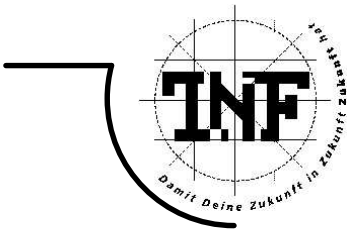
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



A New Approach to a Fast Simulation of Spiking Neural Networks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

DIPLOMINGENIEUR

in der Studienrichtung

INFORMATIK

Angefertigt am *Institut für Systemtheorie*

Betreuung:

o. Univ.-Prof. Ing. Dr. Franz Pichler

Eingereicht von:

Rene Mayrhofer

Mitbetreuung:

Dipl.-Ing. Dr. Michael Affenzeller

Linz, Juli 2002

Johannes Kepler Universität
A-4040 Linz · Altenbergerstraße 69 · Internet: http://www.uni-linz.ac.at · DVR 0093696

Kurzfassung

Spikende Neuronale Netzwerke werden aufgrund ihrer verbesserten Flexibilität und erhöhten Anzahl von Freiheitsgraden gerne als ein neues Berechnungs-Paradigma angesehen – sie stellen den direkten Nachfolger der Künstlichen Neuronalen Netzwerke dar. Obwohl die Eigenschaften dieses neuen Typs Neuronaler Netzwerke derzeit nur in begrenztem Maße bekannt sind, ist er dennoch eindeutig leistungsfähiger als sein Vorgänger; außer der möglichen Simulation Künstlicher Neuronaler Netzwerke in Echtzeit können neue, zuvor unbekannte Berechnungselemente in der Modellierung verwendet werden. Allerdings erfordern aktuelle Implementierung zur Simulation Spikender Neuronaler Netzwerke bisher den Einsatz kontinuierlicher Simulationstechniken, durch die Skalierbarkeit auf große Netzwerke mit vielen Neuronen erschweren.

Diese Diplomarbeit führt ein neues Modell für Spikende Neuronale Netzwerke ein, welches die Anwendung von schneller, diskreter ereignisbasierter Simulation erlaubt; dadurch entstehen möglicherweise enorme Vorteile in Flexibilität und Skalierbarkeit, ohne die qualitative Berechnungsleistung zu mindern. Das neue Modell wurde außerdem in einem Plattform-unabhängigen, in Java geschriebenen Prototyp-Simulationsframework implementiert. Durch die ausschließliche Verwendung diskreter ereignisbasierter Simulation beweist das Framework die Funktionsfähigkeit des neuen Konzepts – es wurde bereits erfolgreich zur Emulation von Standardtypen Künstlicher Neuronaler Netzwerke sowie zur Simulation eines biologisch inspirierten Filter-Modells eingesetzt. Die Resultate dieser Simulationen werden in folgenden Kapiteln präsentiert und mögliche Richtungen für zukünftige Weiterentwicklungen angegeben. Zusätzlich werden einige erweiterte Techniken bezüglich des Einsatzes diskreter ereignisbasierter Simulation angegeben, um die durch das neue Konzept entstandenen Möglichkeiten nutzen zu können.

Abstract

Spiking Neural Networks are considered as a new computation paradigm, representing the next generation of Artificial Neural Networks by offering more flexibility and degrees of freedom for modeling computational elements. Although this type of Neural Networks is rather new and there exists only a vague knowledge about its features, it is clearly more powerful than its predecessor, not only being able to simulate Artificial Neural Networks in real time but also offering new computational elements that were not available previously. Unfortunately, the simulation of Spiking Neural Networks currently involves the use of continuous simulation techniques which do not scale easily to large networks with many neurons.

In this diploma thesis, a new model for Spiking Neural Networks is introduced; it allows the use of fast discrete event simulation techniques and possibly offers enormous advantages in terms of simulation flexibility and scalability without restricting the qualitative computational power. As a proof of concept, the new model has been implemented in a prototype simulation framework, written platform-independently in Java. This simulation framework utilizes solely discrete event simulation and has been successfully used to emulate typical Artificial Neural Networks and to simulate a biologically inspired filter model. The results of the conducted example simulations are presented and possible directions for future research are given. Additionally, a few advanced techniques regarding the use of discrete event simulation, which offers some new opportunities, are shortly discussed.

Acknowledgments

This diploma thesis would not have been possible without the support from numerous people. I am especially thankful to:

- Franz Pichler, my supervisor, who always provided a system theoretic view on the topic, emphasizing the interdisciplinary character and encouraging to use ideas from other fields of research.
- Michael Affenzeller for introducing me into the exciting research topic of Spiking Neural Networks in the first place and for his continuous technical, stylistic and motivational support during the almost 2 years that I have been working on this topic.
- Herbert Prähofer for providing the MOSAIC simulation framework and hints for the object oriented implementation of the formal model.
- Gerhard Höfer for his help with validating the abstractions in the new model for biological plausibility and the idea of constructing neural networks hierarchically.
- Alexander Fried for coming up with the idea of internally representing piecewise linear functions in an optimized way which eased the implementation of parts of the framework.
- All of the people who were proof-reading this work and providing me with invaluable hints for improving it.

