

[3] http://nest.github.io/nest-simulator/extension_modules

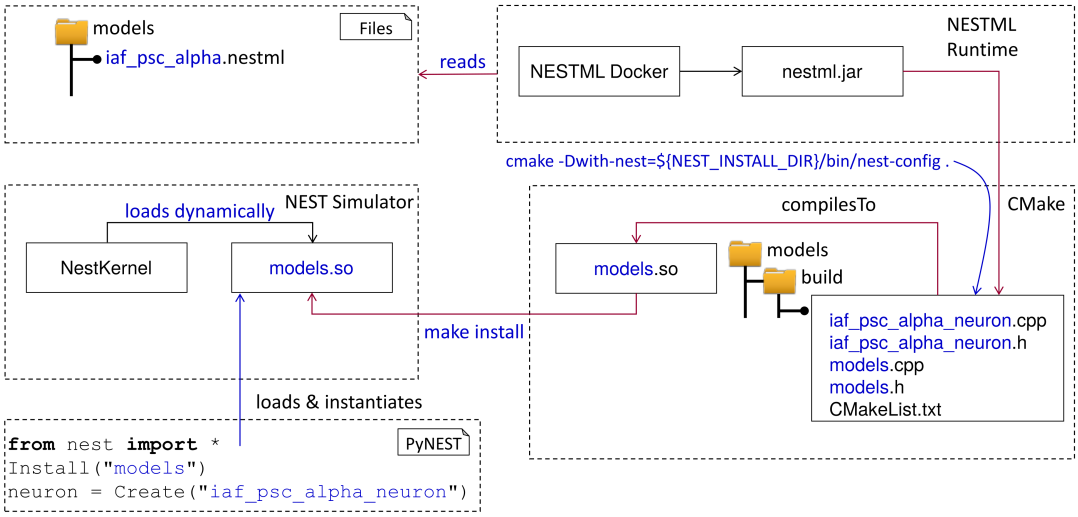


Figure 7.10: Generation and installation of a NEST extension module from a set of NESTML models.

```
1 cmake -Dwith-nest=${NEST_INSTALL_DIR}/bin/nest-config .
2 make
3 make install
```

After executing these steps, the binary library `models.so` is available in NEST's module directory and ready for use. The extension module contains all neurons from the `models` folder. The PyNEST script in Figure 7.10 demonstrates the usage of the module. First, the module `models` must be loaded using a call to the `Install` function in the PyNEST script. It makes all which are part of the module available for simulations. The `Create` function is used to instantiate the neurons which are referenced by their names as defined in the source NESTML files.

Figure 7.11 summarizes the PyNEST API for neurons. As already mentioned the name of the extension module corresponds to the name of the folder where models were stored, e.g. `models`. The `Create`-function instantiates a neuron. Values of all variables from the `state` and `parameters` blocks can be set through the `SetStatus` function. Their current value can be retrieved using the function `GetStatus`.

All variables from the `state` block are marked recordable and can be accessed from different devices, e.g. the `multimeter`. In the example, the values of the `V_m` variable is recorded with a `multimeter` throughout the simulation. This way the trace of the

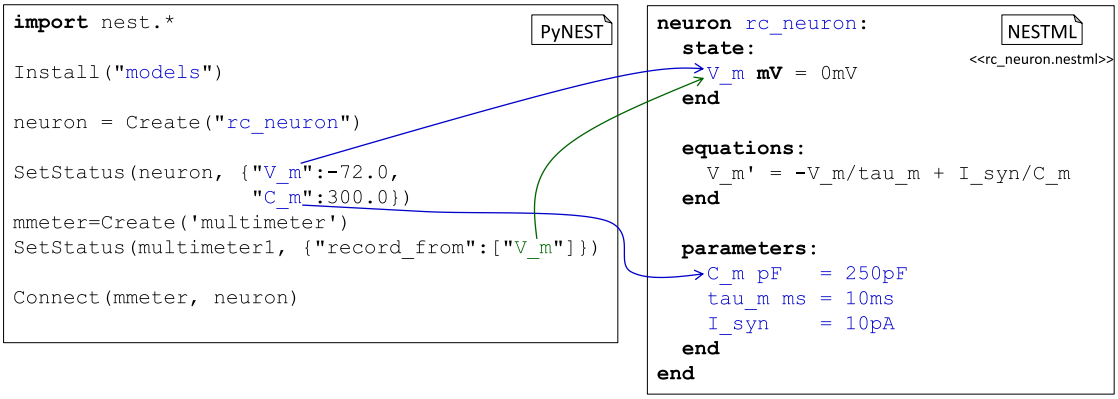


Figure 7.11: Using NESTML models trough the PyNEST API

membrane potential can be plotted and analyzed after the simulation.

7.3 Semantic checks of NESTML neurons and components

NESTML catches a lot of potential modeling errors and gives early feedback on the model specification in order to speed up the model creation. This section describes all errors which are identified by NESTML. The error messages are grouped by the sublanguage in which the error occurs (cf. chapter 5).

7.3.1 Parse errors

Parse error messages issued by the generated MontiCore parser are often rather hard to understand due to the fact that the AST cannot be constructed in case of an error in this stage. To ease debugging, NESTML provides the entire failing line from the source model in addition to the plain error description. Figure 7.12 demonstrates this behavior with a little example. The neuron cannot be parsed due to the misspelled keyword `neuron`. The position of the error as identified by the parser is reported as part of the error message.

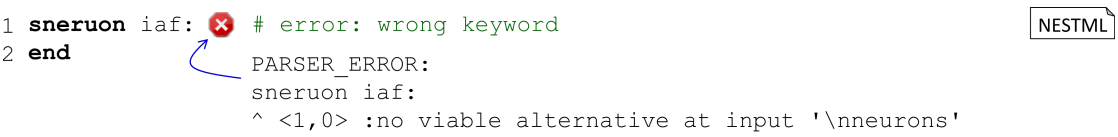


Figure 7.12: Parse error due to a misspelled keyword `neuron`.

7.3.2 Semantic errors in the ProceduralDSL

This sections describes errors that can occur in the *ProceduralDSL* code that is used for the implementation of the `update` and `function` blocks.

`CODE_AFTER_RETURN:` There are unreachable statements in the model. Statements which are located after a return statement are never executed and therefore have no effect on the behavior on the model behavior. Often this hints at a misplaced `return` statement.

In general, detecting dead code is a complex task. However for a set of trivial and common errors such checks can be easily performed. These are implemented in this context condition.

Figure 7.13: `CODE_AFTER_RETURN:` A program with two statements which are unreachable. Since all branches of the `if`-statement have a return statement, the `emit_spike` function call in line 6 and a return statement in line 7 will be never executed.

`FUNCTION_DOES_NOT_EXIST:` A function call used in an expression or statement is undefined. Eeach function referenced in an expression or statement must be either a predefined function (cf. section 5.3) or a function defined by the user.

`ILLEGAL_EXPRESSION:` The type of the expression is not compatible with the expected type. NESTML is strongly and statically typed and types are checked during model analysis. All type violations are reported as errors. This context condition checks that

- the type of the declaration is compatible with the type of the initialization expression
- the type of the right-hand side of an assignment is compatible with the type of its left-hand side
- types of function arguments are compatible with the types in the function signature
- `if`-statements and `while`-loops have a boolean condition

- iterator variables and upper/lower bounds in `for`-loops have numeric type.

```
1 var real = "Bob"   # A real cannot be initialized with
2                    # string an expression of the type
3 if var:  # if statement expects a boolean condition,
4          # var is real.
5 end
6 pow("bob", "alice")  # pow is defined for numeric arguments
```

ProceduralDSL

Figure 7.14: `ILLEGAL_EXPRESSION`: Type mismatch errors during the declaration of a variable, an `if`-statement, and a function call. In line 1 a variable of type `real` is initialized with an expression of the incombativle type `string`. Line 3 shows that `var` cannot be used as condition for the `if` condition, since its type is `real` instead of `boolean`. The function `pow` in line 6 expects two parameter of type `real`, while two `string` typed variables are given.

`VARIABLE_EXISTS_MULTIPLE_TIMES`: A variable name must be unique in every scope (cf. section 6.2). A new scope is opened by each of the blocks in NESTML and every variable in every scope must have a unique name. While it is possible to redefine variables defined in an enclosing scope it is not possible to define two variables with the same name in a single scope.

```
1 var1 integer
2 var1 integer  # var1 is already defined in line 1
3 var2 integer
4 if true:
5   var2 real # OK: var2 integer is overloaded
6 end
```

ProceduralDSL

Figure 7.15: `VARIABLE_EXISTS_MULTIPLE_TIMES`: `var1` (defined in line 1) is defined a second time within the same scope in line 2 which leads to an error. In contrast, `var2` (defined in line 3) is redefined in a subscope in line 5 which is allowed in NESTML.

`VARIABLE_HAS_TYPE_NAME`: In order to reduce the ambiguity of neuron models, NESTML forbids the usage of type names as variable names. This also holds for physical units. The declaration **V mV** is thus invalid, since the name **V** is used for the physical unit **Volt**. The list of NESTML types can be found in section 5.3 and section 5.6.

`VARIABLE_DOES_NOT_EXIST`: Every variable which is used in an expression or as a part of a compound statement must be defined.

`VARIABLE_NOT_DEFINED_BEFORE_USE`: A variable can be used in an expression or statement only after it was defined. This rule applies only to variables which are defined in an alogithmic way in an `update` or `function` block using the *ProceduralDSL*. Member variables from the *state*, *parameters* or *internals* blocks and predefined variables are conceptually always defined.

ProceduralDSL

```
1 var1 integer = var1  # Error: var1 is used in own definition
2 var2 integer = var3  # Error: var3 is defined in line 3
3 var3 integer = 0
```

Figure 7.16: `VARIABLE_NOT_DEFINED_BEFORE_USE`: In line 1 `var1` is used as part of its own initialization expression which is not allowed. The initialization expression of `var2` in line 2 uses a variable which is only defined in line 3 and thus results in an error.

7.3.3 Semantic errors in the neuron specification

This section describes semantic errors in the specification of neurons and components Technically they belong to the NESTML sub-language.

`BUFFER_NOT_ASSIGNABLE`: It not allowed to assign a value to an `input`-buffer although ports in the `input`-block can be used as if they were member variables. However, since the port only allows one-way communication, it is not possible to assign a value to it. In order to fire a spike, `emit_spike`-function has to be used instead.

NESTML

```
1 neruon iaf:
2   input:
3     spikes <- spike
4   end
5   update:
6     spikes = 12  # Error: use emit_spike to emit a spike
7   end
8 end
```

Figure 7.17: `BUFFER_NOT_ASSIGNABLE`: An attempt to assign a value to an `input`-buffer in line 6, which is not possible and thus reported as an error.

`CURRENT_PORT_IS_INH_OR_EXC`: Current ports are not allowed to have additional modifiers. In contrast to `spike`-ports which can be specialized by stating an additional modifier `inhibitory` or `excitatory`, currents are always lump together inhibitory and excitatory currents. Therefore no additional specializations are allowed.

```
1 neruon iaf:                                                    NESTML
2   input:
3     currents       <- current            # OK
4     currentsInh    <- inhibitor current  ⊗
5     currentsExc    <- excitator current  ⊗
6     currentsInhExc <- inhibitory excitator current  ⊗
7   end
8 end
```

Figure 7.18: `CURRENT_PORT_IS_INH_OR_EXC`: Additional modifiers for the `current`-port are not possible. Only the fist port declaration in line 3 is valid. The following three declarations in line 4, 5 and 6 use at least one modifier which is forbidden.

`FUNCTION_DEFINED_MULTIPLE_TIMES`: Every function name must be unique and unambiguous. In contrast to other programming languages like C++ and Java, function overloading is not available in NESTML.

```
1 neruon iaf:                                                    NESTML
2   function f1(V mV) void:
3   end
4   function f1(I pA) void: ⊗ # f1 is alredy defined in line 2
5   end
6 end
```

Figure 7.19: `FUNCTION_DEFINED_MULTIPLE_TIMES`: Two function definitions for `f1` with conflicting types for the single parameter. The second definition of `f1` is forbidden.

`FUNCTION_PARAMETER_HAS_TYPE_NAME`: It is forbidden to name a parameter using the name of an existing NESTML type in order to reduce ambiguity in neuron models. This also applies to physical units. As a consequence the function **f(V mV)** has an invalid parameter declaration, since the name **V** is the NESTML name for the SI unit **Volt**. The list of NESTML types can be found in section 5.3 and section 5.6.

`FUNCTION_RETURNS_INCORRECT_VALUES`: The types of all return statements inside a function must be compatible with the function's declared return type. If an explicit return type is omitted, the type `void` assumed.

`INVALID_TYPE_OF_INVARIANT`: The type of the invariant expression must be `boolean`. Syntactically, it is possible to state an arbitrary expression as an invariant. However, only invariants of the type `boolean` are reasonable. Therefore using a non-`boolean` invariant is regarded as an error.

```
1 neruon iaf:                                                    NESTML
2   function f1(V mV):
3     return 1   # The void return type is assumed
4   end
5
6   function f2(V_m mV) integer:
7     return ""  # string cannot be converted to integer
8   end
9 end
```

Figure 7.20: `FUNCTION_RETURNS_INCORRECT_VALUES`: Definition of two functions `f1` and `f2` which have invalid return values. `f1` contains a return statement of type `integer` while its return type is `void`. `f2` contains a return statement of type `string`, but is declared to return type `integer`.

```
1 neruon iaf:                                                    NESTML
2   parameters:
3     V_th mV = -50mV     [[ V_th >= -100mV ]] # OK
4     V_reset mV = -70mV [[ V_reset + 10mV ]]  # the type is mV and not
5                                              # as expected boolean
6   end
7 end
```

Figure 7.21: `INVALID_TYPE_OF_INVARIANT`: A neuron model with a correct invariant in line 3 and an incorrect invariant of type `mV` instead of the expected type `boolean` in line 4.

`MEMBER_VARIABLES_INITIALIZED_IN_WRONG_ORDER:` A member variable can be used only after it was defined. In every variable block each variable must be declared before it is used in an initialization expression or invariant. Declarations in the `parameters` and `internals` blocks only have access to variables which are defined in the same block. Declarations in the `state` block can access variables from the `parameters` block in addition.

`MEMBER_VARIABLE_DEFINED_MULTIPLE_TIMES:` Independently of the block, every name used for a member variable must be defined only once. It is, however, possible to overload variables in the `update` and `function` blocks.

`MISSING_RETURN_STATEMENT_IN_FUNCTION:` A function with non-`void` return type must have an explicit return statement. In case exit points are in the branches of an `if` statement, all branches must have an explicit return statement.

`NEST_FUNCTION_COLLISION:` Neurons and components are not allowed to have functions which are a part of the prescribed API of the target platform in order to avoid

```
1 neuron iaf:                                                    NESTML
2   state:
3     stateVar real = 1.0 + internalVar  # cannot access internals
4   end
5   parameters:
6     parameterVar real = stateVar  # parameters cannot access state
7   end                             # variables
8   internals:
9     internalVar real = stateVar   # internals cannot access state
10  end                             # variables
11 end
```

Figure 7.22: `MEMBER_VARIABLES_INITIALIZED_IN_WRONG_ORDER`: Several violations of initialization rules referencing variables from a different block. The initialization expression of the declaration in line 3 uses uses a variable from the `internals` block, which is not possible for declarations in the `state` block. The declarations in line 6 and 9 are invalid, because these declarations in the `parameters` and `internals` blocks are using variables from the `state` block in their initialization expression.

```
1 neuron iaf:                                                    NESTML
2   state:
3     V_m mV
4   end
5
6   parameters:
7     V_m mV   # V_m is already defined in line 3
8   end
9
10  update:
11    V_m mV  # OK, the V_m from line 3 is overloaded
12  end
13 end
```

Figure 7.23: `MEMBER_VARIABLE_DEFINED_MULTIPLE_TIMES`: The member variable `V_m` is defined once in line 3 and once in line 7 which is not allowed, although the conflicting definitions are in different blocks. In contrast, the redefinition of `V_m` as a local variable in the `update` block (line 11) is allowed.

compilation errors of the generated code.

The entire list of forbidden function for the NEST simulator is given in the following table:

`NEURON_WITH_MULTIPLE_OUTPUTS:` Neurons must have exactly one `output`-block. Syntactically it would be possible to define a neuron model with no or several `output` blocks. However, a neuron without an output is not reasonable.

```
neuron iaf:
  function f1() real:
    # missing return statement
  end
  function f2(x real) real:
    if x > 0:
      return x * 2
    elif true:
      # either all branches have a return or the
      # last statement in function is return
    end
  end
end
```

NESTML

Figure 7.24: `MISSING_RETURN_STATEMENT_IN_FUNCTION`: Functions with missing return statements in their body. `f1` has return type `real`, but no return statement is present in its body. Only one branch of the `if` condition in `f2` has a return statement.

```
neuron iaf:
  function calibrate() void: # 'calibrate' is part of the
  end                        # generated neuron implementation
end
```

NESTML

Figure 7.25: `NEST_FUNCTION_COLLISION`: A neuron model containing the function `calibrate` which collides with the corresponding function in the generated code for NEST.

Table 7.1: Lift of NEST functions which cannot be used in NESTML

calibrate	check_connection	connect_sender
get_status	handle	init_buffers_
init_state_	set_status	update

`NEURON_WITH_MULTIPLE_OR_NO_INPUTS:` Neurons must have exactly one `input`-block. Syntactically it would be possible to define a neuron model with no or several `input` blocks. However, a neuron without inputs is not reasonable and multiple inputs should be defined in the same block to increase the clarity of the model.

`NEURON_WITH_MULTIPLE_OR_NO_UPDATE:` Neurons must have exactly one `update`-block. Syntactically it would be possible to define a neuron model with no or several `update` blocks. However, a neuron without an update block is not reasonable and all update statements should be defined in a single block to increase the clarity of the model.

`VARIABLE_BLOCK_DEFINED_MULTIPLE_TIMES:` Every NESTML block must be defined at most once. Syntactically it would be possible to define a neuron model with several variable blocks of the same type, e.g. several `state`-blocks. However, a neuron with multiple `state` blocks is ambiguous.

7.3.4 Semantic errors in the equation specification

This section explains semantic checks for errors in the definition of differential equations.

`MISSING_INITIAL_VALUE:` For every order of the differential equation an initial value must be stated. NESTML allows the specification of differential equations with arbitrary order. In order to make the specification unambiguous, an initial value must be stated for every order from 0 up to the highest order minus one.

```
1 neuron iaf:                                                                    NESTML
2   state:
3    V_m mV = 0mV # the initial value for the order 0
4   end ⊗ # There is a missing initial value for V_m', e.g V_m' mV/ms = 0mV/ms
5   equations:
6     V_m'  = V_m'
7     V_m'' = V_m/tau_m + I_syn/C_m ⊗ # the order of the equation is 2
8   end
9 end
```

Figure 7.26: `MISSING_INITIAL_VALUE`: A neuron model with a missing initial value. The highest order of the differential equation for the variable `V_m` in line t is 2 denoted by the two ’ characters. Thus, initial values for order 0 and 1 must be given. As line 3 only specifis an initial value for order 0, this is treated as an error.

`EQUATIONS_ONLY_FOR_STATE_VARIABlE:` Only variables from the `state`-block can be further specified through a differential equation. As state variables are the only ones which evolve over time, a differential equation in the `equations`-block can be defined only for variables in the `state` block.

Kapitel 8

Ein Codegenerator für den NEST-Simulator

Nachdem die grundsätzliche Funktionsweise und die wichtigsten Ausgabeartefakte des NESTML-Generators für NEST in Kapitel 7 erläutert wurden, stellt dieses Kapitel den Generierungsprozess der ausführbaren NEST-Implementierung aus NESTML-Modellen vor. Mithilfe des generativen Ansatzes können diese Neuronen in den NEST-Simulator integriert und für neurowissenschaftliche Simulationen verwendet werden.

Um Neuronenmodelle in die Simulationsumgebung zu integrieren, bietet der NEST-Simulator einen vordefinierten Erweiterungsmechanismus[1] an. Die Benutzung dieses Mechanismus erfordert einiges an Wissen sowohl über die Modell- und Modulschnittstelle als auch über die Interna des Build-Werkzeugs zur Integration neuer Neuronenmodelle.

Der generative Ansatz bietet eine einfache Möglichkeit, dem Modellierer die aufwendigen und fehleranfälligen Aufgaben während der manuellen Implementierung des NEST-Codes abzunehmen. Die Konsolen-API des entwickelten NEST-Generators schafft eine transparente und leichtgewichtige Integration der modellierten Neuronenmodelle in den NEST-Simulator (vgl. Abschnitt 8.3).

Ein neues Neuronenmodell wird als C++-Klasse in den NEST-Simulator integriert. Die Ableitung dieser Klasse aus einem modellierten Neuron ist in Abschnitt 8.1 beschrieben. Mehrere Neuronen können dann zum einen NEST-Erweiterungsmodul zusammengefasst werden. Dieses Modul, das wiederum aus einer C++-Klasse mit vordefinierter Signatur besteht, kann schließlich mithilfe des spezialisierten und Modell-abhängigen CMake-Scripts konfiguriert, kompiliert und in den Simulator integriert werden. Der Aufbau dieser Modulklasse wird in Abschnitt 8.2 vorgestellt.

8.1 Generierung des Simulationscodes für den NEST-Simulator

NEST ist ein neuronaler Simulator [GD07, KMW+17], der in der nativen Programmiersprache C++ implementiert ist. Für die zeitliche Simulation verwendet NEST ein diskretes Zeitmodell [MMG+05], d.h. die modellierte Zeit ist in ein festes Gitter von Intervallen mit fester und äquidistanter Länge eingeteilt [MSPD07]. Neuronen und Synapsen werden jeweils nur an den Intervallgrenzen aktualisiert. Somit können die Elemente des

[1] http://nest.github.io/nest-simulator/extension_modules

simulierten Netzwerkes (Neuronen, Synapsen, Meß- und Stimulationsgeräte) Ereignisse nur an den Grenzen des jeweiligen Zeitintervalls austauschen. Die Länge eines Zeitintervalls wird als die Auflösung der Simulation bezeichnet. Die Auflösung kann individuell eingestellt und beim Start einer Simulation jeweils neu eingestellt werden.