Finally, I am grateful to my beloved family for continuous financial and emotional support, without which I would not have been able to finish my work in time.

Contents

Chapter 1

Introduction and Overview

Currently the technology of Neural Networks is undergoing a change; a new generation of Neural Networks is on the verge of becoming important for theoretical considerations as well as practical applications. At the moment, Neural Networks are successfully used for several application domains in computer science, typically for: pattern recognition such as speech recognition, speaker recognition, face recognition, etc.; pattern classification such as quality assurance, stock index prognosis, creditworthiness rating, etc. Such networks are formal computational models inspired by biological Neural Networks like the human brain. But recent biological evidence led to the conclusion that current *Artificial Neural Networks* (ANNs) may not be very well suited for fast information processing [TFM96], which is needed when trying to recognize or classify live data input. One of the reasons might be that current ANNs model only one aspect of the information transmissions which take place in natural Neural Networks, namely the firing rate of Neurons [Zel94]. However, biological experiments showed that the visual system in the human brain transmits at least some information in the exact timing of single electrical impulses, known as spikes. When averaging over those spikes for computing the firing rate, much of the encoded information will be lost. To model the temporal aspect of information transmission, *Spiking Neural Networks* (SNNs) have been developed; this mathematical model of Neural Networks explicitly models the exact timing of single spikes. Unfortunately current techniques for simulating SNNs need too much computing power to be used in practical applications where computation is in the main point of view, creating the need for better simulation techniques of this mathematical model.

At the moment, there exist a few simulation tools that can be used to model and simulate SNNs; among the most popular is GENESIS, the *General Neural Simulation System* [BB94]. But this system has been developed to simulate the basic elements known in neuro-science at a level of great detail, modeling each neuron with possibly multiple sections, named compartments, and modeling the compartment behavior with neuro-electrical equations. Thus, it is well suited to simulate the biological behavior

very closely, allowing neuro-biologists to conduct research on the level of single neurons or few interconnected neurons. However, Neural Networks in computer science applications – also in embedded systems hiding their usage – are used at another level of abstraction; these different models of SNNs can be seen as a multi-strata system [MT75] (also cf. [Pic00a]). GENESIS implements a model that lies beneath the level of topology, behavior and learning. It can be used to simulate this level, but it was not developed for this purpose and is therefore just barely qualified for it. To conduct research on the topology, behavior and learning of SNNs, which is the "computer-science and information theory level" as opposed to the "neurobiology level", more abstract models are needed. Under this circumstance, it is more important to simulate large populations of neurons together with their interactions than to simulate single neurons in detail.
In order to do so, this diploma thesis tries to utilize the technique of *discrete event simulation* [ZPK00] – by approximating all used functions with piecewise linear functions – to compute SNNs, possibly offering an enormous increase in both simulation speed and scalability. Using discrete event simulation, only those parts of an SNN that are active at a given time need to be simulated, concentrating on the important points of the simulation and thus allowing the simulation of larger and more sophisticated Neural Networks. Within the present thesis, the aim is to construct a prototype simulation framework that can be used to easily conduct simulations of SNNs by only defining the network structure. The increase in simulation flexibility, speed and scalability shall enable the use of the next generation of Neural Networks – the Spiking Neural Networks – in practical applications, offering more potential for self-learning behavior in commonly used soft- and hardware tools.

In chapter 2 the theoretical foundations on which this thesis builds upon are introduced. After summarizing the history of the development of Neural Networks and giving a motivation for research on SNNs, the most important, currently used models for SNNs and discrete event simulation are presented. Although, for the first time these models were introduced nearly 10 years ago, to the author's best knowledge, this seems to be the first attempt to unify them – to apply discrete event simulation to SNNs. A formal model allowing this application, which has already been presented in [MAP+02], is thoroughly introduced in chapter 3. One of the basic abstractions which allows efficient simulation is the usage of piecewise linear functions. To allow formal statements in the new model, a special notation for calculations with piecewise linear functions is presented, followed by the respective formulation of the functions used in the model of SNNs. Since the main task in discrete event simulation is to calculate the times when events arise, an algorithm for calculating neuron firing times in linear time complexity has been developed. This formal model is completed by the description of coding converters at the network in- and outputs and a discussion about first learning algorithms.