Der NEST-Simulator verfügt über zwei native Schnittstellen zur Simulationssteuerung. Die erste Schnittstelle ist als eine DSL mit dem Namen Simulation Language Interpreter (SLI) realisiert [GD07]. SLI ist eine stackbasierte Sprache, die von PostScript [Pre85] inspiriert ist. Die zweite Schnittstelle ist PyNEST [EHM+09]. Dabei handelt es sich um eine Python-API, mit der NEST aus einem Python-Programm gesteuert werden kann. PyNEST ermöglicht es, NEST nahtlos mit anderen in Python erstellten Werkzeugen zu integrieren. Die beiden Schnittstellen bieten denselben Funktionsumfang. Da PyNEST die modernere und komfortablere Schnittstelle ist, wird nur sie im Laufe der Ausarbeitung weiter behandelt.

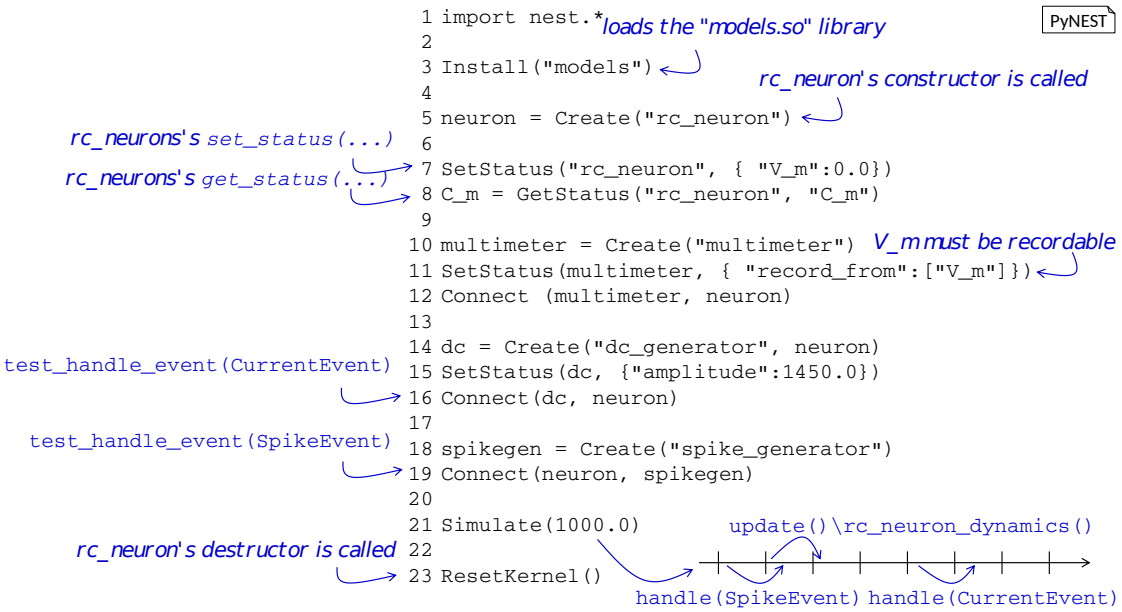


Abbildung 8.1: Überblick der Funktionalität, die das integrierte NESTML-Modul im NEST-Simulator zur Verfügung stellt.

Abbildung 8.1 stellt am Beispiel eines *PyNEST*-Skripts die Funktionsweise der generierten Implementierung des Neurons `rc_neuron` in NEST vor (vgl. Abbildung 8.2). Die in der Abbildung motivieren Funktionen werden im Folgenden genauer eingeführt. Eine zwingende Voraussetzung für die Ausführung des abgebildeten Skriptes ist die Integration des entsprechenden Moduls in den NEST-Simulator (vgl. Abschnitt 7.2 für die Beschreibung der Integrationsschritte).

Das generierte Modul und die darin enthaltenen Neuronen werden durch den Funkti-

onsaufruf `Install` in Zeile 3 des Skriptes verfügbar gemacht. In Zeile 5 wird ein Exemplar des Neuronenmodells `rc_neuron` instanziiert. Dabei wird sein Konstruktor aufgerufen. In den Zeilen 7 und 8 werden Variablen, die innerhalb dieses Neurons spezifiziert sind, mithilfe der Funktionsaufrufe der Funktionen `GetStatus` und `SetStatus` abgefragt und gesetzt. Diese Anfragen leiten an die `get_status`- und `set_status`-Methoden des Neurons weiter. In Zeile 10 wird ein Multimeter (ein Messgerät für verschiedene analoge Größen) erstellt. Die aufzunehmende Größe wird in Zeile 11 auf die Neuronenvariable `V_m` gesetzt. Anschließend wird das Multimeter in Zeile 12 mit dem Neuron verbunden, um den Wert `V_m` dieses Neurons während der Simulation aufzuzeichnen. In den Zeilen 14-16 wird ein Gleichstromgenerator erzeugt und mit dem Neuron verbunden. Die Amplitude des Generators wird auf 1450.0 pA gesetzt, was zu regelmäßigen Spikes des Neurons führt. In den Zeilen 18 und 19 wird ein Spikegenerator erzeugt und mit dem Neuron verbunden. Im vorliegenden Beispiel sendet er keine Spikes an das Neuron und dient nur der Demonstration. In beiden Fällen, in denen ein Generator mit einem Neuron verbunden wird, ist die Methode `test_handle_event` für die korrekte Verbindung verantwortlich.

In Zeile 21 wird eine Simulation für 1000 Millisekunden gestartet. Während dieser Simulationszeit erhält das Neuron Ereignisse vom Typ `SpikeEvent` vom Spikegenerator und vom Typ `CurrentEvent` vom Gleichstromgenerator. Die im Neuronenmodell definierten Differenzialgleichungen werden mithilfe der Methode `rc_neuron_dynamics` propagiert. Schließlich wird in Zeile 23 der Zustand des NEST-Simulators mithilfe der Funktion `ResetKernel` zurückgesetzt. Dabei wird unter anderem der Destruktor des Neurons `rc_neuron` aufgerufen.

Abbildung 8.2 stellt die Struktur der NEST-Neuronenklasse anhand des konkreten Neurons `rc_neuron` vor, das in Kapitel 7 eingeführt wurde. Die Implementierung eines Neurons für den NEST-Simulator ist in einer Klasse gekapselt, in die vier C++-`structs` eingebettet sind, auf die die Ports und der Datenzustand des NESTML-Neurons abgebildet wird. Darüberhinaus werden Getter- und Setter-Methoden für alle Datenmember generiert.

Die Getter- und Setter-Methoden bilden eine Zugriffsfassade (vgl. das Fassadenmuster aus [GHJV93]), die einen einheitlichen Zugriff auf die Variablen und Ports des Neurons in generiertem und handgeschriebenem Code ermöglicht (vgl. Unterabschnitt 8.1.7 für die Einführung von handgeschriebenem Code). Handgeschriebener und generierte Code kann stets auf eine konsistente Modell-API zurückgreifen, um eigene Logik umzusetzen. Einerseits vereinfacht das die Erstellung von handgeschriebenen Erweiterungen, andererseits können mehrere NESTML-Neuronen unabhängig und modular verarbeitet werden, da die Resultate der Codegenerierung auf Basis dieser konsistenten API zusammengeführt werden.

Der NEST-Generator erstellt eine Menge von im NEST-Simulator vorgeschriebener Methoden, mit denen das Kommunikations- und Laufzeitverhalten des Neurons definiert wird. Die wichtigsten Methoden werden in der folgenden Auflistung erläutert:

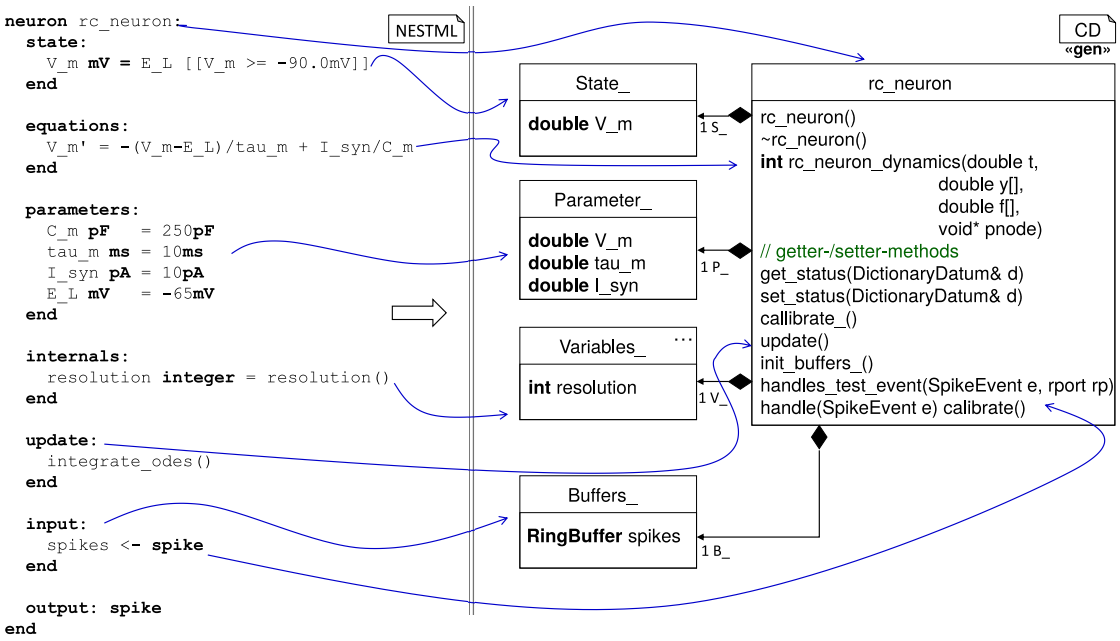


Abbildung 8.2: Überblick der generierten Implementierung des NEST-Neurons anhand einer vereinfachten Version des NESTML-Neurons `rc_neuron`.

`Getter- und Setter-Methoden:` Für alle Variablen aus den Variablenblöcken eines Neurons (d.h. aus den `state`-, `parameters`- und `internals`-Blöcken) und für jeden Port aus dem `input`-Block wird eine Getter-Methode generiert. Zusätzlich wird eine Setter-Methode für alle Variablen außer den `function`-Variablen aus allen Variablenblöcken erstellt.

`get_status:` Mithilfe dieser Methode werden Werte von Variablen eines Neurons in PyNEST ausgelesen.

`set_status:` Mithilfe dieser Methode werden Werte von Variablen eines Neurons von PyNEST aus gesetzt.

`handle_test_event, handle:` Mithilfe einer `handle_test_event`-Methode mit der entsprechenden Signatur signalisiert das Neuron dem NEST-Simulator, welche Typen von Ereignissen das Modell unterstützt und auf welchem Port diese vom Neuron empfangen werden können. Die momentan unterstützten Ereignistypen sind `SpikeEvent` und `CurrentEvent`. `handle_test_event` wird beim Aufbau des neuronalen Netzwerkes im Simulator aufgerufen, um die Kompatibilität von Sender und Empfänger sicherzustellen. Die dazu passende `handle`-Methode wird zur Simulationszeit aufgerufen, wenn das Neuron ein Ereignis des entsprechenden Typs

an den vorgesehenen Port erhält.

`calibrate_`: Diese Methode initialisiert alle Hilfsvariablen aus dem `internals`-Block.

`init_buffers_`: Diese Methode initialisiert NEST-Datenstrukturen, in denen die Ereignisse der `input`-Ports gespeichert werden.

`rc_neuron_dynamics`: Diese Methode führt einen Schritt der Propagation der im Neuronmodell definierten Differenzialgleichungen durch.

`update`: Diese Methode implementiert das Laufzeitverhalten des Neurons. Sie wird nach jedem Übergang ins neue Gitterintervall während der Simulation aufgerufen, damit das Neuron seinen Zustand entsprechend der Zeitentwicklung aktualisieren kann.

Im Weiteren wird die Generierung einzelner Elementen und deren Zielsetzung erklärt. Die Generierungsaspekte werden anhand einer an die Modelltransformationsnotation [Wei12] angelehnten Notation [Rum12] visualisiert. Zur Veranschaulichung werden jeweils Ausschnitte des Neurons aus Abbildung 8.2 verwendet.

8.1.1 Abbildung der Neuronendeklaration

Jedes NESTML-Neuron wird in eine eigenständige C++-Klasse transformiert. Abbildung 8.3 demonstriert die Deklaration dieser Klasse für das Neuron `rc_neuron`. Wie alle anderen Neuronen im NEST-Simulator erweitert auch `rc_neuron` die NEST-Basisklasse `Archiving_Node`. Da die C++-Programmiersprache über keine automatische Speicherverwaltung (insb. Garbage Collection) verfügt, muss der dynamisch belegte Speicher im Destruktor der Klasse manuell freigegeben. Das Loggen des Neuronenzustands, der sich typischerweise auf die Werte der Variablen aus dem `state`-Block beschränkt, findet mithilfe der statischen Variable `recordablesMap_` statt.

Aufgrund von potenziellen Probleme beim Zusammenfügen der Quelltexte und dem anschließendem Linken der binären Artifakten in C++ [Str13] ist es eine übliche Konvention, Klassendeklarationen mit einer Folge von `ifndef, define, endif` Präprozessordirektiven zu schützen. Ansonsten könnte dieselbe Klassendeklration vom C++-Compiler mehrfach übersetzt werden, was zu einem nicht ausführbaren Programm führen würde. Die Präambel der Klassendeklaration bindet eine Menge von notwendigen Headerdateien aus der NEST-Installation ein. Der Klassenname des NEST-Neurons entspricht im generierten Code dem des entsprechenden NESTML-Neurons.

Alle Variablen, deren Historie während der Simulation potenziell aufgenommen werden soll, müssen mit dem entsprechenden `insert`-Aufruf in der Klasse `RecordablesMap` registriert werden. Wie in Abbildung 8.3 zu sehen ist, wird beim `insert`-Aufruf der Name der Variable und ein Zeiger auf die korrespondierende Getter-Methode angegeben. Anhand des Namens können die Werte dieser Variable von PyNEST aus ausgelesen

NESTML

```
neuron rc_neuron:
  # ...
end
```

C++

```
#ifndef RC_NEURON_H
#define RC_NEURON_H
// Includes from nestkernel:
#include "archiving_node.h"
#include "connection.h"
// ...

class rc_neuron : public Archiving_Node {
public:
  rc_neuron();
  ~rc_neuron();
private:
  // ...
  static RecordablesMap< rc_neuron > recordablesMap_;
}

template <> void RecordablesMap< rc_neuron >::create() {
  insert_( "V_m", &rc_neuron::get_V_m );
}
#endif
```

Abbildung 8.3: Auszug der aus dem Neuron `rc_neuron` generierten C++-Klassendeklaration.

werden (vgl. Abbildung 8.1). Der Generator benutzt dazu stets den Namen der Variable im Modell, um die Konsistenz zwischen dem Namen im Modell und dem Namen, der in PyNEST zur Verfügung steht, zu gewährleisten. Im Normalfall werden alle Variablen aus dem `state`-Block in die `RecordablesMap` aufgenommen. Weitere Variablen können aufgenommen werden, wenn deren Deklaration mit dem Schlüsselwort `record` gekennzeichnet wird. Dies erleichtert insbesondere das Debugging während der Modellentwicklung.

Im Weiteren wird die Generierung der in Abbildung 8.2 aufgelisteten C++-Structs und Methoden erläutert, die innerhalb der Klassendeklaration implementiert sind.

8.1.2 Abbildung des Datenzustandes

Jeder Variablenblock von NESTML wird in eine C++-`struct` transformiert, die die Variablen des jeweiligen Blockes kapselt. Abbildung 8.4 zeigt exemplarisch die Transformation eines Variablenblockes am Beispiel des `parameters`-Blockes, der auf eine `Parameters_`-Struct abgebildet wird. Die übrigen NESTML-Variablenblöcke (d.h. die `state`- und `internals`-Blöcke) werden entsprechend transformiert. Der `state`-Block wird dabei zu einer `State_-struct`, der `internals`-Block zu einer `Variables_-struct` trans-

formiert.

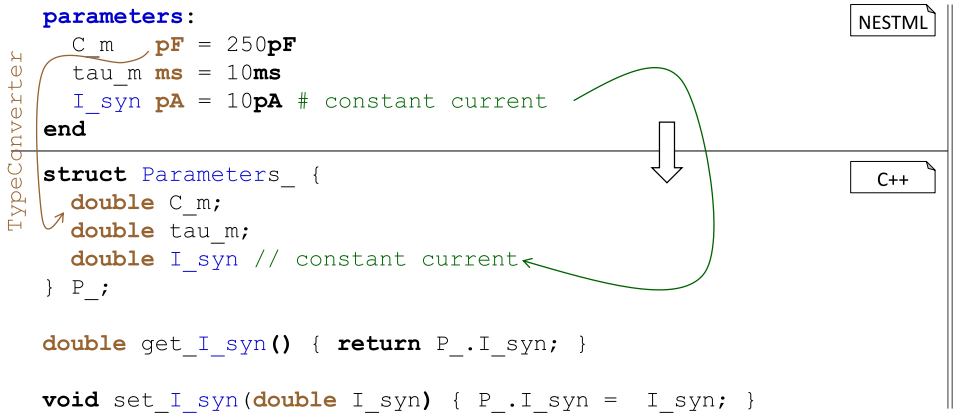


Abbildung 8.4: Exemplarische Ableitung der `Parameters_-struct` aus dem `parameters`-Block. Die Abbildung demonstriert die Erstellung einer eingebetteten C++-`struct`, sowie der Getter- und Setter-Methoden.

Jede Variable aus dem `parameters`-Block wird auf eine entsprechende Variable in der C++-`struct` abgebildet. Dabei wird der Name der Variable aus dem NESTML-Modell als Variablenname in der C++-Struct übernommen.

Alle SI-Typen werden auf den Typ `double` abgebildet, da der konkrete SI-Typ für die Simulation nicht relevant ist und die explizite Implementierung der SI-Typen im generierten C++-Code nur die Performance verschlechtern würde. Da Code nur für Modelle generiert wird, die alle Kontextbedingungen erfüllen ist die Typkorrektheit bereits garantiert und muss zur Laufzeit nicht erneut geprüft werden (vgl. Abschnitt 7.3). Die Abbildung der anderen NESTML-Typen ist in Abschnitt 5.3 zusammengefasst.

Der `TypeConverter` ist für die Konvertierung von NESTML-Typen in entsprechende C++-Typen verantwortlich. Bei Bedarf könnte der `TypeConverter` durch die Subklassenbildung erweitert werden um die Konvertierung der Typen an neue Anforderungen anzupassen. Zum Beispiel könnte durch die Verwendung des `long double`- bzw. `float`-Typs anstelle vom Typ `double` eine höhere bzw. niedrigere Genauigkeit auf Kosten eines niedrigeren bzw. höheren Speicherverbrauches erreicht werden. Um die Verständlichkeit des generierten Codes zu erhöhen bzw. auch die Generierung von Dokumentation zu ermöglichen, werden Modellkommentare in den generierten C++-Code übernommen.

Für jede Struktur wird eine klassenlokale Variable instantiiert: `S_` für den `state`-Block, `V_` für den `internals`-Block und `P_` für den `parameters`-Block.

Schließlich wird für jede Variable aus allen Variablenblöcken und für alle Ports aus dem `input`-Block je eine Getter-Methode generiert. Somit werden explizite und konsistente Schnittstellen für Variablen und Ports geschaffen. Diese Schnittstellen abstrahieren von der konkreten Implementierung und erhöhen die Modularität des generierten Codes.

Mithilfe dieser Schnittstellen kann Code, der durch verschiedene Generatoren produziert wird, transparent komponiert werden. Auch handgeschriebener Code kann auf diese Weise mit dem generierten Code interagieren.

Variablen aus allen Variablenblöcken können mit Standardbelegungen initialisiert werden. Die Initialisierung der Variablen aus den `state`- und `parameters`-Blöcken findet im Klassenkonstruktor statt. Dabei werden zuerst alle `parameters`-Variablen und erst dann alle `state`-Variablen initialisiert. Die Reihenfolge dieser Blöcke im NESTML-Modell ist unerheblich. Abbildung 8.5 demonstriert diesen Sachverhalt an einem Beispiel. Zwar steht der `state`-Block dort im Neuron vor dem `parameters`-Block, dennoch werden die Variablen aus dem `parameters`-Block im Konstruktor zuerst initialisiert. Dadurch wird es möglich, die `parameters`-Variablen im `state`-Block für die Initialisierung zu verwenden. Die Verletzung dieser Konvention wird durch eine Kontentextbedingung geprüft (vgl. Abschnitt 7.3).

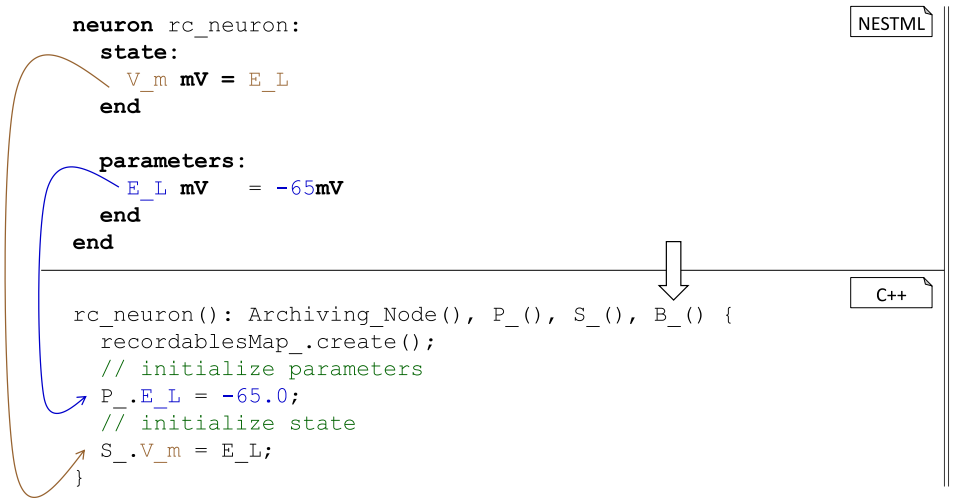


Abbildung 8.5: Exemplarische Initialisierung der Variablen mit den Standardbelegungen aus den `state`- und `parameters`-Blöcken im Klassenkonstruktor.

Die Variablen aus dem `internals`-Block werden in der `calibrate_`-Methode initialisiert. Die `calibrate_`-Methode wird bei jedem Aufruf der `Simulate`-Methode aufgerufen wird. Dadurch können in diesem Block Variablen definiert werden, die vom Wert der Simulationsauflösung abhängen. In Abbildung 8.6 wird die Variable `resolution` auf den Wert der Simulationsauflösung initialisiert. Dafür verantwortlich ist die in NESTML vordefinierte `resolution`-Methode verwendet. Im generierten Code wird diese Methode auf einen C++-Ausdruck abgebildet, der den Wert der Simulationsauflösung in NEST annimmt.

Für die *ExpressionslDSL*-Subsprache (vgl. Kapitel 5 für einen Überblick aller NESTML-Subsprachen) wurde ein modularer und erweiterbarer PrettyPrinter [Völ11] bzw. Unpar-

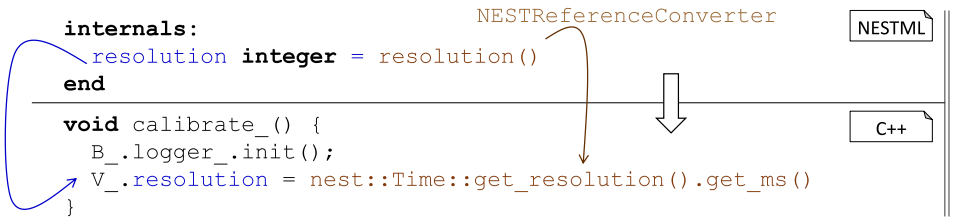


Abbildung 8.6: Initialisierung der NEST-Variablen mit den Standardbelegungen aus dem `internals`-Block

ser [PE88] entwickelt. Die Konvertierung des `ExpressionsDSL`-Codes, der als Initialisierungsausdruck der Variable `resolution` verwendet wird, findet mithilfe dieses PrettyPrinters statt. Aufgrund der Möglichkeit, den PrettyPrinter zu konfigurieren, bietet er hohes Maß an Wiederverwendung. Im vorliegenden Fall wird der PrettyPrinter mit einem `NESTReferenceConverter` komponiert. Der `NESTReferenceConverter` ist dann dafür verantwortlich, die Variablen- und Funktionsreferenzen in entsprechende NEST-Referenzen abzubilden. Beispielsweise wird aus der vordefinierten NESTML-Funktion `resolution()` ein NEST-spezifischer C++-Ausdruck generiert, der die Auflösung der Simulation berechnet. Die genaue Abbildungsvorschrift ist im `NESTReferenceConverter` gekapselt und kann für die Verwendung vom PrettyPrinter in unterschiedlichen Kontexten durch andere Konverterklassen ausgetauscht werden.

NEST erlaubt es, die Werte der Variablen aus dem `state`- bzw. `parameters`-Block von PyNEST aus zu setzen und auszulesen. Diese Funktionalität ist in den `get_status`- und `set_status`-Methoden gekapselt. Abbildung 8.7 demonstriert exemplarisch die Ableitung dieser Methoden anhand des `state`-Blockes im Neuron `rc_neuron`.

Das Kernkonzept für die Kommunikation mit PyNEST ist ein assoziatives Datenfeld (engl: dictionary), das den beiden Methoden als Parameter übergeben wird. Diese Datenstruktur bildet einen Namen auf seinen aktuellen Wert ab. Die `get_status`-Methode wird benutzt, um den aktuellen Wert einer Variablen aus dem Neuron auszulesen. Dafür wird die vordefinierte NEST-Funktion `def` benutzt. Die `set_status`-Methode wird benutzt, um einen neuen Wert einer Variablen aus PyNEST im Neuron zu setzen. Dabei wird die NEST-Funktion `updateValue` benutzt, um einen spezifischen Wert aus dem Datenfeld auszulesen und diesen Wert der Variable zuzuordnen. Der passende Eintrag im Datenfeld wird anhand des Variablennamens identifiziert. Auch hier stellt der NEST-Generator sicher, dass der Variablenbezeichner, der beim Funktionsaufruf als String spezifiziert ist, und die entsprechende C++-Variable konsistent sind.

Schließlich werden am Ende der `set_status`-Methode alle Invarianten geprüft, die im `parameters`- oder `state`-Block definiert sind. Wenn eine Invariante beim Setzen eines neuen Wertes verletzt wird, löst die `set_status`-Methode eine C++-Exception aus. Die verletzte Invariante wird als Teil der Fehlerbeschreibung in der C++-Exception codiert.

NESTML

```
state:
  V_m mV = -65.mV [[V_m >= -90.0mV]]
end
```

C++

```
void get_status( DictionaryDatum& __d ) const {
  def< double >( __d, "V_m", S_.V_m );
}

void set_status( DictionaryDatum& __d ) const {
  updateValue< double >( __d, "V_m", S_.V_m );

  if (!(S_.V_m => -90.0)) {
    BadProperty( "'V_m >= -90.0mV' constraint is violated" );
  }
}
```

Abbildung 8.7: Exemplarische Generierung der `get_status`- und `set_status`-Methoden aus dem `state`-Block. `get_status` sammelt Werte aller relevanten Variablen in `__d`. `set_status` liest die Werte für die relevanten Variablen aus `__d` und weist diese Werte den entsprechenden Variablen in der C++-Implementierung zu.

Somit ist die Fehlernachricht für den Modellierer verständlich, da er die Fehlernachricht auf Basis des Modells nachvollziehen kann.

8.1.3 Abbildung von Ports

Der `input`-Block eines Neurons, der in NESTML aus einer nicht leeren Menge von benannten Ports besteht (vgl. Abschnitt 5.2 für die Beschreibung dieser Modellierungselemente), wird auf die `Buffers_-struct` abgebildet. Jedes Vorkommen eines Ports im Block wird in eine Variable vom Typ `RingBuffer` transformiert. In diesem Puffer werden Ereignisse gespeichert, die am korrespondierenden Port ankommen. Per Konvention enthält die `Buffers_-struct` eine zusätzliche `Logger`-Variable, mit der der Datenzustand des Neurons protokolliert werden kann. Genauso wie schon im Fall der Strukturen für die Blockvariablen, wird auch die `Buffers_-struct` als klassenlokale Variable `B_` instanziiert.

Abbildung 8.8 demonstriert die exemplarische Generierung der `Buffers_-struct`. Um eine Getter-Methode für Ports zu generieren, wird für jeden Port eine zusätzliche Variable erzeugt. Der Name dieser Variable setzt sich aus dem Portnamen und dem Suffix `_last_-sum` zusammen. Während der Simulation enthält diese Variable den Wert aller im letzten Gitterfenster am Port angekommenen Spikes oder Ströme (falls der korrespondierende Port vom Typ `current` ist). Alle `_last_sum`-Variablen werden anschließend benutzt, um die Getter-Methoden für die Ports umzusetzen. Somit wird die gleiche Schnittstelle wie bei allen anderen Blockvariablen eingehalten, um auf die Modellelemente in der C++-

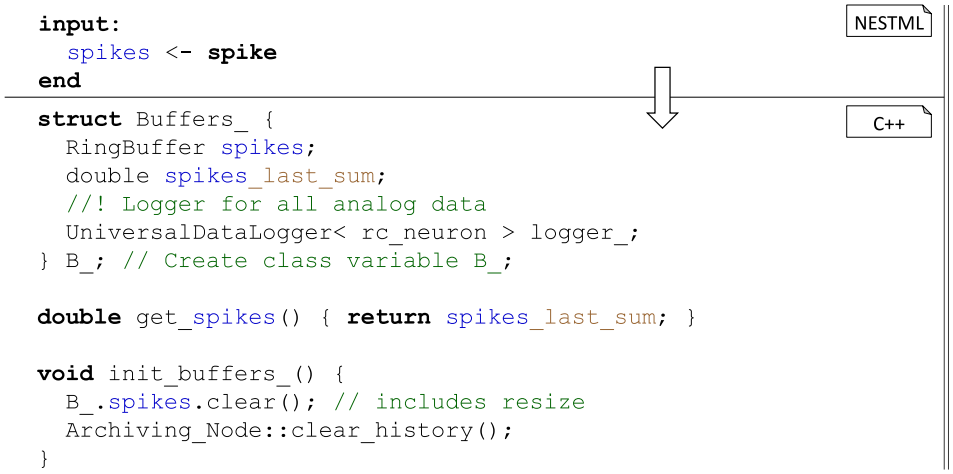


Abbildung 8.8: Exemplarische Generierung der `Buffers_-struct` aus dem `input`-Block.

Implementierung zuzugreifen.

Zwar können Ports aus dem `input`-Block weder mit initialen Werten belegt werden noch aus PyNEST gesetzt werden, dennoch muss auch für diesen Block Initialisierungscode generiert werden. Dafür wird die `init_buffers_`-Methode überschrieben. Dabei werden die für das Speichern der Ereignisse verantwortliche `RingBuffer`-Datenstrukturen geleert.

Die `init_buffers_`-Methode aus Abbildung 8.8 demonstriert, wie diese Methode für das Neuron `rc_neuron` aussieht. Zum einen wird innerhalb dieser Methode der Inhalt des Puffers geleert. Zum anderen wird ein per Konvention festgelegter Nebeneffekt der Methode ausgeführt. Mit dem Aufruf der `clear_history`-Methode wird die bis dahin aufgenommene Historie des Neurons zurückgesetzt.

Im NEST-Simulator wird die Kompatibilität zwischen Sender und Empfänger bereits während der Erstellung von Verbindungen getestet, um Fehler zur Laufzeit zu vermeiden. Diese Prüfung wird durch das Überschreiben der `test_handle_event`-Funktion mit der passenden Signatur vorgenommen. Dabei soll die Implementierung dieser Methoden einem vorgesehenen Protokoll [KSE+14] folgen.

Abbildung 8.9 demonstriert die Kompatibilitätsprüfung der Ports im Neuron `rc_neuron` für den Empfang der Ereignisse vom Typ `SpikeEvent`. Dabei wird die `handles_test_event`-Methode durch die Neuronenklasse überschrieben. Innerhalb dieser Methode teilt das Neuron mit, dass es Ereignisse vom Typ `SpikeEvent` am Port mit der Nummer `0` empfangen kann. Für den Empfang der Ereignisse über einen `current`-Port würde einfach eine weitere Methode mit neuer Signatur generiert, in der der Typ `SpikeEvent` durch den Typ `CurrentEvent` ersetzt würde. Da die Basisklasse für alle möglichen Eventtypen eine C++-Exception wirft, wird durch diese Überladung eine Kompatibilität des

NESTML

```
input:
  spikes <- spike
end
```

C++

```
port handles_test_event( SpikeEvent&, rport r_type ) {
  if (r_type != 0 )
    throw UnknownReceptorType(r_type , get_name() );
  return 0;
}

void handle( SpikeEvent& e ) {
  double weighted_spike = e.get_weight() * e.get_multiplicity();
  B_.spikes.add_value(e.get_rel_delivery_steps(
      kernel().simulation_manager.get_slice_origin() ),
      weighted_spike );
}
```

Inherited method that returns the class name

Abbildung 8.9: Exemplarischer Code für die Kompatibilitätsprüfung der Ports aus dem `input`-Block im NEST-Simulator.

Neurons mit dem entsprechenden Ereignistyp signalisiert.

Schließlich verarbeitet die `handle`-Methode ankommende Ereignisse vom Typ `SpikeEvent`. Dabei wird stets dasselbe Verfahren angewandt: das Gewicht des Ereignisses wird mit seiner Multiplizität verknüpft und im entsprechenden `RingBuffer` gespeichert. Wenn im Neuronenmodell unterschiedliche Ports für `inhibitory`- und `excitatory`-Spikes definiert werden, ändert sich die generierte Implementierung nur insofern, dass eine Fallunterscheidung anhand des Vorzeichens beim Gewicht des Spikes stattfindet. Positiv gewichtete Spikes werden dann entsprechend in den `excitatory`-Port und negativ gewichteten Spikes in den `inhibitory`-Port sortiert. Somit sind die ersten beiden möglichen Fälle für die Spezifikation von Ports aus Abschnitt 5.2 behandelt.

In Fall (C) aus Abbildung 5.9 verfügt das Neuron über mehrere Ports desselben Typs. In diesem Fall wird die Generierung der `Buffers_-struct` und der Registrierungbzw. Verarbeitungsmethoden verändert. Dabei folgt die generierte Implementierung dieser Neuronen einer NEST-Konvention. Anstelle der einzelnen `RingBuffer`-Instanzen für jeden Rezeptorport wird ein Array von `RingBuffer`-Variablen und ein `Enum` mit symbolischen Indizes generiert (vgl. `SynapseTypes` und `spike_inputs_`). Die Namen der Indizes entsprechen den Bezeichnern, die im Neuron für die Ports benutzt wurden. Die `handles_test_event`-Methode verhandelt dabei, wie die Rezeptornummern auf die Portnummer abgebildet werden. Die `handle`-Methode leitet die Ereignisse, die an einem Rezeptor ankommen, an den korrespondierenden `RingBuffer` weiter.

Auch hier garantiert erst der generative Ansatz die konsistente Umsetzung der beiden `handles_test_event` und `handle`-Methoden in allen unterstützen Konstellationen. In manuell erzeugten Neuronenmodellen führte die Komplexität in diesem Bereich oft zu Fehlern und einem langwierigen Entwicklungsprozess.

NESTML

```
input:
  receptor_A <- spike
  receptor_B <- spike
end
```

C++

```
enum SynapseTypes {
  INF = 0, receptor_A, receptor_B, SUP };

struct Buffers_ { std::vector< RingBuffer > spike_inputs_; }

port handles_test_event( SpikeEvent&, rport receptor_type ) {
  if ( !( INF < receptor_type && receptor_type < SUP ) ) {
    throw UnknownReceptorType( receptor_type, get_name() );
  }
  else {
    return receptor_type - 1;
  }
}

void handle( SpikeEvent& e ) {
   B_.spike_inputs_[ e.get_rport() ].add_value(
    e.get_rel_delivery_steps(
      kernel().simulation_manager.get_slice_origin() ),
    e.get_weight() * e.get_multiplicity() );
}
```

Abbildung 8.10: Exemplarische Abbildung des `input`-Blockes mit mehreren Rezeptoren.

8.1.4 Abbildung von `update`- und `function`-Blöcken

`update`- und `function`-Blöcke werden zu eigenständigen Methoden in der generierten Neuronenklasse. Methoden, die im Neuron definiert sind, werden zu gleich benannten Methoden in der Neuronenklasse. Die verwendeten NESTML-Typen werden zu den entsprechenden C++-Typen konvertiert. Der in der Methodendefinition eingebettete *ProceduralDSL*-Code wird mithilfe des entsprechenden sprachspezifischen Generators transformiert, der mithilfe der zuvor erläuterten `TypeConverter`- und `NESTReferenceConverter`-Klassen umgesetzt ist.

Abbildung 8.13 demonstriert diesen Generierungsansatz am Beispiel einer Methode und des `update`-Blocks. Die NESTML-Funktionsdeklaration wird zu einer C++-Funktionsdeklaration. Referenzen zu `state`- und `parameters`-Variablen werden durch Aufrufe der entsprechenden Getter und Setter-Methoden ersetzt. NESTML verbietet das Überladen einer Funktion (vgl. NESTML_FUNCTION_DEFINED_MULTIPLE_TIMES aus Abschnitt 7.3). Diese Einschränkung vermeidet Kollisionen im generierten Code, wenn in einem Neuron zwei gleich benannte Methoden existieren, die sich nur anhand der SI-Typen in der Parameterliste unterscheiden. Anderenfalls wären diese Methoden im C++-Code nicht mehr unterscheidbar, da alle SI-Typen auf den Typ `double` abgebildet

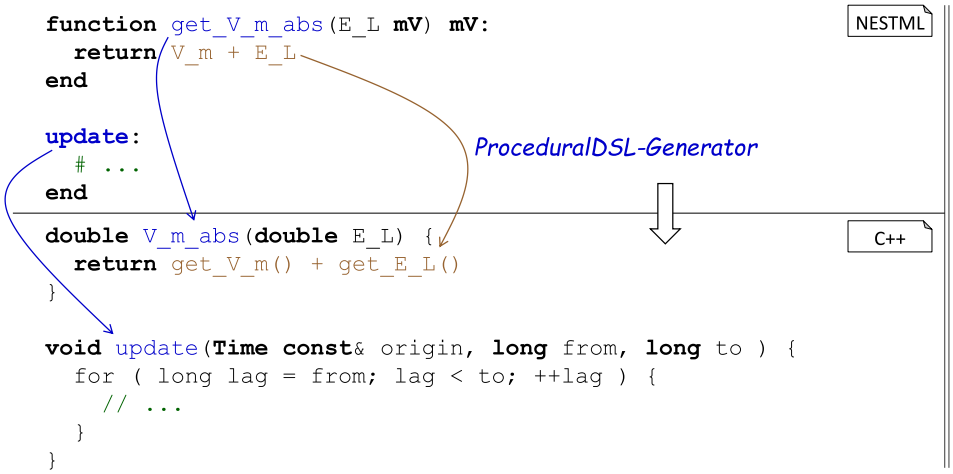


Abbildung 8.11: Exemplarische Abbildung einer Neuronenmethode und Generierung des Grundgerüsts für die Aktualisierung des Neuronenzustandes in der `update`-Methode.

werden.

Der `update`-Block wird zu einer Methode mit der eingebetteten Schleife abgebildet. Aufgrund der internen Optimierung verteilt der NEST-Simulator Ereignisse nicht direkt, sondern akkumuliert sie während eines bestimmten Intervalls. Entsprechend wird die `update`-Methode nicht für jeden simulierten Zeitschritt aufgerufen, sondern jeweils für einen ganzen Block evaluiert. Die eingebettete Schleife iteriert über die einzelnen Gitterpunkte des Blocks. Die `from`- und `to`-Variablen grenzen den zu behandelnden Gitterabschnitt ein. Die Schleifenvariable `lag` referenziert dann jeweils den aktuellen Abschnitt des Zeitgitters. Innerhalb der Schleife wird der Inhalt des `update`-Blockes mithilfe des *ProceduralDSL* Generators transformiert.

8.1.5 Abbildung des `equations`-Blockes

Differenzialgleichungen sind ein wesentlicher Bestandteil der Neuronendefinition. Ein System von Differenzialgleichungen wird in NESTML-Neuronen innerhalb des `equations`-Blockes definiert (vgl. Abschnitt 5.4). In diesem und dem kommenden Abschnitt werden Lösungsstrategien für unterschiedliche Ausprägungen eines solchen Systems erläutert.

Anhand des Neurons `rc_neuron` werden unterschiedliche Strategien vorgestellt, wie das Gleichungssystem mithilfe des Frameworks GNU Scientific Library (GSL) propagiert und mit `shape`-Funktionen (vgl. Abschnitt 5.4), in Differenzialgleichungen umgegangen werden kann.

Im einfachsten Fall besteht der `equations`-Block aus einem System von Differenzi-

algleichungen, in denen keine `shape`-Funktionen vorkommen. In diesem Fall wird das GSL-Framework[2] [Gou09] für die Lösung verwendet. Dafür wird aus dem `equations`-Block eine Schrittfunktion (engl: step function) generiert, die an das GSL-Framework übergeben wird.

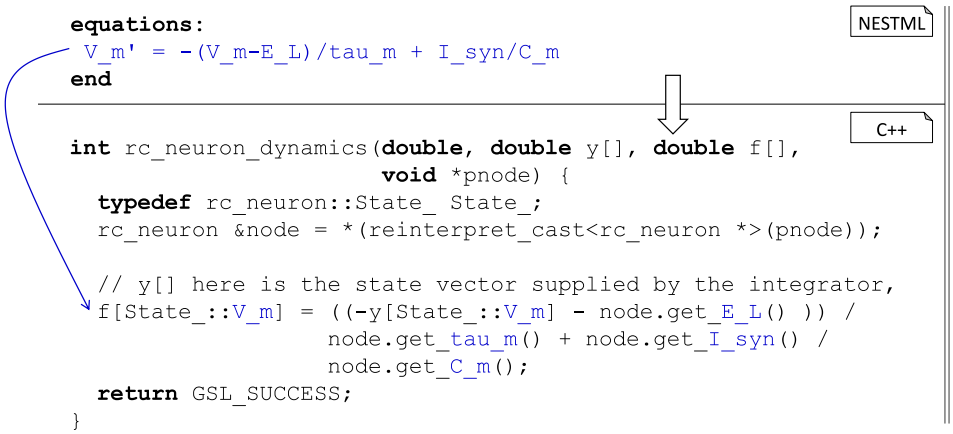


Abbildung 8.12: Exemplarische Generierung der Differenzierungsfunktion aus dem `equations`-Block.

Abbildung 8.12 zeigt eine exemplarische Methode, die aus dem Neuron `rc_neurons` generiert wird. Alle Differenzialgleichungen werden dabei in einer für das GSL-Framework passenden Darstellung ausgegeben.

Um eine nahtlose Integration mit dem GSL-Framework zu ermöglichen, wird die `State_-struct` erweitert. Ähnlich zu Multirezeptorports, werden für die Zustandsvariablen symbolische Indizes erzeugt. Die `State_-struct` enthält ein Array, das als Zeiger an die Differenzierungsfunktion übergeben wird. Diese symbolischen Indizes werden verwendet, um auf die entsprechenden Zustandsvariablen innerhalb der Schrittfunktion zuzugreifen.

Der *ProceduralDSL*-Generator muss in diesem Fall angepasst werden. Die Anpassung geschieht mithilfe der Komposition des Generators mit einem `GSLReferenceConverter`. Dieser Konverter ist in der Lage, die Variablenreferenzen innerhalb des `equations`-Blockes korrekt zu behandeln. So wird beim Zugriff auf die Zustandsvariablen das Zustandsarray und ein passender Index benutzt. Für die anderen Variablen wird ein `node`-Präfix vor jede Getter-Methode hinzugefügt. Dabei ist `node` eine Referenz, die auf das zu differenzierende Neuron referenziert. Somit gewährt die `node`-Referenz den direkten Zugriff auf Attribute und Methoden des Neurons.