Since the second aim of this diploma thesis – in addition to the development of the aforementioned formal model – is the creation of a prototype simulation framework; this implementation of the model in Java code is described in chapter 4. However, the description is restricted to the structural and functional concepts, details are only shown where appropriate and necessary for an understanding of the inner concepts. A detailed description of all implemented components can be found in the form of Javadoc documentation, which is generated directly from the augmented Java source code and is accompanying this diploma thesis. Currently, the prototype simulation framework implements the handling of spike events at the neuronal and synaptic layer and enables to build networks of neurons and synapses. Furthermore, it contains visualization components to gain an overview of the inner network operations. Even though the framework currently is only in the state of being a prototype and mostly acts as a proof of concept, it already works very well for some example simulations and is well suited as the base for conducting research on discrete event simulation of SNNs. Based on the MOSAIC simulation framework and written in the Java programming language, the prototype simulation framework is completely independent of the hardware and the operating system – but currently not especially optimized with respect to running time. The developed prototype simulation framework was used to perform a few simulations of SNNs with discrete event simulation. The structure and results of these examples are presented in chapter 5; four different examples have been constructed to show the capabilities of the simulation framework. After that, a few ideas on the future development are given in chapter 6: First of all, it might be advantageous to apply system theoretical methods to SNNs by constructing them in a hierarchical way, forming components from simpler parts with clearly defined input and output behavior. This approach might help in mastering the complexity of large and powerful Neural Networks. Then, to make learning algorithms more flexible and possibly better in solving given problems, the support to create and remove simulation components during run-time is explained. Because the simulation of large SNNs in real time might not be possible on current single processor systems, a few approaches on parallelizing the discrete event simulation are summarized. Finally, a short discussion about the application of learning algorithms that use only locally available information and their relation to the biological interpretation completes this terse list of ideas for possible future research topics.
Chapter 7 then gives a summary of the whole thesis by drawing conclusions and presents a more speculative future perspective concerning the usage of SNNs.

It should be pointed out that this diploma thesis is not about neurobiology. Although some details of the biological model of Neural Networks are presented and Spiking Neural Networks are inspired by biology, the main interest stems from computer science and information technology – to show what can be done with biologically in-

spired computational models. Many of the details from biological models have been intentionally abstracted to allow a shift to a higher level, where the simulation of large populations of neurons is possible with current computing resources.

Chapter 2

Theoretical Foundations for Spiking Neural Networks

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are systems for information processing, modeled after biological Neural Networks like the human brain. All Neural Networks share a common characteristic: the use of a large number of simple, connected information processing elements (called neurons) forming a network. The strength of all Neural Networks lies in their connections, which contain the stored information and form the processing structure of the system. Another major feature of Neural Networks is their ability to learn. Although Neural Networks act as an information processing system, they typically cannot be "programmed" or "parameterized" [Zel94].
Instead of this, Neural Networks learn their behavior. This learning can be performed using either supervised methods, giving the Neural Networks input values and also the respective desired output values, or unsupervised methods where the Neural Network classifies input values on its own. During the learning process, a typical Neural Network not only stores the presented input patterns to recognize them in the future, but also accomplishes a generalization such that untrained input patterns that are close to trained ones can also be recognized. There are quite some advantages of Neural Networks in contrast to conventional algorithms (some items were taken from [Zel94]), due to which they are a widely accepted and successful computational element:

- As already mentioned, Neural Networks have the ability to learn, enabling them to compute functions for which no formal, mathematical representation is currently known. In addition, they are able to represent any function, simple or complex, linear or non-linear; they are universal approximators [Kol57].

- Additionally, they are more fault-tolerant than other algorithms because small changes in the input values normally cause no changes in the output values at

all. Neural Networks can also adapt to the failure of single or multiple Neurons, allowing the whole system to still be efficient enough when some parts fail.

- Furthermore, due to their inherently parallel nature, very high computational rates can be achieved when Neural Networks are implemented directly in hardware or simulated using parallel algorithms (see section 6.3).

- Another major advantage for some applications is that information in Neural Networks is stored associatively; i.e. it is much simpler to recall a pattern that is close to the input pattern than it would be with random memory machines.

- Due to the automatic classification and generalization of input patterns (that some Neural Network types are capable of), sensible default values are automatically chosen for incompletely specified input patterns.

But, as helpful Neural Networks are in some situations, they also have some disadvantages:

- A Neural Network typically behaves as a black box in the system theoretical sense [PS90]; when given an input value, the network produces an output value. But in the general case it will not be possible to deduce the behavior of the network from its internal parameters without completely simulating it. This makes it difficult, if not even impossible, to validate Neural Networks for their correctness in solving a given problem.

- Gaining knowledge within a Neural Network is – in many cases – impossible without using time-intensive learning algorithms. Currently there are only a few approaches for equipping Neural Networks with "instincts" (see [SO01]); these would be a kind of basic knowledge that help the Neural Networks in the early stages of their learning.