Die eigentliche Propagation der Differenzialgleichungen aus dem `equations`-Block findet durch den Aufruf der vordefinierten `integrate_odes`-Funktion innerhalb des `upda-`

[2] https://www.gnu.org/software/gsl/

`te`-Blockes statt. Im Allgemeinen wird dieser Aufruf in eine `while`-Schleife transformiert. Innerhalb der Schleife findet die wiederholte Integration des `equations`-Blockes statt. Die Wiederholung des Integrationsschrittes ist aufgrund des adaptiven Verfahrens des GSL-Frameworks notwendig.

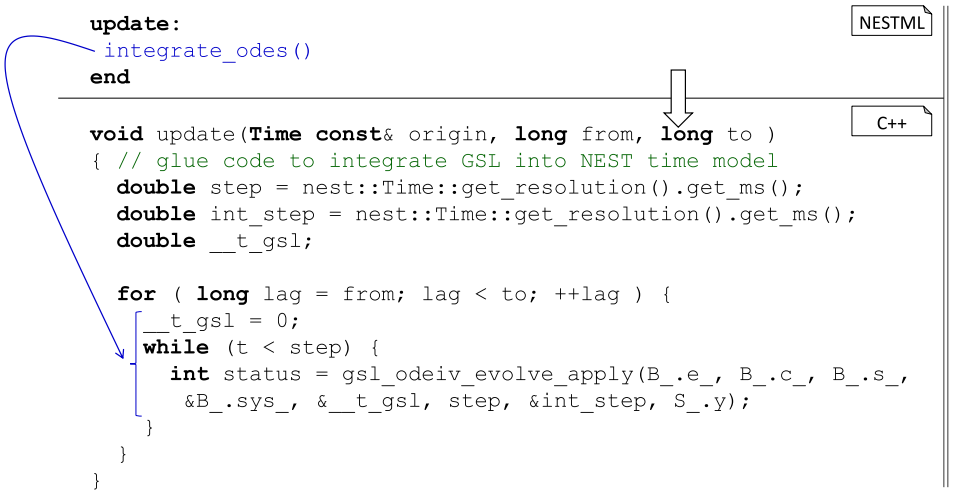


Abbildung 8.13: Propagation der Differenzialgleichungen aus dem `equations`-Block

Somit ist der NEST-Generator in der Lage, alle Sprachkonzepte mit Ausnahme von `shape`-Funktionen zu verarbeiten, die in Kapitel 5 vorgestellt wurden. Als Nächstes wird vorgestellt, wie `shape`-Funktionen in eine Propagatormatrix oder eine Menge von Differenzialgleichungen transformiert werden, sodass eine individuelle Behandlung dieser Modellierungselemente im NEST-Codegenerator überflüssig wird.

8.1.6 Das Analyseframework für Differenzialgleichungen

Hängt eine der Differenzialgleichungen im `equations`-Block in einem Neuron von mindestens einer `shape`-Funktion ab, wird eine Analyse und Modelltransformation durchgeführt, um entweder eine exakte Lösung zu berechnen oder das Gleichungssystem in eine für das GSL-Framework passende Form zu transformieren. In beiden Fällen werden alle `shape`-Funktionen durch ein äquivalentes Gleichungssystem substituiert.

Um die mathematische Analyse der Gleichungen durchzuführen, wurde das *SymPy*-Framework [MSP+16] ausgewählt. *SymPy*[3] ist ein leichtgewichtiges, Python-basiertes Framework für die symbolische Algebra. Im Unterschied zu kommerziellen Tools wie *Matlab* [HV16], *Mathematica* [AB16] oder Modelica [Fri10] ist *SymPy* frei unter einer *BSD*-Lizenz verfügbar, womit auch die generierten Dateien frei verwendet werden können [KR14]. *SymPy* lässt sich mithilfe der Konsolen-API mit anderen NESTML-Modulen

[3] http://www.sympy.org/en/index.html

nahtlos integrieren. Desweiteren unterstützt *SymPy* alle Anforderungen, um die nötige Analyse und Umformung der `shape`-Funktionen und Gleichungen durchzuführen.

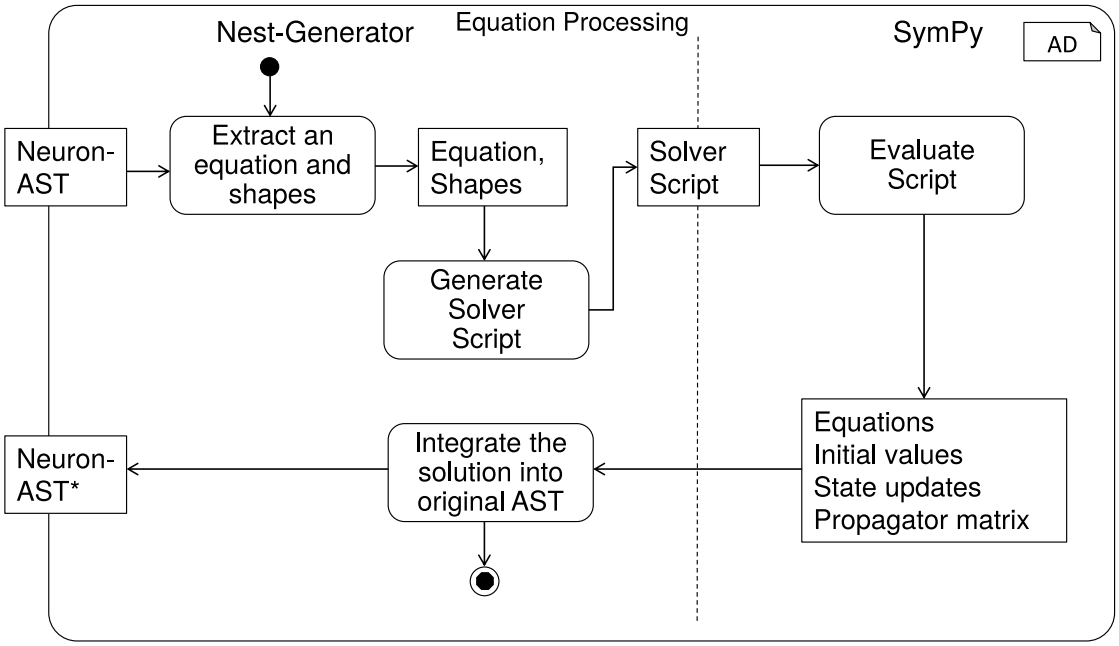


Abbildung 8.14: Verknüpfung des NEST-Generators mit der *SymPy*-Laufzeitumgebung.

Abbildung 8.14 skizziert den Ablauf der Analyse der Differenzialgleichungen mithilfe des entwickelten Analyseframeworks, das auf *SymPy* basiert. Die wesentlichen Schritte dieser Analyse sind die folgenden:

1. Die Gleichungen aus dem `equations`-Block werden auf Abhängigkeiten untersucht. Nur Gleichungen, die von mindestens einer `shape`-Funktion und von keiner anderen Differenzialgleichung abhängen, werden genauer analysiert. Alle anderen Gleichungen werden direkt mit dem GSL-Framework gelöst.

2. Auf Basis einer Differenzialgleichung mit Abhängigkeit von mindestens einer `shape`-Funktion wird ein modellspezifisches Lösungsskript generiert. Somit können je nach Neuronenmodell mehrere solchee Skripte generiert und evaluiert werden. Die Funktionsweise eines einzelnen Skripts ist in Abbildung 8.15 genau spezifiziert.

3. Generierte *SymPy*-Scripte werden während der Laufzeit des Generators evaluiert.

4. Während der Ausführung produziert das *SymPy*-Lösungsskript Dateien, deren Inhalt syntaktisch zur *ProceduralDSL*-Sprache konform sind. Somit kann der *ProceduralDSL*-Parser diese Dateien zu einem validen *ProceduralDSL*-AST konvertieren.

5. Die *ProceduralDSL*-ASTs können nahtlos mit dem AST des initialen Neurons zusammengeführt werden. Auf diese Art und Weise werden neue Variablen, Gleichungen und Anweisungen ins Ursprungsneuron integriert.

6. Der transformierte NESTML-AST* wird als Ergebnis zurückgegeben. Er enthält eine exakte bzw. numerische Lösung für die Propagation der Differenzialgleichungen.

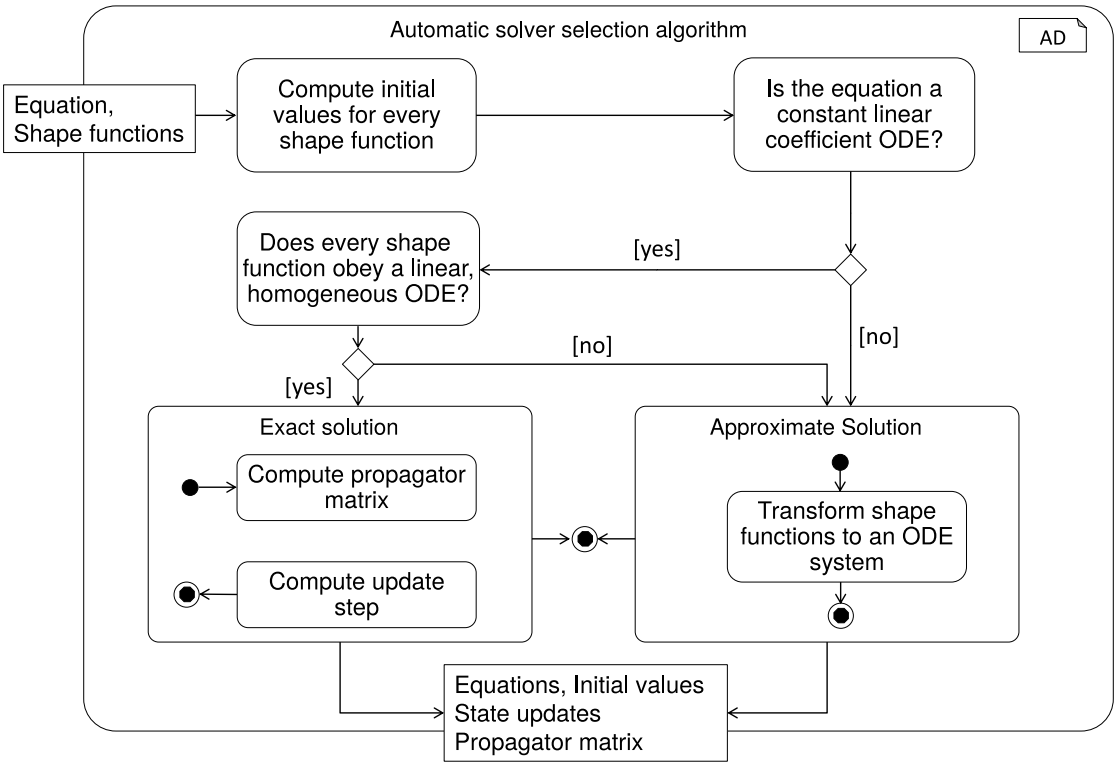


Abbildung 8.15: Analyseverfahren für eine Differenzialgleichung der Form $V' = \text{RHS}$. Die RHS hängt in diesem Fall von einer nicht leeren Menge von `shape`-Funktionen ab.

Gleichungen, die mindestens eine *shape*-Funktion enthalten, müssen entweder exakt gelöst werden oder in eine für das GSL-Framework kompatible Form transformiert werden. Um dies festzustellenn, wird der als *SymPy*-Skript implementierte Algorithmus ausgewertet. Abbildung 8.15 fasst die essenziellen Schritte des Algorithmus zusammen, der für eine Gleichung mit `shape`-Funktionen ausgeführt wird. Die wesentlichen Schritte dieses Verfahrens lassen sich wie folgt zusammenfassen:

1. Die Eingabe für den Algorithmus ist eine Gleichung und eine nicht leere Menge von `shape`-Funktionen. Um validen Code in der Programmiersprache Python zu erhalten, werden die Aufrufe von `conv`-Funktionen in der rechten Seite der Gleichung durch die darin vorkommende `shape`-Funktion ersetzt. Da das zweite Argument des *conv*-Aufrufs für die Analyse unerheblich ist, bleibt die Semantik der Gleichungen dadurch erhalten.

2. Für alle `shape`-Funktionen werden alle Anfangswerte bestimmt, die für die Propagation der Differenzialgleichungen innerhalb des `update`-Blockes notwendig sind. Da die `shape`-Funktionen bereits als Funktion vorliegen, können sie einfach für den Wert `0` ausgewertet werden.

3. Der Algorithmus prüft, ob
   a) die vorliegende Gleichung eine lineare Gleichung mit konstanten Koeffizienten erfüllt. Diese Prüfung wird durch die zweifache Ableitung der rechten Seite der Differenzialgleichung durchgeführt. Für eine Differenzialgleichung $V' = \frac{V}{\tau} + \frac{I(t)}{C}$ wird der folgende Test ausgeführt: $\frac{d}{dV}\frac{d}{dt}(\frac{V}{\tau} + \frac{I(t)}{C}) == 0$.
   b) alle `shape`-Funktionen eine lineare homogene Differenzialgleichung erfüllen.

4. Wenn beide Bedingungen erfüllt sind, kann die Differenzialgleichung exakt und inkrementell [PBI+16], gelöst werden.
   - Eine Differenzialgleichung mit linearen Koeffizienten kann effizient durch Berechnung der Propagatormatrix gelöst werden (vgl. Abschnitt 2.3). Auf dieser Basis wird auch der Aktualisierungsschritt bestimmt.
   - Ansonsten wird für die Propagation das GSL-Framework verwendet.

5. Wenn eine Bedingung nicht erfüllt ist, wird die Gleichung für die Lösung mit dem GSL-Framework vorbereitet. Jede `shape`-Funktion wird in ein äquivalentes Gleichungssystem transformiert. Zusammen mit den bereits bestimmten Anfangswerten stellt diese neue Form eine äquivalente Schreibweise dar. Die neue Darstellung kann vom GSL-Framework effektiv ausgewertet werden.

Das Resultat des Algorithmus kann wieder als valides NESTML-Modell angesehen werden und das transformierte Modell kann mit dem NESTML-PrettyPrinter serialisiert werden. Dadurch kann der Modellierer die Lösung einsehen und nachvollziehen. Diese Möglichkeit steigert das Vertrauen des Modellierers in die vom NEST-Generator durchgeführten Analysen. Der in diesem Kapitel beschriebene NEST-Generator ist in der Lage, das transformierte Modell direkt zu verarbeiten und wird somit vollständig wiederverwendet.

Das Analyseframework ist modular entworfen und implementiert. Die Eingabe des Algorithmus beschränkt sich ausschließlich auf die Definition der Differenzialgleichungen

und `shape`-Funktionen in textueller Form. Durch die syntaktische Kompatibilität der `ExpressionsDSL` zu Python-Ausdrücken können die Gleichungen und Funktionen direkt mit Python-Werkzeugen verarbeitet werden. Dies hat den Vorteil, dass Analysen unabhängig von NESTML in anderen Kontexten wiederverwendet werden können. Die lose Kopplung des Anslyseframeworks und der restlichen Teile von NESTML ermöglicht die unabhängige Weiterentwicklung und ein einfaches Testen des Analyseframeworks.

8.1.7 Integration von handgeschriebenem Code

Für Anwendungsfälle, in denen eine manuelle Optimierung von bestimmten Aspekten der Neuronenimplementierung unabdingbar ist, bietet NESTML eine leichtgewichtige und transparente Integration von handgeschriebenem Code.

In NESTML wird die Integration von handgeschriebenem Codes mithilfe des GAP-Patterns [Fow10, Vli98] unterstützt. Die Wahl dieses Musters geschah aufgrund von zwei wesentlichen Vorteilen des GAP-Patterns im Vergleich zu den anderen alternativen Erweiterungsmechanismen (vgl. [GHK+15a, GHK+15b] für einen ausführlichen Vergleich der unterschiedlichen Alternativen der Integration von handgeschriebenem und generierten Code). Der erste Vorteil ist, dass Neuronenmodelle mit plattformspezifischen Informationen nicht überladen werden. Der zweite Vorteil besteht in einer sauberen Trennung der handgeschriebenen Implementierung des Neuronenmodells von generiertem Code. Dies erlaubt eine modulare und unabhängige Entwicklung des Codegenerators und des handgeschriebenen Codes. Ein wesentlicher Nachteil des GAP-Patterns besteht darin, dass die Erweiterung der Schnittstelle von generierten Klassen nicht erlaubt ist. Dennoch ist es möglich, die Schnittstelle der generierten Klassen auf andere Weise zu erweitern. Die NESTML-Sprache unterstützt die Schnittstellenerweiterung mithilfe der eingebetteten Aktionssprache *ProceduralDSL*. *ProceduralDSL* ermöglicht es, Methoden direkt als Teil des Neuronenmodells zu implementieren. Somit bietet die Aktionssprache konzeptionell eine bessere Lösung an, da alle Referenzen stets auf der Modellebene bleiben und dadurch keine Abhängigkeiten zur technischen Realisierung entstehen.

Abbildung 8.16 demonstriert die Umsetzung des GAP-Patterns in NESTML an einem Beispiel. Der NEST-Generator wird in diesem Fall so parametrisiert (vgl. Abschnitt 8.3 für die Liste aller Generatorparameter), dass aus NESTML-Neuronen Klassen mit einem `top`-Postfix generiert werden. Diese Klasse erweitert die Klasse, die der Benutzer des Generators als handgeschriebene Klasse zur Verfügung stellt. Im vorliegenden Beispiel erweitert die handgeschrieben Klasse `rc_neuron` die generierte `rc_neuron_top`. Die Erweiterung ermöglicht beispielsweise die Überschreibung der `update`-Methode, die für das Laufzeitverhalten des Neurons verantwortlich ist. Auf diese Weise ist es auch möglich, andere numerischen Frameworks zu integrieren und zu benutzten.

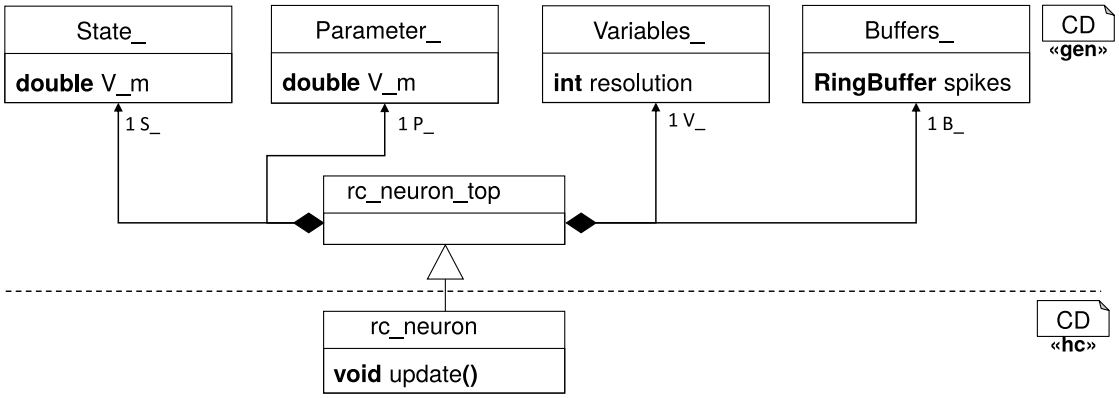


Abbildung 8.16: Erweiterung der generierten Neuronenklasse durch zwei handgeschriebene Methoden.

8.2 Generierung des Modulintegrationscodes

Bis jetzt wurde die Generierung der NEST-Implementierung aus modellierten Neuronen vorgestellt. Dieser Abschnitt erläutert den Integrationscode, mit dem Neuronenmodelle dynamisch (d.h. ohne Neuinstallation der NEST-Umgebung) in den Simulator integriert werden können. Abschnitt 7.2 erläutert die Verwendung des generierten Moduls bereits. Im Unterschied zum bisherigen Generierungsansatz wird der Integrationscode ausschließlich auf Basis der Datei- und Neuronennamen und nicht auf Basis eines ASTs generiert.

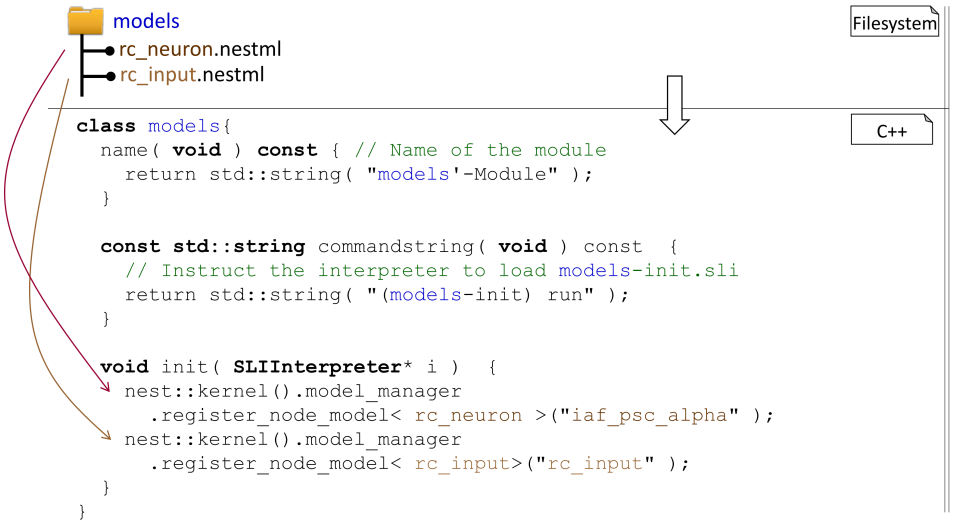


Abbildung 8.17: Exemplarische Ableitung des modellabhängigen Abschnittes der NEST-Modulintegrationsklasse.

Auch beim Integrationscode ist der Kern der Implementierung in einer C++-Klasse gekapselt. Abbildung 8.17 zeigt exemplarisch die Ableitung dieser Modulklasse. Die relevante Information besteht aus der Liste von Neuronen und des Speicherorts der entsprechenden NESTML-Dateien, die diesem Fall im `models`-Verzeichnis gespeichert sind. Diese Information wird aus den vorliegenden Dateien extrahiert und an den Modulgenerator übergeben. Das Generierungstemplate traversiert die Liste mithilfe der in den Freemarker-Templates eingebauten Kontrollsprache und erzeugt Referenzen zu den generierten Neuronendateien.