- Currently, almost all of the recently used learning algorithms (including the Backpropagation learning rule) are slow in spite of many optimizations.

Due to these reasons, Neural Networks should not be used for applications where good deterministic solutions already exist. However, there are many application domains in which either no deterministic solution is currently known (or in which no such solution is possible due to some intrinsic properties) or in which the deterministic solution is not feasible (e.g. due to running time).

There are a few motivations for theoretical research on Neural Networks. An obvious one is to achieve a deeper understanding of the behavior of biological systems by trying to reproduce some effects in simulations. Another motivation is that ANNs can learn

from given input and output values, therefore enabling the system to calculate functions for which no mathematical representation is known. A less obvious motivation for research on Neural Networks, mainly from computer science and complexity theory, lies in the fact that they are massively parallel systems that can be seen as parallel algorithms. But one of the main goals and possibly the driving force for many research projects might still be the vision of an intelligent machine; Neural Networks seem to be *the* formal model that could supposedly be able to reproduce the "intelligence" and "consciousness" of a human brain, however we choose to define these terms.
Depending on the motivation for looking at Neural Networks, different features will be significant. If the system is to be used as a simulator for studying effects in biological systems, it will be essential that the artificial neurons act like their biological counterparts as closely as possible. In contrast, if human psychology or complexity theory is the main point of view, then the properties of single neurons might not be as important as the number of neurons in the system and the connection structure (the topology) of the network. However, within this diploma thesis, the focus is not research on biological Neural Networks. Instead, the focus is on showing that the next generation of ANNs, the Spiking Neural Networks (SNNs) can offer advantages for almost all points of view, making them an effective successor of ANNs in current applications.

2.1.1 History

The history of (artificial) Neural Networks is almost as long as the history of programmable computers built of transistors. The first papers dealing with ANNs were written over 50 years ago, forming a base that is still used in current research.
This short summary of the development of ANNs is a summary of the respective chapter in [Zel94], extended by the history of SNNs.

- Early beginnings (1942 – 1955): As early as 1943 the essay "A logical calculus of the ideas immanent in nervous activity" was written by Warren McCulloch and Walter Pitts, describing a first form of Neural Networks based on the "McCulloch-Pitts" neuron. It also showed that even simple classes of Neural Networks are in principle able to calculate arbitrary arithmetic or logical functions. Although no practical application was given in this document, it had an influence on other, later famous researchers including Norbert Wiener and John von Neumann. Pitts and McCulloch wrote another article named "How we know universals" in 1947, in which they discussed the problem of recognizing spatial patterns invariant of their position.
  In 1949 Donald O. Hebb described the classical Hebbian learning rule in his book "The Organization of Behaviour". It represents a simple concept of learning for individual neurons. Hebb also used this rule for arguing results from

psychological experiments. In its universal form, this learning rule is the basis for almost all known learning methods in the context of Neural Networks. Also the concept of cell assemblies, which are laterally connected, mutually exciting subsets of neurons, traces back to Hebb.
Karl Lashley, a neuro-physiologist, stated in 1950 in his work "In search of the engram" the thesis that information in the brain must be stored in a distributed representation. He came to this conclusion by conducting experiments on rats. In these experiments, only the extent and not the position of the destruction of neural cells determined the ability to run through a labyrinth. Although today the idea of a fully distributed information storage has been discarded and we know that the brain features functionally distinguishable areas, his work had a lot of influence on the following research.

- First successes (1955 – 1969): The first successful neuro-computer "Mark I Perceptron" was built in the years 1957–1958 by Frank Rosenblatt, Charles Wightman and employees at the MIT. It was used for pattern recognition problems and was able to recognize simple numbers with a 20*20 pixel optical sensor. Although Marvin Minsky had already developed a neuro-computer with automatically adjusting weights in 1951 ("Snark", which he used in his PhD thesis in 1954), "Mark I Perceptron" had 512 motor-driven potentiometers for his variable weights. Besides this technical achievement, Frank Rosenblatt became generally known for his book "Principles of Neurodynamics", which was published in 1959. In this book he describes different variations of the perceptron and also shows a proof that a perceptron can learn every function that can be possibly represented with the network by applying his learning method.
  In 1958, Oliver Selfridge presented in his work "Pandemonium" dynamic, interactive mechanisms for solving the practical problem of Morse-Code translation using models of human information transmission and the hill climbing learning method.
  Karl Steinbruch showed in 1961 in his work "Die Lernmatrix" simple technical realizations of associative memory, the predecessor of today's neural associative memory. They were constructed as technical realizations of Pawlow's conditional reflexes. Besides a binary model there was also a model for continuous input and learning methods for both models.
  Bernard Widrow formed some time after 1960 the Memistor Corporation, the first neuro-computing company. This company produced memistors, elements like transistors but for realizing the adjustable weights of an ANN.
  In the period between 1955 and 1969, researchers thought that the basic principles of self-learning, "intelligent" systems have been discovered. This overestimation, particularly in the media, led to the following intermission in popularity

as soon as the limits of the used models and learning methods became clear.

- The silent years (1969 – 1982): In 1969 Marvin Minsky and Seymour Papert conducted a detailed mathematical analysis of the perceptron and showed that this model is unable to represent many important problems at all. Using a few very simple problems like the XOR-problem, the parity-problem and the connectivity-problem they were able to show that the pristine perceptron as well as different variants are inherently unable to solve these and related problems. They also concluded that even more powerful models than the perceptron would have the same problems and that Neural Networks would be a dead-end. Fortunately, this conclusion is not correct according to today's point of view. But at a time of stagnation in the field of Neural Networks, this statement caused that researchers who were working on Neural Networks did not receive any funding for the next 15 years.
  Although there were no real breakthroughs during this time, some famous researchers were able to build up a lot of the theoretical foundations that Artificial Neural Networks are based on today.
  E.g. Teuvo Kohonen introduced in 1972 in his work "Correlation matrix memories" a model of a linear associator (a special associative memory) that uses linear activation functions and continuous values for weights, activation values and outputs.
  In 1974 Paul Werbos developed the backpropagation learning method in this PhD thesis (which was used about 10 years later due to work by Rumelhart and McClelland).
  Stephen Grossberg published a number of papers, including a work on the problem of letting a Neural Network learn new patterns without destroying already learned ones. He was one of the first to use sigmoidal activation functions and nonlinear lateral inhibition. His models of Adaptive Resonance Theory (ART) are best known.
  John Hopfield, a well-known physicist, wrote his article "Neural Networks and physical systems with emergent collective computational abilities" in 1982, describing binary Hopfield-networks as the neural equivalent to Ising-models in physics. Two years later he enhanced his model to continuous Hopfield-networks.
  Teuvo Kohonen got known especially for his self-organizing maps (SOMs), which he described in this article "Self-organized formation of topologically correct feature maps" in 1982.

- Renaissance (1985 – today): In the early eighties the field of Neural Networks was revived. It is often cited that John Hopfield had a major influence on the

revival with his article "Neural Computation of Decisions in Optimization Problems" in 1985, in which he showed how Hopfield-networks can solve the traveling salesman problem. He also personally convinced many researchers of the importance of this research topic.
Another, maybe stronger influence was caused by the development and wide publication of the backpropagation learning method in 1986 by Rumelhart, Hinton and Williams in "Learning internal representations by error propagation" in the book "Parallel Distributed Processing" published by Rumelhart and McClelland. The method was also described in the article "Learning representations by back-propagating errors" in Nature in the same year. The backpropagation learning method is – compared to older methods – very fast and robust for learning patterns in multi-layer feed-forward networks. Another advantage is that it can be described in a mathematically elegant way as a gradient descent method.
In 1986 Terrence Sejnowski and Charles Rosenberg showed with "Nettalk: a parallel network that learns to read aloud" an impressive application that used a feed-forward network trained with backpropagation to learn the pronunciation of written single words in English. The network learned the pronunciation by itself and the whole project reached a performance level almost as good as the knowledge-based DECtalk-system (in which many man-years of development were invested) after only a few weeks of work.
Since 1986, the field of Neural Networks has been developing explosively: the number of researchers working on this topic is currently a few thousand, there are many scientific publication journals with Neural Networks as their main topic, large recognized scientific communities like INNS (International Neural Network Society), ENNS (European Neural Network Society), a large IEEE group and groups of national computer science communities like the GI (Gesellschaft für Informatik).
Since 1986, the number of important researchers grew too large to be listed here. There are many good books covering the history since 1986 in full detail.

- The next generation (1995 – today): Although ANNs have been applied very successfully to arbitrary kinds of static pattern recognition, their application in the processing or recognition of dynamic, non-stationary patterns was very difficult and unsolved in many application domains. To allow the general advantages of Neural Networks to be applied to embedded systems – where the temporal aspect of signals and the response of the system to external events are in the main point of view – the need of a new model arose.
  To the best of the authors knowledge, models of Neural Networks that are comparable to the model of Spiking Neural Networks first appeared in 1995, although some approaches were made a bit earlier (e.g. [GvH94, JA93, Wat94,

HT87]).

It was again John Hopfield who made an important step in the topic of integrating the temporal aspect of biological systems into Artificial Neural Networks [Hop95]. But nearly at the same time, Wolfgang Maass published his model of SNNs [Maa95, Maa96], which is also used in this diploma thesis. The author was unable to reconstruct the exact series of papers that led to the introduction of Spiking Neural Networks, but since that time it has been an active and fruitful research topic (e.g. [GML99, Gel89, Gel90, Ruf98, Maa99a, RWdRvSB97, Maa99b, Maa97b, RS98, Maa97a] and many more).