Zusätzlich zur Modellintegrationsklasse ist eine CMake-Konfigurationsdatei notwendig. Diese Konfigurationsdatei kann bis auf die Referenzen zu generierten Dateien und Modulnamen mit einem ansonsten statischen Template erzeugt werden. Daher verwendet der CMake-Generator ausschließlich die templatebasierte Strategie [CH03, CH06b].

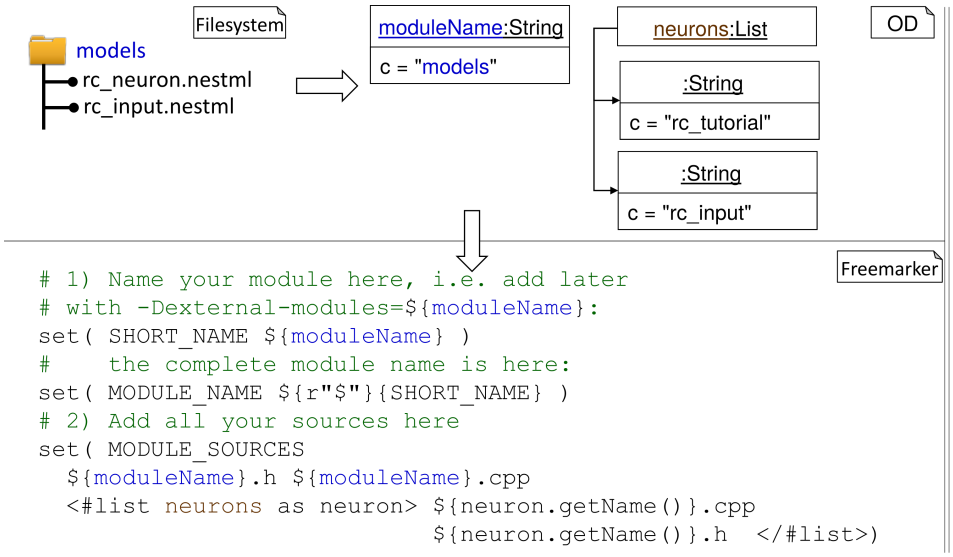


Abbildung 8.18: Ausschnitt des Generatortemplates für die Erzeugung der modellabhängigen CMake-Konfigurationsdatei.

Abbildung 8.18 zeigt einen Ausschnitt des CMake-Generators, der die wesentliche modellabhängige Funktionalität der Generierung beinhaltet. Die Eingabe für den Generator wird aus der vorliegenden Dateikonstellation abgeleitet. Diese wird in Form von Strings und Listen an das Generierungstemplate übergeben. Am Ende wird eine auf die vorliegenden Neuronenmodelle zugeschnittene CMake-Konfigurationsdatei erstellt.

8.3 Konsolen-basierte Schnittstelle für das NESTML-Frontend

Um NESTML in existierende neurowissenschaftliche Werkzeuge zu integrieren [GKRS06] und Qualitätssicherung auf einem *Continuous Integration* System [Duv07, SB14] zu erlauben, wurde eine Konsolen-basierte Schnittstelle für die in dieser Ausarbeitung entwickelte Implementierung erstellt. Diese Schnittstelle wird als NESTML-Frontend bezeichnet.

Ein exemplarischer Aufruf des NESTML-Frontends, der als ein Java-Archiv zur Verfügung steht, sieht wie folgt aus:

```
1 java nestml.jar --target_path $HOME/output $HOME/nestml/models
```

Die folgende Liste fasst die verfügbaren Optionen des NESTML-Frontend zusammen:

`$model_path`: Dieses Argument beim Frontendaufruf ist der einfache Pfad, an dem NESTML-Dateien gespeichert sind. Für das Auflösen der Importanweisungen (vgl. Abschnitt 6.2) wird der Pfad als Modellpfad (engl: modelpath) verwendet. Dieses Argument ist das einzige obligatorische Argument beim Aufruf des NESTML-Frontends.

`--target $path`: Mit diesem Argument und einem konkreten Pfad im Filesystem kann der Pfad angegeben werden, an dem die generierten Dateien gespeichert werden sollen. Wenn kein Argument angegeben ist, werden die generierten Dateien im `build`-Subverzeichnis des Verzeichnisses gespeichert, von dem aus das NESTML-Frontend gestartet wurde.

`--enable_tracing`: Mithilfe dieses Argumentes können Templates, die für die Generierung des Zielsystems verwendet wurden, nachverfolgt werden. Dafür werden im generierten Code spezielle Kommentare generiert.

`--dry-run`: Mithilfe dieses Arguments wird die Codegenerierung unterbunden. Dabei werden aber alle Modelle aus dem `$model_path` geparst und alle Kontextbedingungen geprüft.

`--json_log $filename`: Das NESTML-Frontend produziert eine weitreichende diagnostische Ausgabe während seiner Ausführung. Im Standardfall wird diese Information auf die Konsole geschrieben. Um eine Weiterverbreitung dieser Information zu vereinfachen, kann das NESTML-Frontend mithilfe dieses Arguments angewiesen werden, die Ausgabe im JSON-Format in eine benutzerdefinierte Logdatei zu schreiben und so eine automatisierte Auswertung zu ermöglichen.

`--module_name $name`: Im Standardfall wird der Name des Moduls aus dem $model_path-Argument abgeleitet. Dem NESTML-Frontend kann mit diesem Argument ein anderer Name für das generierte Modul (vgl. Abschnitt 8.2) übergeben werden. Dabei bestimmt der `$name`-Parameter den zu verwendenden Modulnamen.

`--hc $path`: Der Pfad zu handgeschriebenem Code.

`--help`: Mithilfe dieses Arguments können Information über die Benutzung des NESMTL-Frontends ausgegeben werden.

8.4 Zusammenfassung

In diesem Kapitel wurde die Generierung der ausführbaren Implementierung aus NESTML-Neuronenmodellen für den NEST-Simulator vorgestellt. Bei der Konzeption des generierten Codes wurde stets darauf geachtet, dass der resultierende Code sowohl effizient als auch möglichst nah an der bereits existierenden Implementierung von Neuronenmodellen in NEST ist. Diese Angleichung vereinfacht zum einen das Verständnis des generierten Codes für die NEST-Entwickler, zum anderen erhöht dies die Akzeptanz des Generierungsframeworks. Aus diesem Grund wurde in der aktuellen Version von NESTML die Angleichung an existierende Implementierungsmuster der effizienteren Codegenerierung vorgezogen, da dies die Akzeptanz des NEST-Generators bei den NEST-Entwicklern und NEST-Benutzern steigert. Dank des modularen Aufbaus von NESTML können weiter Optimierungen in zukünftigen Versionen nachgerüstet werden.

Die Verbindung der Modellierungssprache NESTML mit dem Codegenerierungsframework erlaubt es, sowohl die existierenden als auch neue Neuronenmodelle ausschließlich mit NESTML auszudrücken. Somit wird die manuelle Implementierung neuer Neuronenmodelle in Form von C++-Code überflüssig. Schließlich erlaubt die entwickelte Konsolenschnittstelle eine flexible und transparente Integration des NESTML-Frontends in bereits existierende neurowissenschaftliche Werkzeugen und zukünftig entwickelte Systeme und Plattformen wie das Kollaborationsportal des Human Brain Project[4].

[4] `https://collab.humanbrainproject.eu`

Kapitel 9

Evaluierung von NESTML

Dieses Kapitel stellt die wichtigsten Resultate aus Umfragen zur Evaluierung von NESTML vor. Insbesondere ging es dabei um die entwickelten Werkzeuge (vgl. Kapitel 8) und das in dieser Ausarbeitung vorgestellte NESTML-Tutorial (vgl. Kapitel 7). Die Konzeption und Umsetzung von NESTML und den zugehörigen Werkzeugen wurde durch regelmäßige Evaluationen durch die Anwender und Entwickler des NEST-Simulators begleitet. Diese beiden Gruppen wirkten in einer engen und agilen Kooperation auch direkt auf die Entwicklung ein. Dabei fanden regelmäßige Präsentationen der NESTML-Sprache und -Werkzeuge mit dem Ziel statt, die Sprache besser an die Bedürfnisse der Anwender anzupassen.

9.1 NESTML-Workshops

Der modellbasierte Ansatz wurde im Kontext dieser Ausarbeitung auf zweierlei Arten evaluiert. Zum einen wurde eine Menge bereits existierender Neuronenmodelle aus NEST[1] in NESTML spezifiziert, um den Anwendern der Modellierungssprache charakteristische Modellbeispiele aus der neurobiologischen Praxis zur Verfügung zu stellen. Dabei wurden im Laufe der Entwicklung alle *Integrate-and-Fire*-Neuronen, die in NEST zur Verfügung stehen, mithilfe von NESTML modelliert [PBE+17b].

Zum anderen wurden wurden zwei dedizierte NESTML-Workshops organisiert und durchgeführt, um potenzielle Nutzer in der neuen Modellierungssprache zu schulen und so die Akzeptanz bei potenziellen Nutzern zu steigern. Im Rahmen dieser Workshops [PBE+15, PBE+16] wurde die NESTML und ihre Werkzeuginfrastruktur detailliert vorgestellt. Dabei wurden sowohl die sprachlichen Konstrukte und Konzepte zur Modellierung von Neuronen als auch die Benutzung der Werkzeuge zum Generieren von C++Code sowie dessen Benutzung ausführlich an praktischen Beispielen demonstriert. Die Diskussionen während dieser Workshops führten zu einer signifikanten Veränderung und Verbesserung der konkreten Syntax von NESTML. Anschließend wurden Benutzerumfragen während verschiedener Workshops durchgeführt und ausgewertet.

Das Benutzerfeedback hatte starken Einfluss auf die Weiterentwicklung von NESTML.

[1] https://github.com/nest/nest-simulator/blob/master/models

Folgend werden beispielhaft zwei Änderungen an der NESTML-Syntax aufgelistet, die aufgrund dieses Feedbacks umgesetzt wurden:

- Die Darstellung der Differenzialgleichungen wurde mehrfach überarbeitet. Initial war es möglich, einen `equation`-Block innerhalb des `update`-Blockes bzw. einen Block für jede Differenzialgleichung zu spezifizieren. Anstatt dieser Schreibweise wurde ein dedizierter `equations`-Block im Rumpf des Neurons eingeführt. Mit der `integrate_odes`-Funktion werden die Gleichungen aus dem `equations`-Block dann im `update`-Block propagiert.

- Das Einheitensystem wurde überarbeitet. Zusammengesetzte physikalische Einheiten wurden erst auf Benutzerwunsch eingeführt. Ebenso wurde die verkürzte Schreibweise für die Einheiten der Form: `ms**-1` eingeführt, die jetzt als `1/ms` spezifiziert werden können.

Um NESTML als neue Modellierungssprache für den NEST-Simulator zu etablieren, ist es notwendig, eine kritische Masse an Benutzern zu erreichen [MH02, LC06]. Die Workshops dienten deshalb neben der Möglichkeit zum Sammeln von Feedback auch als Plattform um neue Benutzer auf NESTML aufmerksam zu machen und Fragen direkt beantworten zu können. Eine weitere Möglichkeit, die in diesem Kontext bereits rege genutzt wird ist die NEST User Mailingliste[2], auf der auch Fragen zu NESTML gestellt und beantwortet werden.

Insgesamt wurden 27 Fragebögen während der beiden NESTML-Workshops ausgefüllt und ausgewertet. Das obere Histogramm in Abbildung 9.1 fasst die akademischen Grade der Teilnehmer zusammen. Überwiegend handelte es sich um Studierende mit Masterabschluss, die sich in einem Phd-Programm mit neurowissenschaftlichen Schwerpunkt befanden. Außerdem waren Bachelorstudenten, Postdoktoranden und ein Professor vertreten. Das untere Histogramm fasst die Erfahrung der Teilnehmer mit dem NEST-Simulator zusammen. Die Erfahrung erstreckt sich über alle Ausprägungen mit klarem Fokus auf den Erfahrungsstufen Little (wenig) und Medium (mittel).

Ein wichtiger Teil der beiden Workshops war eine praktische Aufgabe, die mithilfe von NESTML gelöst werden sollte. Die Aufgabe bestand darin, ein in einer wissenschaftlichen Publikation definiertes Neuronenmodell in Form eines NESTML-Modells zu spezifizieren. Dieses Neuronenmodell sollte anschließend im NEST-Simulator simuliert werden, um die publizierten Resultate zu reproduzieren (vgl. Abschnitt C.2). Beim Neuronenmodell handelte es sich um das Izhikevich-Neuronenmodell [Izh03, Izh04], das aus zwei Differenzialgleichungen und einer Vorschrift für die Aktualisierung des Neuronenzustandes besteht. Alle Teilnehmer waren in der Lage innerhalb einer Stunde, sowohl das Neuronenmodell zu spezifizieren als auch die veröffentlichten Spike-Trains mithilfe des generierten NEST-Quellcodes zu reproduzieren.

[2] http://www.nest-simulator.org/community/

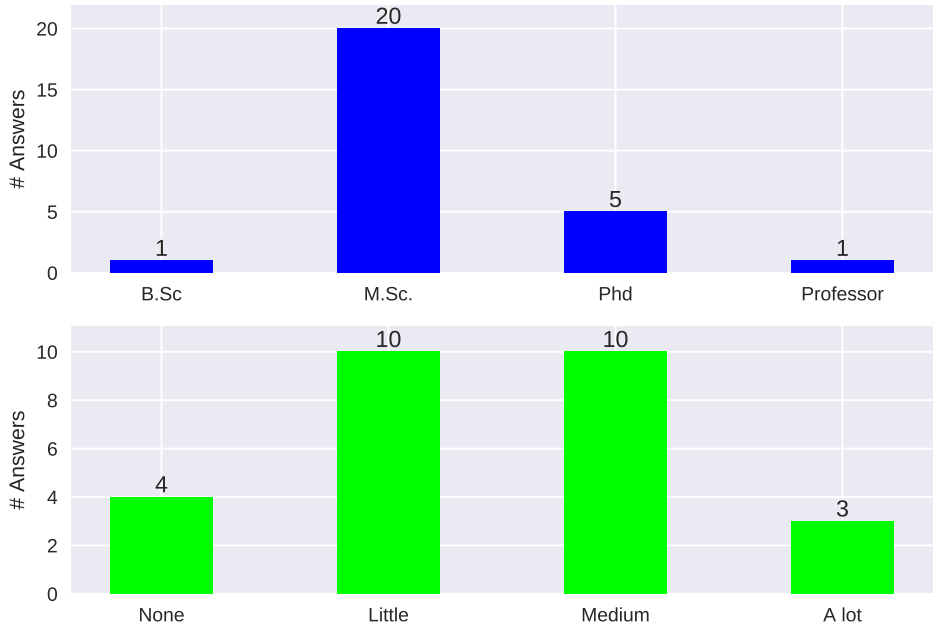


Abbildung 9.1: Zusammenfassung der akademischen Grade der Teilnehmer der Evaluierung (oben) und die Erfahrung der Teilnehmer mit dem NEST-Simulator (unten).

Abbildung 9.2, Abbildung 9.3, Abbildung 9.4, Abbildung 9.5, Abbildung 9.6 und Abbildung 9.7 fassen die Resultate der Auswertung zusammen. Dabei wurden die in der Bildunterschrift erläuterten Fragen auf einer Skala von 0 (trifft nicht zu) bis 5 (trifft voll zu) beantwortet.

Wähnend der Evaluierung wurden sowohl die NESTML-Sprache (vgl. **Q1**) als auch die Funktionsweise der NESTML-Werkzeuge (vgl. **Q2**) als sehr gut verständlich eingestuft. Die meisten Befragten gaben an, dass das Erstellen eines NESTML-Modells einfacher ist als das Schreiben desselben Modells als eine C++-Erweiterung für den NEST-Simulator (vgl. **Q3**). Die Spezifikation der mathematischen Ausdrücke mit NESTML wurde einfacher bewertet als die Spezifikation mit der C++-Programmiersprache (vgl. **Q5**). Die Fehlerinformationen, die durch das NESTML-Werkzeug ausgegeben werden (vgl. **Q4**), wurden befriedigend bewertet. Schließlich fanden alle Teilnehmer das Tutorial hilfreich (vgl. **Q6**).

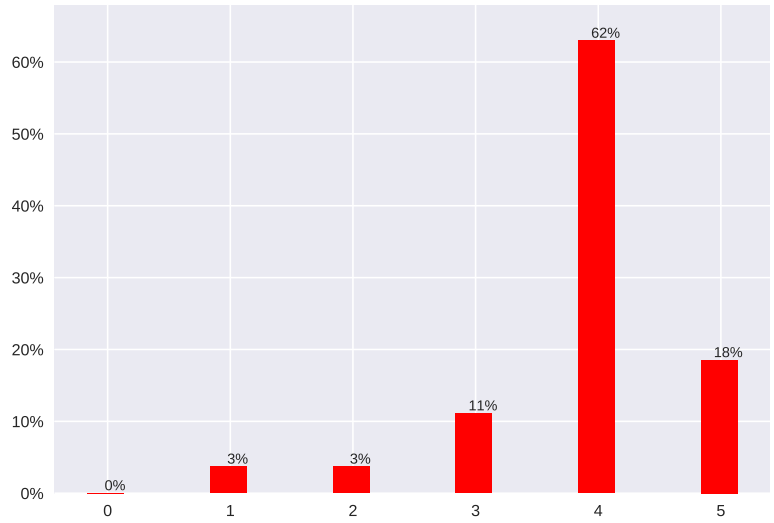


Abbildung 9.2: **(Q1)** Die Syntax von NESTML ist klar und verständlich.

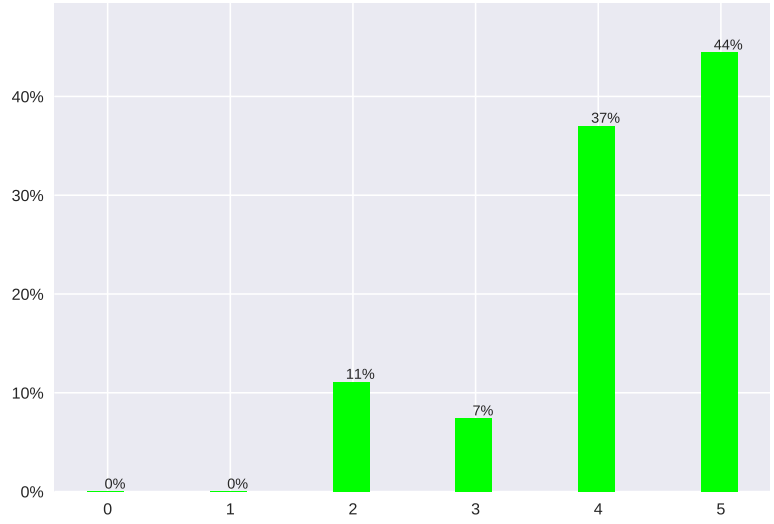


Abbildung 9.3: **(Q2)** Es ist einfach, die Funktionsweise von NESTML und den Werkzeugen zu verstehen.

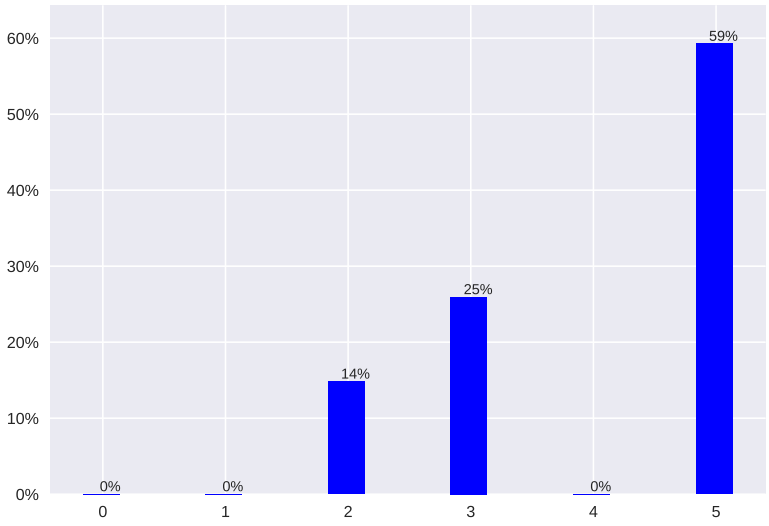


Abbildung 9.4: **(Q3)** Das Erstellen von NESTML-Modellen ist einfacher als das Schreiben von C++-Code für den NEST-Simulator.

9.2 Validität von der Evaluierung

Aufgrund der Struktur der beiden Workshops und der geringen Teilnehmerzahl war es nicht möglich, die Teilnehmer in eine randomisierte Test- bzw. Kontrollgruppe aufzuteilen. Somit ist sowohl die interne als auch externe Validität der Untersuchung gefährdet.

Die interne Validität ist durch folgenden Beobachtungen gefährdet: Teilnehmern fehlte die Erfahrung im Umgang mit NESTML-Werkzeugen, die Teilnehmer hatten jedoch Erfahrung mit herkömmlichen Programmiersprachen wie C, C++ und Python. Diese Beobachtung erklärt die Verzerrung der Resultate beim direkten Vergleich von NESTML mit der C++-Programmiersprache (vgl. **Q2** und **Q5**). Eine umfragenbasierte Untersuchung könnte selbst ein Problem darstellen, da jeder Teilnehmer die Antwortskalen anders interpretiert. Alle Teilnehmer wussten zudem, dass die Workshops unter anderem zum Zweck der Evaluierung von NESTML durch die NESTML-Entwickler organisiert wurden. Dadurch entsteht eine Gefährdung aufgrund des kompensatorischen Wettstreiteffekts [BD07]. Allein schon eine gezielte Einladung könnte demnach dazu führen, dass sich Teilnehmer sehr engagiert mit NESTML befassen, was die Wahrnehmung von NESTML verzerren kann. Da es keine randomisierten Kontrollgruppen gab, sind diese Effekte quantitativ jedoch schwer abschätzbar.