2.2 Spiking Neural Networks

Spiking Neural Networks (SNNs) are considered as the third generation of artificial neural networks and try to model the biological behavior more closely than the last generation. Although the currently used Artificial Neural Networks (ANNs) which use the firing rate of neurons as their computational element have proven to be very powerful for specific kinds of problems, some major properties of biological neural networks are ignored. Through empirical evidence from biological experiments [TFM96] it became clear that some effects and computations of the human brain [CBG01] cannot be carried out by just using the firing rate of the respective neurons as information transmitter – additionally, the exact timing of single spikes has to carry at least some of the transmitted information.
In the following subsections, an introduction into the research topic of Spiking Neural Networks will be given. This introduction will start with a motivation for the model of SNNs, followed by a short explanation of biological neurons. After that, a few models of biological neurons will be summarized to lay the grounds for the detailed definition of the formal model of SNNs that will be used as the basis for this diploma thesis.

2.2.1 Motivation

ANNs are currently used very successfully in some assorted sets of applications, so the question arises why there is a need for SNNs ? There seems to be a limit on what ANNs are able to do and although many possible extensions of ANNs have been developed, they do not seem to come close to what the human brain can do. One of the problems that ANNs cannot really solve is the simulation of the oscillation and synchronization effects in the human brain. Although simulating these effects on itself might not gain computational power, recent work suggests that oscillation and synchronization might be of high importance for parts of the vision system [Hen02].
As already stated in the introduction, Artificial Neural Networks typically encode the firing rate of biological neurons as real numbered values which are used as input and output values of the neurons [Zel94]. However, there is growing empirical evidence [TFM96] for the importance of the timing of single spikes. It has been shown that the human brain can process visual patterns in 150 msec, where about 10 processing levels (neuron layers) are involved and neurons in these regions of the human brain usually have a firing rate of less then 100 Hz. Therefore, since using a firing rate code would involve averaging over the firing times of at least 2 received spikes, the processing time available to each layer is not sufficient for estimating the firing rate – and therefore the output values – of neurons. As a result of these observations it can be argued that the computation has to be carried out using only the information transmitted with the first spike that is fired by each layer.

In principle, there are just three different possibilities currently known in neurophysiology for encoding information in such spike trains [Ruf98]:

- rate coding: The essential information of the spike train is encoded in the firing rates, which are determined by averaging over some time window.
- temporal coding: The timing of single spikes, i.e. the exact firing time, carries the encoded information.
- population coding: The information is encoded by the activity of different populations of neurons, i.e. the percentage of neurons concurrently active in a population.

As mentioned above, ANNs are typically limited to using a simulation of rate coding by passing real numbered values between the neurons. However, this is not powerful enough to solve some of the problems that biological neural networks solve easily [Maa97b]. It is currently not clear if this is the main reason why ANNs seem to be unable for solving some special problems [CBG01], but SNNs offer more flexibility while also allowing straight-forward solutions of simpler problems.
In the context of fast information processing, temporal coding seems to be the most important coding scheme. Therefore, this diploma thesis mainly focuses on this coding scheme; but all of the developed principles and techniques can be applied directly to arbitrary coding schemes - only the network inputs and outputs have to be adapted (see section 3.4), the inner network structure can remain unchanged. However, for the simulations described in sections 5.2 and 5.3 rate coding was used, which proves that the developed methods as well as the current implementation are indeed completely independent of the coding scheme.

Another advantage of SNNs over ANNs is that they can solve some of the problems that ANNs are currently used for with less or simpler neurons. One example is the simulation of RBF (Radial Basis Functions) networks: when an RBF network is to be built as an ANN, the neurons have to use special activation functions to achieve the RBF effect. Therefore, an RBF network cannot be built of the same neurons that are used in a back-propagation network. With SNNs, this behavior can easily be achieved [Maa99a]. Moreover, SNNs as defined in this diploma thesis already contain additional parameters that can be used as the center of RBF neurons in a very intuitive way. These parameters are the synaptic delays, which will be described in more detail in section 4.2.
Another, more theoretical example for a problem that can be solved more easily with SNNs is coincidence detection. Coincidence detection means the detection of events that occur at the same time or in a short time window. When using temporal coding, the

equivalent for ANNs would be equality detection, i.e. detecting if two or more inputs have the same or nearly the same value. With ANNs, solving this problem for n inputs needs at least $\frac{n-4}{2}$ neurons [Maa97b], while with SNNs this can be achieved with a single neuron for an arbitrary number of inputs. Although this might not be relevant for practical applications, it shows that SNNs can also offer advantages for problems that are already solvable with ANNs.
However, it remains to be shown that the additional complexity of SNNs – which leads to more complex simulation algorithms – does not cause more overhead than what is gained in terms of new flexibility. The ultimate goal should always be to utilize the available processing power most efficiently when solving a given problem, so SNNs have to compete against ANNs in various problem areas. For SNNs to be adapted for practical applications, they will need to be able to solve new problems and to solve currently known problems more efficiently, with additional flexibility or with higher solution quality then ANNs – at least for some application domains.

2.2.2 Biological Neurons

Before someone is about to conduct research on Artificial Neural Networks, it is always advisable to look at the biological model, i.e. neurons and neuronal structures of animals and humans. However, when inspecting those complex models, it quickly becomes clear that they do not have very much in common with current ANNs; there is simply too much abstracted. The following part of this subsection is summarized from the respective chapters in [Zel94] and [Ruf98].
Nervous cells, the basic parts of the brain, are different from other cells mostly in their shape, their type of cell membrane and their property to form bulges (*Synapses*) at their ends to connect to other nervous cells. Furthermore, they are normally not able to reproduce themselves by cell division anymore after the embryonic phase.
In Fig. 2.1 a typical neuron is shown schematically, including its cell body *(soma*), its inputs (*dendrites*) and its output (*axon*). Neurons are connected to each other through *synapses*. Essentially they operate on an electro-chemical basis, where the interior of a neuron is separated from the surrounding by a *membrane* that contains *ion channels* which are highly specific to their respective sort of ions; one distinguishes between Na^+, K^+, Ca^{2+} and Cl^- channels. The potential difference across the membrane is called the *membrane potential.* It is normally in an equilibrium state due to the different concentrations of the ions in the interior and the outside, which cause an osmotic pressure, and the charge displacement opposing this concentration gradient. When there is no input to the neuron, the membrane potential will reach the so-called *resting membrane potential* E_{rest} which is usually between -40 and $-90mV$ depending on the type of the neuron. Ion channels can either be active or passive; passive ion channels have

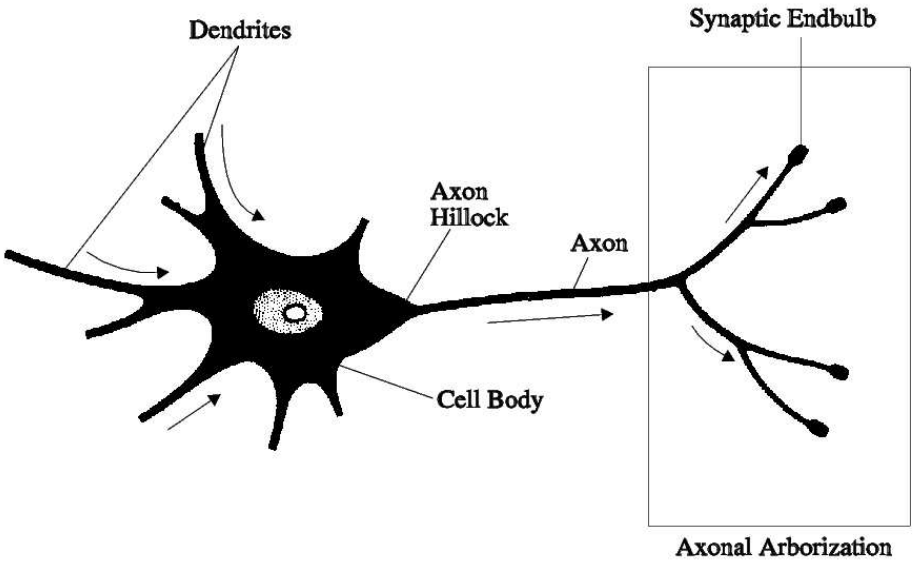


Figure 2.1: Schematic view of a nervous cell (from [Ruf98], after [Arb89]).

	Value
Average output connectivity (number of connections to subsequent neurons)	ca. 1000 to 10000
Length of an axon	few mm to ca. 1 m
Thickness of an axon	0.3 to 1.3 μm
Width of the synaptic gap	ca. 20 nm
Number of neurotransmitters in a synaptic vesicle	ca. 1000 to 10000
Duration of a nervous impulse (spike)	ca. 1 ms
Thickness of the neuron cell membrane	ca. 5 nm
Resting membrane potential	-70 mV
Amplitude of a nervous impulse (Spike)	ca. 100 mV
Electrical field of the membrane in equilibrium	ca. 12000 V/mm
Transmission speed in an axon	ca. 120 m/s
Speed of diffusion of neurotransmitters	ca. 2 mm/min
Transmission time of a synapse	ca. 0.6 ms
Membrane capacitance	ca. 1μF/cm^2

Table 2.1: Some numbers about neurons (from [Zel94]).

a constant conductance, the conductance of active ones can vary depending on certain factors. When the membrane potential reaches some threshold, which is normally at about $-30mV$, a pulse is generated that propagates along the axon (a s*pike*). After a spike has been generated (the neuron has *fired*), it enters a *refractory period* consisting of two parts: the *absolute refractory period* in which no spike can be generated (the threshold is infinitely high) and the *relative refractory period* in which the threshold is higher. Table 2.1 shows a few selected numbers about biological neurons.