Die Gefährdung der externen Validität ist aufgrund der kleinen Stichprobe der aus-

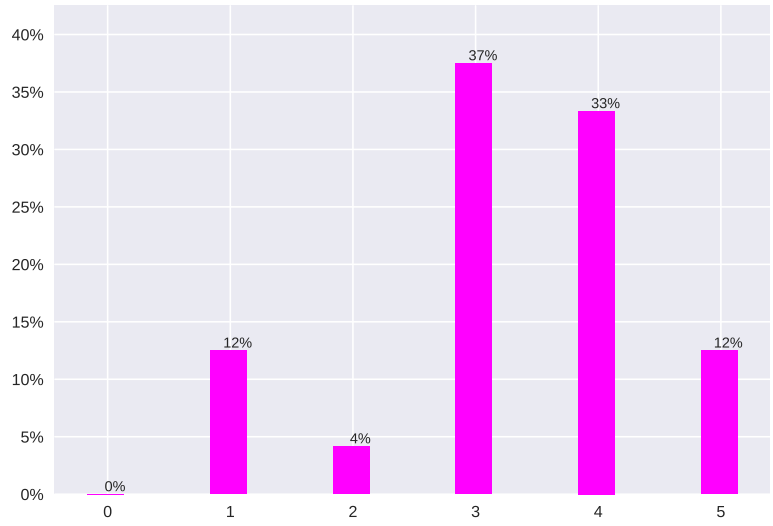


Abbildung 9.5: **(Q4)** Die Fehlerausgabe, die während der Modellverarbeitung produziert wird, ist hilfreich.

gewerteten Fragebögen gegeben. Daher ist eine Generalisierung der beobachteten Ergebnisse auf die Grundgesamtheit nicht ohne weiteres möglich. Dieses Risiko wurde dadurch verkleinert, dass die Umfragen in zwei unabhängigen und zeitversetzten Workshops durchgeführt wurden.

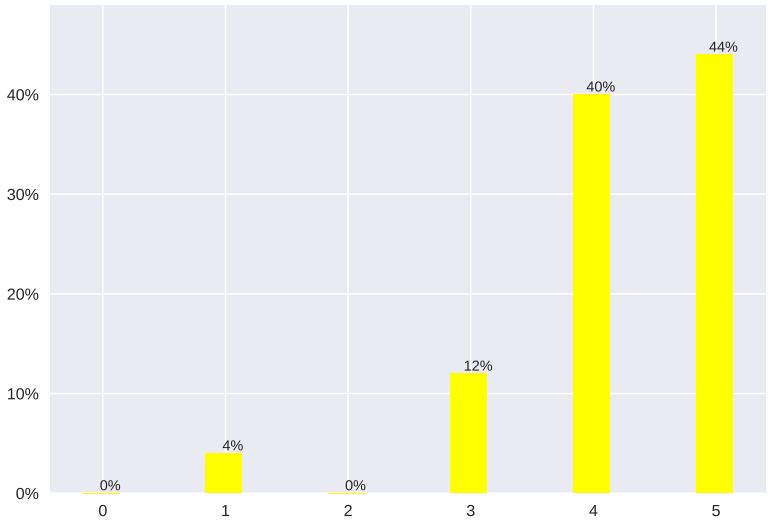


Abbildung 9.6: **(Q5)** Die Spezifikation von mathematischen Termen in NESTML ist einfacher als in der Programmiersprache C++.

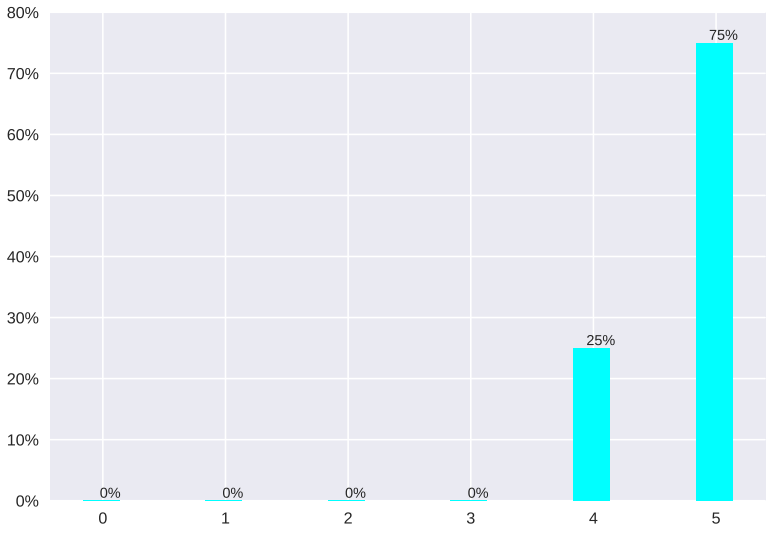


Abbildung 9.7: **(Q6)** Das Tutorial war hilfreich.

Kapitel 10

Diskussion und Zusammenfassung

Im Rahmen der vorliegenden Ausarbeitung wurde die neue domänenspezifische Sprache NESTML konzipiert und mithilfe der MontiCore Workbench umgesetzt. Der Entwurf von NESTML basiert einerseits auf einer detaillierten Analyse der Nutzungsszenarien von Neuronenmodellen im Simulator NEST und andererseits auf den Anforderungen an eine Neuronenmodellierungssprache, die sich aus einer ausführlichen Literaturrecherche ergaben. Die Erfüllung dieser Anforderungen war beim Entwurf und der Umsetzung der Modellierungssprache wichtig, um eine einfach zu erlernende und wiederverwendbare Sprache zu erstellen. Dabei war insbesondere die Wiederverwendung sowohl auf Grammatik- als auch auf Modellebene von großer Bedeutung.

10.1 Evaluierung der Sprachanforderungen

Dieser Abschnitt diskutiert, ob NESTML alle in Kapitel 3 herausgearbeiteten Sprachanforderungen an eine Neuronenmodellierungssprache erfüllt.

RQ1 Modellierungsstil: Da die NESTML-Sprache als externe DSL entwickelt wurde, konnte deren Modellierungsstil frei gewählt werden. Aufgrund der kompakten Syntax vermitteln NESTML-Modelle ihren Zweck klar und verständlich. NESTML-Modelle haben keinen direkten Bezug zu einer bestimmten Simulationsumgebung. Dieser Bezug wird ausschließlich durch einen Codegenerator hergestellt. Somit wird das Teilen von NESTML-Modellen vereinfacht. Dennoch können die Vorteile einzelner Simulatoren durch handgeschriebene Codeerweiterungen vollständig ausgenutzt werden. NESTML-Modelle sind stets in sich geschlossen formuliert. Auf der Modellebene sind keine Referenzen zu nicht-Modellelementen erlaubt. Somit können die NESTML-Modelle modular verarbeitet werden.

RQ2 Konkrete Syntax: Die Syntax von NESTML wurde mit dem Fokus auf die Verständlichkeit entwickelt. NESTML wurde primär für die Modellierung von biologischen Punktneuronen konzipiert und umgesetzt. Daher wurden nur diejenigen fachlichen Konzepte in die NESTML-Sprache integriert, die sich für die Modellierung von Punktneuronen eignen. Da NESTML als externe DSL implementiert ist, konnte die syntaktische Repräsentation dieser Elemente frei gewählt und als direkte Sprachelemente implementiert werden. NESTML ist modular und kompositionell aus verschiedenen Subsprachen

aufgebaut. Jede Subsprache enthält dabei eine minimale Anzahl der Sprachkonstrukte. Somit ist NESTML auf der einen Seite übersichtlich und einfach zu erlernen. Auf der anderen Seite kann NESTML über die vorgesehenen Mechanismen der Sprachkomposition erweitert werden. NESTML verwendet eine Notation, die an die konkrete Syntax von Python angelehnt ist. Für die Modellierung von Differenzialgleichungen wurde eine mathematische Notation entwickelt. Diese Notation erlaubt es, Gleichungen unkompliziert in NESTML-Modelle zu übertragen. Die Möglichkeit, physikalische Einheiten als Datentypen und Literale zu verwenden, ermöglicht es, die Konsistenz von Neuronenmodellen bereits zur Modellierungszeit zu sichern. Neuronen und Komponenten sind in unterschiedliche Blöcke strukturiert. Jeder Block wird durch ein passend gewähltes Schlüsselwort eingeleitet. Die Bezeichnung des Blockes vermittelt die Semantik des Blockes. Aufgrund der unterschiedlichen Schlüsselwörter können die Blöcke sehr einfach voneinander unterschieden werden.

RQ3 Wiederverwendung: Die Wiederverwendung wird in NESTML-Modellen auf zwei Arten unterstützt. Zum einen können explizite und eindeutig definierte Komponenten zur Verfügung gestellt werden. Durch die klare Definition der öffentlichen Schnittstelle von Komponenten, die explizit definiert ist, kann die innere Implementierung verborgen bleiben. Zum anderen können Neuronen an sich durch eine Vererbungsbeziehung erweitert und wiederverwendet werden. Dabei können sowohl Variablen in unterschiedlichen Blöcken ergänzt bzw. überschrieben werden, als auch das dynamische Verhalten angepasst werden, das durch die `update`- und `equations`-Blöcke spezifiziert ist.

RQ4 Metamodell: Das von MontiCore auf Basis der Grammatik generierte Metamodell ist per Konstruktion modular aufgebaut [KRV10]. Die sprachliche Dekomposition von NESTML ermöglicht es, die Subsprachen und das aus diesen Sprachen generierte Metamodell gezielt und effizient anpassen. Alle generierten und handgeschriebenen Werkzeuge sind sprachspezifisch und modular aufgebaut. Das erlaubt es, die Subsprachen und deren Analysewerkzeuge individuell zu entwickeln, zu testen und einzusetzen. Die Verzahnung der Sprachkomponenten findet erst auf der Ebene der Symboltabelle statt.

RQ5 Codegenerierung: Aufgrund der Modularität und Abstraktion von Neuronenmodellen können verschiedene Plattformen unterstützt werden. Bereits jetzt existiert ein Codegenerator für den NEST-Simulator, vgl. Kapitel 8. Eine aktive Entwicklung des Generators für die Modellierungssprache *LEMS* [PPE+17] findet aktuell statt. NESTML strebt an, möglichst alle Erweiterungen entweder durch die eingebauten Sprachkonstrukte, also auf der Abstraktionsebene der Modelle, oder durch eine Spracherweiterung zu unterstützen. Wenn handgeschriebener Code unabdingbar ist, verwendet NESTML das GAP-Pattern [Fow10, Vli98] für die Integration von handgeschriebenen Erweiterungen. Dieser Integrationsmechamismus ermöglicht es, plattformspezifischen Code in den Codegenerierungsporzess zu integrieren, ohne Neuronenmodelle mit unnötigen technischen Details zu überladen. Modellkommentare aus Neuronen und Komponenten werden während der Codegenerierung ins Zielsystem übernommen. Auf diese Weise wird ein agiler

Entwicklungsprozess unterstützt, da externe Dokumentationsartefakte vermieden werden und die Dokumentation durch den Code der Modelle selbst gegeben ist. Wenn Neuronenmodelle sich ändern, passt sich die Dokumentation automatisch an. NESTML stellt ein modulares Framework zur effizienten Lösung der Differenzialgleichungen zur Verfügung (vgl. Unterabschnitt 8.1.6). Die Lösung der Differenzialgleichungen wurde auf Basis von SymPy umgesetzt. Die entwickelten Analysewerkzeuge sind modular aufgebaut und lassen sich auch außerhalb von NESTML verwenden.

RQ6 Benutzerfreundlichkeit: Die Modellierungssprache NESTML soll in einem bereits existierenden und ausgereiften Softwareökosystem zur Anwendung kommen. Um den Prozess der Integration möglichst einfach zu gestalten, werden bei der Integration von NESTML in den NEST-Simulator die dafür vorgesehenen Erweiterungsschnittstellen bedient. Diese Entkopplung erlaubt eine unabhängige Entwicklung beider Produkte. Für die Benutzung von NESTML steht eine Konsolen-Schnittstelle zur Verfügung. NESTML wird sowohl als Quellcode als auch in Form eines Docker-Containers mit einer Konsolen-API zur Verfügung gestellt. Auf diese Weise werden alle für die Ausführung von NESTML notwendigen Bibliotheken und Frameworks gekapselt. Desweiteren kann NESTML auf diese Weise transparent mit anderen neurowissenschaftlichen Werkzeugen integriert werden, die oft selbst eine Konsolen-API anbieten. Das Tutorial für NESTML wurde in Kapitel 7 vorgestellt. Es wurde mehrfach in unterschiedlichen Workshops evaluiert und als hilfreich eingestuft.

10.2 Zusammenfassung

Diese Ausarbeitung stellt am Anfang die folgende Forschungsfrage auf:

> *Wie muss eine problemadäquate, domänenspezifische Sprache für die Modellierung von biologischen Neuronen aussehen, die sich zur Simulation auf Basis des NEST-Simulators eignet?*

Um diese Frage zu beantworten, wurden zuerst die existierenden Modellierungsansätze für die Spezifikation von Neuronenmodellen analysiert. Dabei wurde festgestellt, dass keine der existieren Modellierungssprachen die Anforderungen ausreichend erfüllt. Da sich die existierenden Modellierungsansätze aufgrund deren technischer Beschaffenheit nicht oder nur schlecht anpassen lassen, wurde eine neue domänenspezifische Modellierungssprache NESTML entwickelt und folgende Forschungsfragen adressiert:

- *FF1: Wie sieht eine leichtgewichtige und einfach zu erlernende Neuronenbeschreibungssprache aus?* Diese Frage ist durch die Entwicklung von NESTML beantwortet. Diese Modellierungssprache enthält nur eine beschränkte Menge von deklarativen Modellierungselementen, die für die Modellierung der Punktneuronen notwendig sind.

- *FF2: Wie können Neuronenmodelle in eine für den NEST-Simulator ausführbare Form übersetzt werden?* Diese Fragestellung wurde durch die Definition eines Codegenerators für den NEST-Simulator adressiert. Dieser Codegenerator ist in der Lage den Quellcode für den NEST-Simulator zu erstellen.
- *FF3: Wie werden die Sprachkomponenten modular und erweiterbar entworfen?* Einerseits ist diese Frage durch den passenden Entwurf der Grammatiken beantwortet, die Erweiterungspunkte zur Verfügung stellen. Andererseits ist der Entwurf der Sprachwerkzeuge wie Kontextbedingungen, PrettyPrinter, Symboltabellen und Codegenerierungswerkzeuge kompositionell aufgebaut. Diese Kompositionalität wird erst durch die Anwendung einer kompositionalen und modularen Infrastruktur ermöglicht, wie sie z.B. von der MontiCore Language Workbench zur Verfügung gestellt wird.
- *FF4: Welche Konzepte helfen dabei, die Neuronenmodelle wiederverwendbar zu machen?* Um diese Fragestellung zu adressieren, wurden zwei unterschiedliche Konzepte vorgeschlagen. Zum einen können wiederverwendbare Komponenten mit wohldefinierten Schnittstellen definiert werden. Zum anderen können Neuronen durch die Vererbungsbeziehung erweitert werden. Die Vererbung erlaubt eine feingranulare Überschreibung bestimmter Aspekte von Neuronen. Damit kann die Wiederverwendbarkeit erhöht und die gleichzeitig die Redundanz in der Modellspezifikation minimiert werden.
- *FF5: Wie sieht die Methodik aus, um Modelle in der neuen DSL zu erstellen und in den NEST-Simulator zu integrieren?* Um die Flexibilität und Agilität der Entwicklung von Neuronen mit NESTML zu gewährleisten, wird NESTML als externes Modul entwickelt. Die Integration des generierten Codes findet mithilfe der dafür vorgesehenen Schnittstellen der Zielplattform statt. Des Weiteren wird der technische Softwarestack in der Form eines Docker-Containers gebündelt. Eine ausführliche Vorgehensweise zur Entwicklung von Neueronenmodellen mit NESTML ist in Kapitel 7 vorgestellt.

Erfahrungen, die während der NESTML Entwicklung gesammelt wurden, wurden direkt in die Weiterentwicklung der MontiCore Workbench integriert. Beispielsweise wurde in diesem Kontext die Unterstützung für die Linksrekursion in MontiCore implementiert. Auch die Stabilität und die Funktionalität von generierten Visitoren und der Symboltablle verbessert und erweitert. NESTML wird als ein Projekt mit offenem Quellcode auf GitHub [PBE+17a] entwickelt und durch die NEST-Community benutzt und weiterentwickelt. Ausgehend vom aktuellen Stand von NESTML können nun die folgenden Erweiterungen vorgenommen werden:

Modelle aus mehreren Segmenten: Die meisten existierenden Neuronenmodelle im NEST-Simulator sind zum Zeitpunkt dieser Arbeit einfache Punktneuronen. Es gibt al-

lerdings keine technische Einschränkung für die Komplexität von Modellen, außer dass diese als eine einzelne C++-Klasse ausgedrückt sein müssen. Es ist in NEST also auch möglich, Neuronenmodelle mit mehreren Segmenten zu simulieren. Dies bedeutet jedoch oft eine große Menge von repetitivem Code, da für jedes der modellierten Segmente ähnlicher C++-Code erstellt werden muss. Der Umfang dieses Codes hängt linear von der Anzahl der modellierten Segmente ab. Eine Erweiterung der NESTML-Sprache und des NEST-Generators würden dieses Problem lösen.

Synapsen: NESTML wurde für die Modellierung der Neuronenmodelle entwickelt, da dies bisher der häufigste Erweiterungsfall für den NEST-Simulator war. Dennoch spielt die Modellierung von Synapsen eine immer wichtigere Rolle um Lern- und Anpassungsprozesse im Gehirn zu verstehen. Bei der manuellen Implementierung von Synapsenmodellen im NEST-Simulator treten grundsätzlich dieselben Probleme wie bei der von Neuronenmodellen auf. Die Implementierung neuer Synapsenmodelle erfordert wieder Kenntnis der internen Struktur von NEST und der Programmiersprache C++, was eine hohe Eintrittsbarriere für das Erstellen eigener Synapsenmodelle darstellt. Ein modellbasierter Ansatz für die Spezifikation von Synapsenmodellen wäre deshalb eine wertvolle Erweiterung für NESTML.

Cross-validation: Auf Basis von NESTML kann eine Menge von Codegeneratoren entwickelt werden, die es erlauben, unterschiedliche Zielplattformen mit demselben Modell zu adressieren Beispielsweise kann auf der Basis von NESTML die Kreuzvalidierung (engl: cross-validation) [BPT94] von Neuronenmodellen in unterschiedlichen Simulatoren und Plattformen ermöglicht werden. Auf ähnliche Weise kann mithilfe von NESTML der Effekt verschiedener numerischer Methoden untersucht werden, ohne die Modellbeschreibung ändern zu müssen.

Die rege Beteiligung an Workshops und Tutorials rund um NESTML, der Austausch der Benutzer über entsprechende Online-Foren, sowie die Ergebnisse der empirischen Fallstudie haben gezeigt, dass NESTML bereits in diesem frühen Entwicklungsstadium erfolgreich verwendet werden kann. NESTML ermöglicht es Wissenschaftlern in der Computational Neuroscience somit, aktuelle und zukünftige Forschungsfragen schneller zu beantworten und durch die neuen Möglichkeiten der Sprache zu besserer Reproduzierbarkeit in diesem Feld beizutragen.

Literaturverzeichnis

[AB16] Martha Abell and James Braselton. *Differential equations with Mathematica*. Academic Press, 2016.

[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.

[AP98] Uri Ascher and Linda Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. Siam, 1998.

[ASU86] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[Bal00] Helmut Balzert. *Lehrbuch der Software–Technik: Software–Entwicklung*. Akademischer Verlag, Berlin, Heidelberg, 2000.

[BD07] Jürgen Bortz and Nicola Döring. *Forschungsmethoden und Evaluation für Human-und Sozialwissenschaftler*. Springer, 2007.

[Bed15] James A Bednar. Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components. *Python in Neuroscience*, page 104, 2015.

[BG05] Romain Brette and Wulfram Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of neurophysiology*, 94(5):3637–3642, 2005.

[BH03] Jan Benda and Andreas V. M. Herz. A universal model for spike-frequency adaptation. *Neural Comput*, 15:2523–2564, 2003.

[Blo08] Joshua Bloch. *Effective Java*. Pearson Education India, 2008.

[BMP+16] Vincent Bertram, Peter Manhart, Dimitri Plotnikov, Bernhard Rumpe, Christoph Schulze, and Michael von Wenckstern. Infrastructure to Use OCL for Runtime Structural Compatibility Checks of Simulink Models. In *Modellierung 2016 Conference*, volume 254, pages 109–116. Bonner Köllen Verlag, 2016.

[Boo48] George Boole. The calculus of logic. *Cambridge and Dublin Mathematical Journal*, 1848.

[BPT94] R Michael Bagby, James DA Parker, and Graeme J Taylor. The twenty-item Toronto Alexithymia Scale—I. Item selection and cross-validation of the factor structure. Elsevier, 1994.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 2007.

[Bra84] Wilfried Brauer. *Automatentheorie: eine Einführung in die Technik endlicher Automaten.* Teubner, 1984.

[BSW07] K. Buser, T. Schneller, and K. Wildgrube. *Medizinische Psychologie, medizinische Soziologie: Kurzlehrbuch zum Gegenstandskatalog.* Elsevier, Urban & Fischer, 2007.

[Bud04] Frank Budinsky. *Eclipse modeling framework: a developer's guide.* Addison-Wesley Professional, 2004.

[Bur06] Anthony N Burkitt. A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input. *Biological cybernetics*, 95(1):1–19, 2006.

[CBB+12] Sharon Crook, James Bednar, Sandra Berger, Robert Cannon, Andrew Davison, Mikael Djurfeldt, Jochen Eppler, Birgit Kriener, Steven Furber, Bruce Graham, Hans Ekkehard Plesser, Lars Schwabe, Leslie Smith, Volker Steuber, and Sacha van Albada. Creating, Documenting and Sharing Network Models. *Network: Computation in Neural Systems*, 23:131–149, 2012.

[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.

[CGC+15] Sharonand Crook, Padraig Gleeson, Robert Cannon, Michael Vella, and Angus Silver. Neuroml. In *Encyclopedia of Computational Neuroscience*, pages 1954–1956, New York, NY, 2015. Springer New York.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*. Springer, 2009.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Citeseer, 2003.

[CH06a] Nicholas T Carnevale and Michael L Hines. *The NEURON book*. Cambridge University Press, 2006.

[CH06b] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Che01] James Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 163–172. IEEE Computer Society, 2001.

[Chi17] Chicken Scheme a Practical and Portable Scheme System. `http://www.call-cc.org/`, 2017. Letzter Zugriff: 06.04.2017.

[CKS93] Patricia S. Churchland, Christof Koch, and Terrence J. Sejnowski. What is Computational Neuroscience? In *Computational Neuroscience*, pages 46–55. MIT Press, 1993.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[DA01] Peter Dayan and Laurence F Abbott. *Theoretical neuroscience*, volume 10. Cambridge, MA: MIT Press, 2001.

[Dav15] Andrew P Davison. NineML. *Encyclopedia of Computational Neuroscience*, pages 2086–2087, 2015.

[DDG+13] Andrew P. Davison, Markus Diesmann, Marc-Oliver Gewaltig, Satrajit S. Ghosh, Fernando Perez, Eilif Benjamin Muller, James A. Bednar, Bertrand Thirion, and Yaroslav O. Halchenko. Research topic: Python in Neuroscience II. *Frontiers in Neuroinformatics*, 2013.

[DK15] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 2015.

[DMW89] Rodney J Douglas, Kevan Martin, and David Whitteridge. A canonical microcircuit for neocortex. *Neural computation*, 1(4):480–488, 1989.

[Duv07] Paul M Duvall. Continuous Integration: Improving Software Quality and Reducing Risk, 2007.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2010.

[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.

[Epp10] Jochen Martin Eppler. *Architectures for communication between processes and software layers for a simulator for biological neural networks.* PhD thesis, University of Freiburg, 2010.

[eV12] Neurowissenschaftliche Gesellschaft e. V. Aufbau eines Neurons. `https://www.dasgehirn.info/entdecken/kommunikation-der-zellen/aufbau-eines-neurons-2907`, 2012. Letzter Zugriff: 06.04.2017.

[EVDSV+13] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*. Springer, 2013.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als indikator für softwarequalität: eine taxonomie. *Informatik-Spektrum*, 31(5):408–424, 2008.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. `https://martinfowler.com/articles/injection.html`, 2004. Letzter Zugriff: 06.04.2017.

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

[Fri10] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[GB08] Dan FM Goodman and Romain Brette. The Brian simulator. *Frontiers in neuroscience*, 3:26, 2008.

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

[GBY08] Michael Grant, Stephen Boyd, and Yinyu Ye. CVX: Matlab software for disciplined convex programming, 2008.

[GCC+10] Padraig Gleeson, Sharon Crook, Robert C. Cannon, Michael L. Hines, Guy O. Billings, Matteo Farinella, Thomas M. Morse, Andrew P. Davison, Subhasis Ray, Upinder S. Bhalla, Simon R. Barnes, Yoana D. Dimitrova, and R. Angus Silver. NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail. *PLoS Comput Biol*, 6(6), 06 2010.

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

[GHH+01] N. H. Goddard, M. Hucka, F. Howell, H. Cornelis, K. Shankar, and D. Beeman. Towards NeuroML: model description methods for collaborative modelling in neuroscience. In *Philosophical Transactions of the Royal Society of London. Series B*. The Royal Society, 2001.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*. Springer, 1993.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*. SciTePress, 2015.

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*. Springer, 2015.

[GKNP14] Wulfram Gerstner, Werner M Kistler, Richard Naud, and Liam Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on*

Software Language Engineering, Nashville, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Monticore: a framework for the development of textual domain specific languages. In *Companion of the 30th international conference on Software engineering*, pages 925–926. ACM, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, 2006.

[Gli00] Martin Glinz. Improving the quality of requirements with scenarios. In *Proceedings of the second world congress on software quality*, volume 9, pages 55–60. IEEE Computer Society, 2000.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMW06] E. Göbel, I.M. Mills, and A.J. Wallard. Das Internationale Einheitensystem (SI). *PTB-Mitteilungen*, 117, 2006.

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

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*. Springer, 2011.

[Gra03] Mark Grand. *Patterns in Java: a catalog of reusable design patterns illustrated with UML*. John Wiley & Sons, 2003.

[GRHLF11] Anatoli Gorchetchnikov, Ivan Raikov, Mike Hull, and Yann Le Franc. Network Interchange for Neuroscience Modeling Language (NineML) – Specification. `http://software.incf.org/software/nineml/wiki/nineml-specification/`, 2011. Letzter Zugriff: 06.04.2017.

[GS117] A NEURON Programming Tutorial. `http://web.mit.edu/neuron_v7.3/nrntuthtml/tutorial/tutD.html`, 2017. Letzter Zugriff: 06.04.2017.

[GSS07] Padraig Gleeson, Volker Steuber, and R Angus Silver. neuroConstruct: a tool for modeling networks of neurons in 3D space. *Neuron*, 54(2):219–235, 2007.

[HC97] M. L. Hines and N. T. Carnevale. The NEURON Simulation Environment. *Neural Computation*, 9(6):1179–1209, 1997.

[HC00] Michael L Hines and Nicholas T. Carnevale. Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation*, 12(5):995–1007, 2000.

[HC15] Michael Hines and Ted Carnevale. NEURON Simulation Environment. *Encyclopedia of Computational Neuroscience*, pages 2012–2017, 2015.

[HH52] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 2015.

[Hin93] Michael Hines. Neuron — a program for simulation of nerve equations. In *Neural Systems: Analysis and Modeling*, pages 127–136. Springer US, 1993.

[HJ12] Roger A Horn and Charles R Johnson. *Matrix analysis*. Cambridge university press, 2012.

[HKM+12] Moritz Helias, Susanne Kunkel, Gen Masumoto, Jun Igarashi, Jochen Martin Eppler, Shin Ishii, Tomoki Fukai, Abigail Morrison, and Markus Diesmann. Supercomputers ready for use as discovery machines for neuroscience. *Frontiers in Neuroinformatics*, 6, 2012.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HL95] Walter L Hürsch and Cristina Videira Lopes. Separation of concerns. 1995.

[HLMSN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. SciTePress, 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 *Domain-Specific Modeling Workshop (DSM'15)*. ACM, 2015.

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*. Springer, 2016.

[Hoa73] CAR Hoare. Hints on programming language design. 1973.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 2004.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*. Springer, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*. IEEE, 2015.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142. IEEE Computer Society, 1998.

[HUM02] John E Hopcroft, Jeffrey D Ullman, and Rajeev Motwani. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*, volume 2. Pearson Studium Deutschland, München, 2002.

[HV16] Brian Hahn and Daniel Valentine. *Essential MATLAB for engineers and scientists*. Academic Press, 2016.

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

[Izh04] Eugene M Izhikevich. Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5):1063–1070, 2004.

[JB06] Frédéric Jouault and Jean Bézivin. KM3: a DSL for Metamodel Specification. In *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

[JH07] Patricia Jablonski and Daqing Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 16–20. ACM, 2007.

[JLG04] Renaud Jolivet, Timothy J. Lewis, and Wulfram Gerstner. Generalized Integrate-and-Fire Models of Neuronal Activity Approximate Spike Trains of a Detailed Model to a High Degree of Accuracy. *Journal of Neurophysiology*, 2004.

[JNT75] James Julian Bennett Jack, Denis Noble, and Richard W Tsien. *Electric current flow in excitable cells*. Clarendon Press Oxford, 1975.

[JOP14] Eric Jones, Travis Oliphant, and Pearu Peterson. SciPy: open source scientific tools for Python, 2014.

[KAM92] Thomas B Kepler, LF Abbott, and Eve Marder. Reduction of conductance-based neuron models. *Biological Cybernetics*, 1992.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, pages 7–13. Helsinki School of Economics, 2009.

[KLR96] Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *International Conference on Advanced Information Systems Engineering*. Springer, 1996.

[KMB+96] Richard B Kieburtz, Laura McKinney, Jeffrey M Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society, 1996.

[KMW+17] Susanne Kunkel, Abigail Morrison, Philipp Weidel, Jochen Martin Eppler, Ankur Sinha, Wolfram Schenck, Maximilian Schmidt, Stine Brekke

Vennemo, Jakob Jordan, Alexander Peyser, Dimitri Plotnikov, Steffen Graber, Tanguy Fardet, Dennis Terhorst, Håkon Mørk, Guido Trensch, Alex Seeholzer, Rajalekshmi Deepu, Jan Hahne, Inga Blundell, Tammo Ippen, Jannis Schuecker, Hannah Bos, Sandra Diaz, Espen Hagen, Sepehr Mahmoudian, Claudia Bachmann, Mikkel Elle Lepperød, Oliver Breitwieser, Bruno Golosio, Hendrik Rothe, Hesam Setareh, Mikael Djurfeldt, Till Schumann, Alexey Shusharin, Jesús Garrido, Eilif Benjamin Muller, Arjun Rao, Juan Hernando Vieites, and Hans Ekkehard Plesser. NEST 2.12.0, 03 2017.

[KR14] Carsten Kolassa and Bernhard Rumpe. The Influence of the Generator's License on Generated Artifacts . In *Open Source Software for Model Driven Engineering Workshop (OSS4MDE'14)*. CEUR, 2014.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Number 1 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2010.

[KRR15] Carsten Kolassa, Holger Rendel, and Bernhard Rumpe. Evaluation of Variability Concepts for Simulink in the Automotive Domain. In *System Sciences Conference (HICSS'15)*. IEEE, 2015.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*. Jyväskylä University, Finland, 2006.

[KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *International Conference on Objects, Components, Models and Patterns*. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.

[KSE+14] Susanne Kunkel, Maximilian Schmidt, Jochen Martin Eppler, Hans Ekkehard Plesser, Gen Masumoto, Jun Igarashi, Shin Ishii, Tomoki Fukai,

Abigail Morrison, Markus Diesmann, and Moritz Helias. Spiking network simulation code for petascale computers. *Frontiers in Neuroinformatics*, 8(78), 2014.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation.* John Wiley & Sons, 2008.

[Küh06] Thomas Kühne. Matters of (meta-)modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[KV10] Lennart CL Kats and Eelco Visser. *The spoofax language workbench: rules for declarative specification of languages and IDEs.* ACM, 2010.

[Lap07] Louis Lapicque. Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation. *J. Physiol. Pathol. Gen*, 9(1):620–635, 1907.

[LC06] Frederic Lucas-Conwell. Technology evangelists: a leadership survey. In *Technology Leadership and Evangelism in the Participation Age*, 2006.

[lem17] LEMS: Low Entropy Model Specification. `http://incf.github.io/nineml/specification/`, 2017. Letzter Zugriff: 06.04.2017.

[LHB14] Ran Libeskind-Hadas and Eliot Bush. *Computing for biologists: Python programming and principles.* Cambridge University Press, 2014.

[LNPR+13] Markus Look, Antonio Navarro Pérez, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In *Globalization of Modeling Languages Workshop (GEMOC'13)*, 2013.

[Loo17] Markus Look. Unterstützung der modellgetriebenen, agilen Entwicklung von Enterprise Applikationen durch Generatoren, 2017.

[LRSS] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10).* Springer.

[Lut96] Mark Lutz. *Programming Python.* O'Reilly, 1996.

[Mac12] Ronald MacGregor. *Neural and brain modeling.* Elsevier, 2012.

[MBD+09] Eilif Muller, James A. Bednar, Markus Diesmann, Marc-Oliver Gewaltig, Michael Hines, and Andrew P. Davison. Research topic: Python in neuroscience. *Frontiers in Neuroinformatics*, 2009.

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

[Met10] MetaCase. Nokia Case Study MetaEdit+ revolutionized the way Nokia develops mobile phone software. `http://www.metacase.com/papers/metaedit_in_nokia.pdf/`, 2010. Letzter Zugriff: 06.04.2017.

[MH02] Ben McConnell and Jackie Huba. *Creatingcustomerevangelists: how loyal customers become a volunteer sales force.* Dearborn Trade Publishing, 2002.

[MMG+05] Abigail Morrison, Carsten Mehring, Theo Geisel, AD Aertsen, and Markus Diesmann. Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural computation*, 2005.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[MSN17] Pedram Mir Seyed Nazari. MontiCore: Efficient Development of Composed Modeling Language Essentials, 2017.

[MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*. Bonner Köllen Verlag, 2016.

[MSP+16] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, Thilina Rathnayake, et al. SymPy: Symbolic computing in Python. *PeerJ Preprints*, 2016.

[MSPD07] Abigail Morrison, Sirko Straube, Hans Ekkehard Plesser, and Markus Diesmann. Exact subthreshold integration with continuous spike times in discrete-time neural network simulations. *Neural computation*, 2007.

[Nin17a] lib9ML is a Python library for reading, writing, validating and manipulating models defined in the NineML language. `https://github.com/INCF/lib9ML`, 2017. Letzter Zugriff: 06.04.2017.

[nin17b] LEMS: Low Entropy Model Specification. `http://incf.github.io/nineml/`, 2017. Letzter Zugriff: 06.04.2017.

[NMWF01] John G Nicholls, A Robert Martin, Bruce G Wallace, and Paul A Fuchs. *From neuron to brain*, volume 271. Sinauer Associates Sunderland, MA, 2001.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par09] Terence Parr. *Language implementation patterns: create your own domain-specific and general programming languages*. Pragmatic Bookshelf, 2009.

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[Par17] Terence Parr. Antlr (another tool for language recognition): Homepage. `http://www.antlr.org/`, 2017. Letzter Zugriff: 06.04.2017.

[PBE+15] Dimitri Plotnikov, Inga Blundell, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML Community Workshop. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/SimLab/slns/news_events/2015/nestml-ws/Overview/_node.html/`, 2015. Letzter Zugriff: 06.04.2017.

[PBE+16] Dimitri Plotnikov, Inga Blundell, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. Code Generation from Model Description Languages II. `https://indico-jsc.fz-juelich.de/event/25/`, 2016. Letzter Zugriff: 06.04.2017.

[PBE+17a] Dimitri Plotnikov, Inga Blundell, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. Nestml homepage, 2017. Letzter Zugriff: 06.04.2017.

[PBE+17b] Dimitri Plotnikov, Inga Blundell, Jochen Martin Eppler, Philip Traeder, Bernhard Rumpe, and Abigail Morrison. NESTML Models Repository. `https://github.com/nest/nestml/tree/master/models`, 2017. Letzter Zugriff: 06.04.2017.

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*. Bonner Köllen Verlag, 2016.

[PE88] F. Pfenning and C. Elliott. Higher-order Abstract Syntax. *SIGPLAN Not.*, 1988.

[Pet12] Nico Petry. *Experimentelle Untersuchung aeroakustischer und aeroelastischer Phänomene in Hochdruck-Radialverdichtern.* PhD thesis, Universität Duisburg-Essen, Fakultät für Ingenieurwissenschaften» Maschinenbau und Verfahrenstechnik» Institut für Energie-und Umweltverfahrenstechnik, 2012.

[PL95] Zhuo-Hua Pan and Stuart A Lipton. Multiple GABA receptor subtypes mediate inhibition of calcium influx at rat retinal bipolar cell terminals. *The Journal of neuroscience*, 1995.

[POB00] Richard F. Paige, Jonathan S. Ostroff, and Phillip J Brooke. Principles for modeling language design. *Information and Software Technology*, 42, 2000.

[PPE+17] Dimitri Plotnikov, Konstantin Perun, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML2LEMS Codegenerator Project. `https://github.com/kosti1992/nestml`, 2017. Letzter Zugriff: 06.04.2017.

[Pre85] Adobe Press. *PostScript language reference manual.* Addison-Wesley Longman Publishing Co., 1985.

[Ral64] Wilfrid Rall. Theoretical significance of dendritic trees for neuronal input-output relations. *Neural theory and modeling*, pages 73–97, 1964.

[RCC+11] Ivan Raikov, Robert Cannon, Robert Clewley, Hugo Cornelis, Andrew Davison, Erik De Schutter, Mikael Djurfeldt, Padraig Gleeson, Anatoli Gorchetchnikov, Hans Ekkehard Plesser, Sean Hill, Mike Hines, Birgit Kriener, Yann Le Franc, Chung-Chan Lo, Abigail Morrison, Eilif Muller, Subhasis Ray, Lars Schwabe, and Botond Szatmary. NineML: the network interchange for neuroscience modeling language. *BMC Neuroscience*, 12, 2011.

[RD99] Stefan Rotter and Markus Diesmann. Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biological cybernetics*, 81(5-6):381–402, 1999.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the.* Pearson Higher Education, 2004.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, 2014.

[RRW16] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Model-Based Specification of Component Behavior with Controlled Underspecification. In *Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16)*. fortiss, An-Institut TU München, Technical Report, 2016.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*. Idea Group Publishing, 2002.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML*. Xpert.press. Springer Berlin, 2nd edition edition, 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, 2016.

[SB14] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 2014.

[Sch99] Axel Schmolitzky. *Ein Modell zur Trennung von Vererbung und Typabstraktion in objektorientierten Sprachen*. Shaker Verlag GmbH, 1999.

[Sch00] Wolfgang Schwerin. Models of Systems, Work Products, and Notations. In *Proceedings of Intl. Workshop on Model Engineering ECOOP, Cannes France*. Citeseer, 2000.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Number 11 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2012.

[SEHV12] Thomas Stahl, Sven Efftinge, Arno Haase, and Markus Völter. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* dpunkt Verlag, 2012.

[SGB15] Marcel Stimberg, Dan FM Goodman, and Romain Brette. Multicompartmental modeling in Brian 2. *BMC Neuroscience*, 16, 2015.

[She91] Gordon M. Shepherd. *Foundations of the Neuron Doctrine.* Oxford Univ. Press, 1991.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In *Advances in Object-Oriented Information Systems*, pages 298–311. Springer, 2002.

[Sta73] Herbert Stachowiak. Allgemeine Modelltheorie. *Plädoyer für die Vernunft. Signale einer Tendenzwende, München*, 1973.

[Str86] Bjarne Stroustrup. An overview of C++. In *ACM Sigplan Notices*, volume 21, pages 7–18. ACM, 1986.

[Str13] Bjarne Stroustrup. *The C++ programming language.* Pearson Education, 2013.

[Tay95] Barry Taylor. *Guide for the Use of the International System of Units (SI): The Metric System.* DIANE Publishing, 1995.

[TS90] Henry C Tuckwell and Idan Segev. Introduction to Theoretical Neurobiology: Vol. 1: Linear Cable Theory And Dendrite Structure; Vol. 2: Nonlinear and Stochastic Theories. *Physics Today*, 43:119, 1990.

[VCC+14] Michael Vella, Robert C. Cannon, Sharon Crook, Andrew P. Davison, Gautham Ganapathy, Hugh P. C. Robinson, R. Angus Silver, and Padraig Gleeson. libNeuroML and PyLEMS: using Python to combine imperative and declarative modelling approaches in computational neuroscience. *Frontiers in Neuroinformatics*, 8(38), 2014.

[VDK98] Arie Van Deursen and Paul Klint. Little languages: Little maintenance? *Journal of software maintenance*, 10(2):75–92, 1998.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[VDWCV11] Stefan Van Der Walt, Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[Vli98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.

[Vli04] John Matthew Vlissides. Extensible and efficient double dispatch in single-dispatch object-oriented programming languages, 2004.

[Voe14] Markus Voelter. Language Workbench Challenge. `http://www.languageworkbenches.net/`, 2014. Letzter Zugriff: 06.04.2017.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2011.

[VS10] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering, SLE*, 16, 2010.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2012.

[WH91] Ying Wang and N. J. Herron. Nanometer-sized semiconductor clusters: materials synthesis, quantum size effects, and photophysical properties. *The Journal of Physical Chemistry*, 95(2):525–532, 1991.

[Wil03] David Wile. Lessons learned from real DSL experiments. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*. IEEE, 2003.

[Wir74] Niklaus Wirth. On the Design of Programming Languages. In *IFIP Congress*, volume 74, pages 386–393, 1974.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Shaker Verlag, 2016.

[YBP+04] François Yergeau, Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0. *W3C Recommendation*, 4, 2004.

Anhang A

Abkürzungsverzeichnis

AST Abstract Syntax Tree

ANN Artificial Neural Networks

DSL Domain Specific Language

EBNF Extended Backus–Naur Form

GPL General Purpose Lanuage

GSL GNU Scientific Library

HOC Higher Order Calculator

LEMS Low Entropy Model Specification

NineML Network Interchange for Neuroscience Modeling Language

M2T Model-to-Text

M2M Model-to-Model

NESTML Nest Modeling Language

PSA postsynaptische Antwort

PNS periphere Nervensystem

SLI Simulation Language Interpreter

ZNS zentrale Nervensystem

Anhang B

Tags

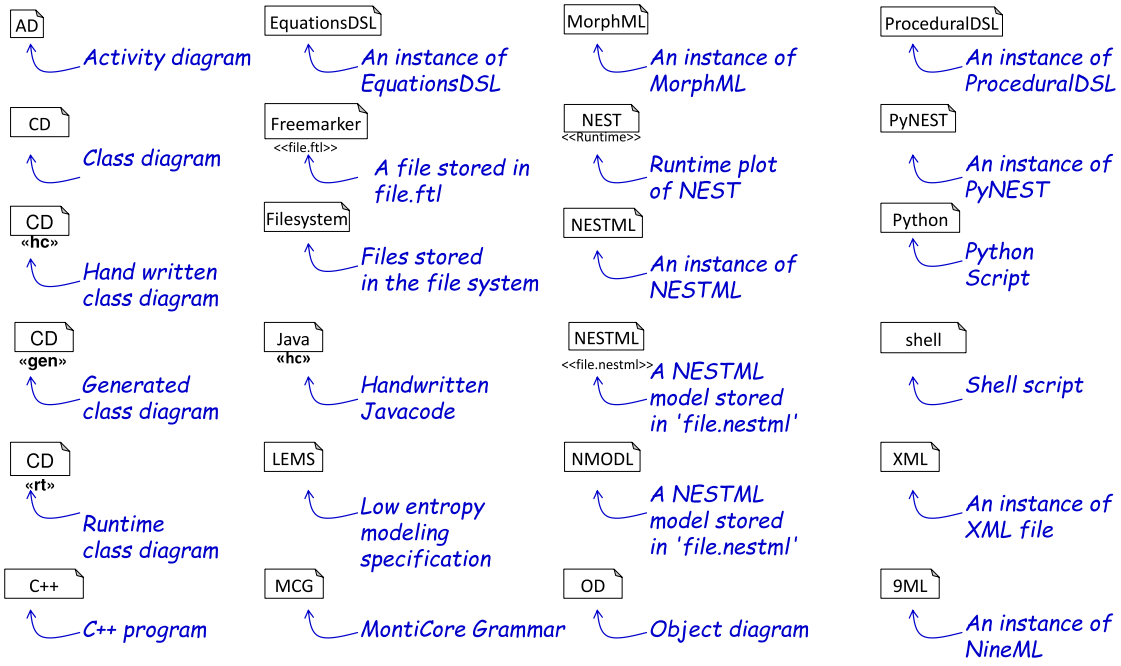


Abbildung B.1: Diese Abbildung fasst alle verwendeten Tags zusammen.

Anhang C

Tutorialevaluierung

C.1 Fragebogen

- **Q1:** What is your academic degree?
- **Q2:** How much experience do you have with NEST? (None, Little, Medium, A lot)
- **Q3:** How much programming experience do you have in general? (None, Little, Medium,A lot)
- **Q4:** The syntax of a NESTML language is clear. (0 strongly **disagree** - 5 strongly **agree**)
- **Q5:** It is easy to understand/learn the functionalities of NESTML. (0 strongly **disagree** - 5 strongly **agree**)
- **Q6:** Writing NESTML model is easier than writing a NEST C++ code. (0 strongly **disagree** - 5 strongly **agree**)
- **Q7:** Expressing mathematical statements in NESTML is easier than expressing them in NEST C++ code. (0 strongly **disagree** - 5 strongly **agree**)
- **Q8:** The error messages that are produced during the model analysis are helpful (0 strongly **disagree** - 5 strongly **agree**).
- **Q9:** This NESTML tutorial was useful for you. (0 strongly **disagree** - 5 strongly **agree**)

C.2 Praktische Aufgabe

- State equations

$$v' = 0.04 * v * v + 5 * v + 140 - u + I$$
$$u' = a * (b * v - u)$$

- State dynamics:

$$\texttt{if } v \geq 30mV \texttt{ then } \begin{cases} v = c \\ u = u + d \end{cases}$$

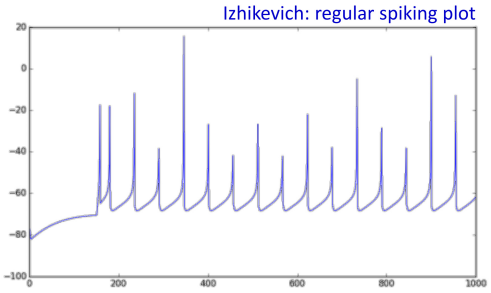


Abbildung C.1: Mathematische Spezifikation des Izhikevich-Neuorns.

Anhang D

Grammatiken von NESTML

D.1 Units-Grammatik

grammars/Units.mc4

```
/*
 * Copyright (c) 2015 RWTH Aachen. All rights reserved.
 *
 * http://www.se-rwth.de/
 */
package org.nest;

/**
  Grammar representing ODE expressions.
*/
grammar Units extends de.monticore.types.Types {

  /**
    ASTDatatype. Represents predefined datatypes and gives a possibility to use an unit
    datatype.
    @attribute boolean getters for integer, real, ...
    @attribute unitType a SI datatype
  */
  Datatype = ["integer"]
           | ["real"]
           | ["string"]
           | ["boolean"]
           | ["void"]
           | UnitType;
  /**
    ASTUnitType. Represents an unit datatype. It can be a plain datatype as 'mV' or a
    complex data type as 'mV/s'
  */
  UnitType = leftParentheses:"(" UnitType rightParentheses:")"
           | base:UnitType pow:["**"] exponent:IntLiteral
           | left:UnitType (timesOp:["*"] | divOp:["/"]) right:UnitType
           | unit:Name;

}
```

D.2 Commons-Grammatik

grammars/Commons.mc4

```
/*
 * Copyright (c) 2015 RWTH Aachen. All rights reserved.
 *
 * http://www.se-rwth.de/
 */
package org.nest;

grammar Commons extends de.monticore.types.Types, org.nest.Units {

  token SL_COMMENT =
    "#" (~('\n' |
      '\r' )
    )* :
    { _channel = HIDDEN;
      if (getCompiler() != null) {
        de.monticore.ast.Comment _comment = new de.monticore.ast.Comment(getText());
        de.se_rwth.commons.SourcePosition startPos =
        new de.se_rwth.commons.SourcePosition(_tokenStartLine, _tokenStartCharPositionInLine);
        _comment.set_SourcePositionStart(startPos);
        _comment.set_SourcePositionEnd(getCompiler().computeEndPosition(startPos, getText()));
        getCompiler().addComment(_comment);
      }
    };

  token NEWLINE = ('\r' '\n' | '\r' | '\n' );

  token WS = (' ' | '\t'):{ _channel = HIDDEN;};

  token LINE_ESCAPE = '\\' '\r'? '\n':{_channel = HIDDEN;};

  BLOCK_OPEN = ":";

  BLOCK_CLOSE = "end";

  Expr = leftParentheses:["("] Expr rightParentheses:[")"]
         | <rightassoc> base:Expr pow:["**"] exponent:Expr
         | (unaryPlus:["+"] | unaryMinus:["-"] | unaryTilde:["~"]) term:Expr
         | left:Expr (timesOp:["*"] | divOp:["/"] | moduloOp:["%"]) right:Expr
         | left:Expr (plusOp:["+"] | minusOp:["-"]) right:Expr
         | left:Expr (shiftLeft:["<<"] | shiftRight:[">>"]) right:Expr
         | left:Expr bitAnd:["&"] right:Expr
         | left:Expr bitXor:["^"] right:Expr
         | left:Expr bitOr:["|"] right:Expr
         | left:Expr (lt:["<"] |
                     le:["<="] |
                     eq:["=="] |
                     ne:["!="] |
                     ne2:["<>"] |
                     ge:[">="] |
                     gt:[">"]) right:Expr
         | logicalNot:["not"] Expr
         | left:Expr logicalAnd:["and"] right:Expr
         | left:Expr logicalOr:["or"] right:Expr
```

```
54          | condition:Expr "?" ifTrue:Expr ":" ifNot:Expr
55          | FunctionCall
56          | BooleanLiteral // true & false;
57          | NESTMLNumericLiteral
58          | StringLiteral
59          | ["inf"]
60          | Variable;
61
62    NESTMLNumericLiteral = NumericLiteral ("["type:UnitType"]"|plainType:Name)?;
63
64
65    /**
66      ASTVariable Provides a 'marker' AST node to identify variables used in expressions.
67      @attribute name
68    */
69    Variable = name:QualifiedName (differentialOrder:"\'")*;
70
71    /**
72      ASTFunctionCall Represents a function call, e.g. myFun("a", "b").
73      @attribute name The (qualified) name of the fucntions
74      @attribute args Comma separated list of expressions representing parameters.
75    */
76    FunctionCall = name:QualifiedName "(" args:(Expr& || ",")* ")";
77
78  }
```

D.3 Equations-Grammatik

grammars/ODE.mc4

```
/*
 * Copyright (c) 2015 RWTH Aachen. All rights reserved.
 *
 * http://www.se-rwth.de/
 */
package org.nest;

/**
  Grammar representing ODE expressions.
*/
grammar ODE extends org.nest.Commons, org.nest.Units {
  /** ASTOdeDeclaration. Represents a block of equations and
      differential equations.

    @attribute Equation      List with equations, e.g. "I = exp(t)" od
                             differential equations.
  */
  OdeDeclaration = (Equation | Shape | ODEAlias | NEWLINE)+;

  ODEAlias = ([record:"record"] | [suppress:"suppress"])?
             variableName:Name Datatype "=" Expr (";")?;

  /** ASTeq Represents an equation, e.g. "I = exp(t)" or epresents an
             differential equations, e.g. "V_m' = V_m+1"..
    @attribute lhs       Left hand side, e.g. a Variable.
    @attribute rhs       Expression defining the right hand side.
  */
  Equation = lhs:Derivative "=" rhs:Expr (";")?;

  Derivative = name:QualifiedName (differentialOrder:"'")*;

  Shape = "shape" lhs:Variable "=" rhs:Expr (";")?;

}
```

D.4 Procedural-Grammatik

grammars/SPL.mc4

```
/*
 * Copyright (c) 2015 RWTH Aachen. All rights reserved.
 *
 * http://www.se-rwth.de/
 */
package org.nest;

grammar SPL extends org.nest.Commons {

  SPLFile = ModuleDefinitionStatement Block;

  ModuleDefinitionStatement = "module" moduleName:QualifiedName;

  Block = ( Stmt | NEWLINE )*;

  Stmt = Small_Stmt | Compound_Stmt;

  Compound_Stmt = IF_Stmt
                | FOR_Stmt
                | WHILE_Stmt;

  Small_Stmt = Assignment
             | FunctionCall
             | Declaration
             | ReturnStmt;

  Assignment = lhsVarialbe:Variable
    (assignment:["="]      |
     compoundSum:["+="]    |
     compoundMinus:["-="]  |
     compoundProduct:["*="] |
     compoundQuotient:["/="]) Expr;


  /** ASTDeclaration A variable declaration. It can be a simple
      declaration defining one or multiple variables:
   'a,b,c real = 0'. Or an alias declaration 'alias a = b + c'.
    @attribute vars            List with variables
    @attribute Datatype        Obligatory data type, e.g. 'real' or 'mV/s'
    @attribute sizeParameter An optional array parameter. E.g. 'tau_syn ms[n_receptros]'
    @attribute expr An optional initial expression, e.g. 'a real = 10+10'
   */
  Declaration =
    vars:Name ("," vars:Name)*
    Datatype
    ("[" sizeParameter:Name "]")?
    ( "=" Expr)? SL_COMMENT?;

  /** ASTDefiningVariable Signed variable name

    @attribute minus An optional sing
    @attribute definingVariable Name of the variable
   */
```

```
54    DefiningVariable = (minus:["−"])? definingVariable:Name;
55
56    /** ATReturnStmt Models the return statement in a function.
57
58      @attribute minus An optional sing
59      @attribute definingVariable  Name of the variable
60     */
61    ReturnStmt = "return" Expr?;
62
63    IF_Stmt = IF_Clause
64              ELIF_Clause*
65              (ELSE_Clause)?
66              BLOCK_CLOSE;
67
68    IF_Clause = "if" Expr BLOCK_OPEN Block;
69
70    ELIF_Clause = "elif" Expr BLOCK_OPEN Block;
71
72    ELSE_Clause = "else" BLOCK_OPEN Block;
73
74    FOR_Stmt = "for" var:Name "in" from:Expr "..." to:Expr
75               ("step" step:SignedNumericLiteral)?
76               BLOCK_OPEN Block BLOCK_CLOSE;
77
78    WHILE_Stmt = "while" Expr BLOCK_OPEN Block BLOCK_CLOSE;
79
80  }
```

D.5 NESTML-Grammatik

grammars/NESTML.mc4

```
/*
 * Copyright (c) 2015 RWTH Aachen. All rights reserved.
 *
 * http://www.se-rwth.de/
 */
package org.nest;

/**
  Grammar representing the Simple Programming Language (SPL).
  It is easy to learn imperative language which leans on the Python syntax.
*/
grammar NESTML extends org.nest.SPL, org.nest.ODE {

  /** ASTNESTMLCompilationUnit represents the complete entire file with neuron
      and component models.
    @attribute packageName     The qualified name to artifact
    @attribute Import          List of imported elements
    @attribute Neuron          The neuron representation
    @attribute Component       The component representation
  */
  NESTMLCompilationUnit =
    (Import | NEWLINE)*
    (Neuron | Component | NEWLINE)*
    EOF;

  /** ASTImport represents the import line. Can be the qualified name oder a
      wirldcard import.
    @attribute qualifiedName   The qualified name to artifact
    @attribute star            Optional wildcard ('*')
  */
  Import = "import" QualifiedName ([star:".*"])?  (";")?;

  /** ASTNeuron represents neuron.
    @attribute Name   The name of the neuron
    @attribute Body   The body of the neuron, e.g. internal, state, parameter...
  */
  Neuron = "neuron" Name ("extends" base:Name)? Body;

  /** ASTComponent represents neuron.
    @attribute Name   The name of the component
    @attribute Body   The body of the component, e.g. internal, state, parameter...
  */
  Component = "component" Name Body;

  /** ASTBodyElement represents a single entry in the neuron or component:
      e.g. internal, state, parameter... The interface is used to enable
      language extension.
  */
  interface BodyElement;

  /** ASTBody The body of the neuron, e.g. internal, state, parameter...
  */
  Body = BLOCK_OPEN
```

```
54            (NEWLINE | BodyElement)*
55          BLOCK_CLOSE;
56
57  /** ASTUSE_Stmt represent a reference to an another neuron or component. E.g.:
58    neuron AliasNeuron:
59              use nest.EmptyComponent as Soma
60    end
61
62    @attribute Name     The name of the referenced component
63    @attribute alias    The name under which the referenced component can be used.
64  */
65  USE_Stmt implements BodyElement = "use" name:QualifiedName "as" alias:Name;
66
67  /** ASTVar_Block represent a block with variables, e.g.:
68    state:
69      y0, y1, y2, y3 mV [y1 > 0; y2 > 0]
70    end
71
72    @attribute state true if the varblock ist a state.
73    @attribute parameter true if the varblock ist a parameter.
74    @attribute internal true if the varblock ist a state internal.
75    @attribute AliasDecl a list with variable declarations.
76  */
77  Var_Block implements BodyElement =
78    (["state"]|["parameter"]|["internal"])
79    BLOCK_OPEN
80      (AliasDecl | NEWLINE)*
81    BLOCK_CLOSE;
82
83  /** ASTDynamics a special function definition:
84      update:
85        if r == 0: # not refractory
86          integrate(V)
87        end
88      end
89    @attribute block Implementation of the dynamics.
90   */
91  Dynamics implements BodyElement =
92    "update"
93    BLOCK_OPEN
94      Block
95    BLOCK_CLOSE;
96
97  /** ASTEquations a special function definition:
98       equations:
99         G = (e/tau_syn) * t * exp(-1/tau_syn*t)
100        V' = -1/Tau * V + 1/C_m * (I_sum(G, spikes) + I_e + currents)
101      end
102    @attribute ddeDeclaration Block with equations and differential equations.
103   */
104 Equations implements BodyElement =
105   "equations"
106   BLOCK_OPEN
107     OdeDeclaration
108   BLOCK_CLOSE;
109
110
111 /** ASTVar_Block represents a block with variables, e.g.:
```

```
112
113     @attribute hide is true iff. declaration is not trackable.
114     @attribute alias is true iff. declaration is an alias.
115     @attribute declaration embeds the SPL variable declaration.
116     @attribute invariants List with optional invariants.
117   */
118   AliasDecl =
119     ([record:"record"] | [suppress:"suppress"])? (["alias"])?
120     Declaration
121     ("[["  invariant:Expr "]]")?;
122
123   /** ASTInput represents the input block:
124     input:
125       spikeBuffer    <- inhibitory excitatory spike
126       currentBuffer <- current
127     end
128
129     @attribute inputLine set of input lines.
130   */
131   Input implements BodyElement = "input"
132     BLOCK_OPEN
133       (InputLine | NEWLINE)*
134     BLOCK_CLOSE;
135
136   /** ASTInputLine represents a single line form the input, e.g.:
137       spikeBuffer    <- inhibitory excitatory spike
138
139     @attribute sizeParameter Optional parameter representing multisynapse neuron.
140     @attribute sizeParameter Type of the inputchannel: e.g.
141                   inhibitory or excitatory (or both).
142     @attribute spike true iff the neuron is a spike.
143     @attribute current true iff. the neuron is a current.
144   */
145   InputLine =
146     Name
147     ("[" sizeParameter:Name "]")?
148     "<-" InputType*
149     (["spike"] | ["current"]);
150
151   /** ASTInputType represents the type of the inputline e.g.: inhibitory or excitatory:
152     @attribute inhibitory true iff the neuron is a inhibitory.
153     @attribute excitatory true iff. the neuron is a excitatory.
154   */
155   InputType = (["inhibitory"] | ["excitatory"]);
156
157   /** ASTOutput represents the output block of the neuron:
158         output: spike
159       @attribute spike true iff the neuron has a spike output.
160       @attribute current true iff. the neuron is a current output.
161     */
162   Output implements BodyElement = "output" BLOCK_OPEN (["spike"] | ["current"]) ;
163
164   /** ASTFunction a function definition:
165       function set_V_m(v mV):
166         y3 = v - E_L
167       end
168     @attribute name Functionname.
169     @attribute parameters List with function parameters.
```

```
170     @attribute returnType Complex return type, e.g. String
171     @attribute primitiveType Primitive return type, e.g. int
172     @attribute block Implementation of the function.
173   */
174   Function implements BodyElement = "function" Name "(" Parameters? ")"
175             (returnType:Datatype)?
176             BLOCK_OPEN
177               Block
178             BLOCK_CLOSE;
179
180   /** ASTParameters models parameter list in function declaration.
181     @attribute parameters List with parameters.
182   */
183   Parameters = Parameter ("," Parameter)*;
184
185   /** ASTParameter represents singe:
186       output: spike
187     @attribute compartments Lists with compartments.
188   */
189   Parameter = Name Datatype;
190 }
```

Anhang E

Modelle für Tutorial

E.1 rc_neuron

tutorial_models/11.rc_neuron.nestml

```
/*
  A straight forward implementation of the RC circuit approach
*/
neuron rc_neuron:

  state:
    V_abs mV = 0mV
  end

  equations:
      V_abs' = -1/tau_m * V_abs + 1/C_m*I_syn
  end

  parameter:
    E_L   mV = -65mV
    C_m   pF = 250pF
    tau_m ms = 10ms
    I_syn pA = 10pA
  end

  input:
    spikes    <- spike
  end

  output: spike

  update:
    integrate_odes()
  end

end
```

E.2 alpha-shaped postsynaptic responce

tutorial_models/32_rc_alpha.nestml

```
/*
  RC circuit approach and alpha shapes for synaptic currents.
*/
neuron rc_alpha:
  state:
    V_m mV = E_L
    g_ex pA = 0pA
  end

  equations:
    g_ex'' = -g_ex' / tau_syn
    g_ex' = g_ex' - ( g_ex / tau_syn )
    I_syn pA = g_ex + currents + I_e
    V_m' = -(V_m - E_L)/tau_m + I_syn/C_m
  end

  parameter:
    E_L   mV = -70mV
    C_m   pF = 250pF
    tau_m ms = 10ms
    tau_syn ms = 2.0ms
    alias V_th mV = -55mV - E_L
    alias V_reset mV = -70mV - E_L
    refractory_timeout ms = 2ms
    refractory_counts integer = 0
    I_e pA = 0pA
  end

  internal:
    PSConInit_E pA/ms = 1.0 pA * e / tau_syn
  end

  input:
    spikes  <- spike
    currents <- current
  end

  output: spike

  update:
    if refractory_counts == 0:
      integrate_odes()
      if V_m > V_th:
        V_m = V_reset
        emit_spike()
        refractory_counts = steps(refractory_timeout)
      end
    else:
      refractory_counts -= 1
    end
    g_ex' += PSConInit_E * spikes
  end
end
```

E.3 shapes

tutorial_models/33_rc_shape.nestml

```
/*
  A straight forward implementation of the RC circuit approach.
  Adds alpha shapes for synaptic currents which is modelled as an excplicit function.
*/
neuron rc_shape:

  state:
    V_m mV = E_L
  end

  equations:
    shape g_ex = (e/tau_syn) * t * exp(-1/tau_syn*t)
    I_syn pA = I_sum(g_ex, spikes) + currents + I_e
    V_m' = -(V_m - E_L)/tau_m + I_syn/C_m
  end

  parameter:
    E_L   mV = -70mV
    C_m   pF = 250pF
    tau_m ms = 10ms
    tau_syn ms = 2.0ms

    alias V_th mV = -55mV - E_L
    alias V_reset mV = -70mV - E_L
    refractory_timeout ms = 2ms
    refractory_counts integer = 0
    I_e pA = 0pA
  end

  internal:
    PSConInit_E real = 1.0 * e / tau_syn
  end

  input:
    spikes    <- spike
    currents <- current
  end

  output: spike

  update:
    if refractory_counts == 0:
      integrate_odes()
      if V_m > V_th:
        V_m = V_reset
        emit_spike()
        refractory_counts = steps(refractory_timeout)
      end
    else:
      refractory_counts -= 1
    end
  end
end
```

E.4 Izhikevich neuron

tutorial_models/izhikevich.nestml

```
/*
  Solution to the tutorial exercise. It is a reference
  implementation of the izhikevich_neuron.
*/
neuron izhikevich_neuron:
  state:
    V_m mV = -65mV # Membrane potential in mV
    U_m real
    # TODO add new variable U_m with the type real
    # NESTML syntax for variables: variable_name real = initial_value
  end

  equations:
    # TODO Add 2 ODEs for the V_m und U_m.
    # You can use current buffer I directly in the ODE
     V_m' = 0.04 *V_m *V_m + 5 * V_m + 140 - U_m + I
     U_m' = a*(b*V_m -U_m)
  end

  parameter:
    # Add 4 variables a,b,c, d of real type
    a real = 0.02
    b real = 0.2
    c mV = -65.0mV
    d real = 2mV
  end

  input:
    # TODO add current buffer named I
    # NESTML Syntax for current buffers: buffer_name <- current
    I <- current
  end

  output: spike

  update:
    integrate_odes()
    # TODO: Implement threshold crossing check

    # use an if-conditional. The NESTML systax looks like:
    # if a >= b:
    #   a += b
    #   b = a
    #   emit_spike()
    # end
    # threshold crossing
    if V_m >= 30.0mV:
      V_m = c
      U_m += d
    end
  end
end
```

Anhang F

Curriculum Vitae

Name	Plotnikov
Vorname	Dimitri
Geburtstag	15.04.1985
Geburtsort	Simferopol, Ukraine
Staatsangehörigkeit	deutsch
seit 2012	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Software Engineering RWTH Aachen
2012	Master of Science in Informatik
2007 - 2012	Studium der Informatik an der RWTH Aachen
2007	Abitur
2005 - 2007	Cusanus Gymnasium Erkelenz
2005	Fachabitur
2002 - 2005	Städtisches Gymnasium Neuwerk, Mönchengladbach
2002	Mittlere Schulreife
1992 - 2002	Staatliche mittlere Schule, Simpferopol/Ukraine

Tabellenverzeichnis

Abbildungsverzeichnis

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and

extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, **?**] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise

technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems

are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split

into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbour-

hoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering.* Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing.* Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton.* Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage.* Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage.* Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods.* Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.