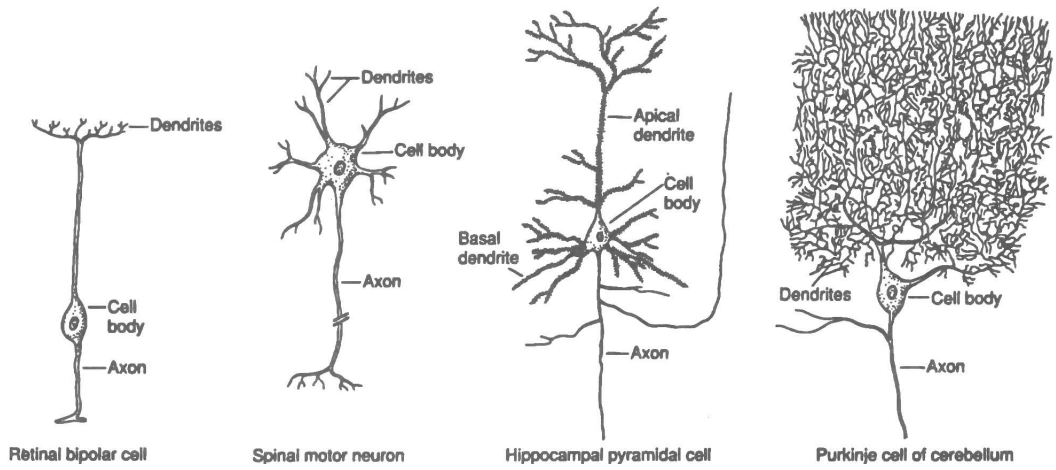


Figure 2.2: Different types of multi-polar neuron cells (from [KSJ91]).

It is commonly assumed that humans have about 100 billions (10^{11}) of neuron cells. Although all of them are different in their shape, they can be classified according to their structure and region in the brain. *Unipolar cells* have, besides their soma and their *nucleus*, only one appendage, the axon. *Bipolar cells* additionally have one dendrite and *multi-polar cells* have one axon and many dendrites (see Fig. 2.2). Dendrites are thin, tube-like and mostly ramified appendages of neuron cells which act as the inputs of the cell. Forwarding the electrical impulses (spikes) is done via the – normally much longer – axon. The axon differs from the dendrites in its structure and the properties of its membrane and can have a length between a few millimeters and almost one meter (cf. table 2.1); but only in the end region it starts to ramify into the *axonal arborization* (cf. Fig. 2.1). At the end of those ramifications end-bulbs, the synapses, are formed. Usually human neuron cells have about 1000 to 10.000 of such synapses to connect to subsequent neurons, but some cells such as the *Purkinje* cells can have up to 150.000 synapses. Most of the neurons receive input from about 2000 to 10.000 other neurons. A spike that is propagated along the axon finally branches to the synapses, which converts this *pre-synaptic spike* to a *post-synaptic potential* (*PSP*), influencing the membrane potential of the *post-synaptic neuron*. Such a post-synaptic potential can either increase the membrane potential, making the firing of the neuron more likely – this is called an *excitatory post-synaptic potential* (*EPSP*) caused by an *excitatory synapse*, or decrease it, making the firing less likely – this is called an *inhibitory post-synaptic potential* (*IPSP*) caused by an *inhibitory synapse* (see Fig. 2.3 for examples of PSPs). The strength of those PSPs depends on the strength of the synapse, which may vary over time. As the synapses may be placed either on the dendritic tree or directly at the soma, PSPs can interact in time and space. Although in biological neurons there are several other non-linear effects involved in the PSPs influencing the membrane potential (for details refer to [Ruf98, chapter 2] and [Zel94, chapter 2]), it is usually abstracted to the

PSPs summing up linearly in the soma. This abstraction is the so-called *integrate-and-fire neuron* (*IFN*) model or an extension thereof, the *leaky integrate and fire neuron* model (*LIFN*, also refer to subsection 2.2.3).
It is very important to note that the action potential (spike) is equal for all neurons, there are no significant differences in the amplitude or shape. Therefore a single spike cannot transmit any other information than the space (of the axon) and the time (of the firing). In [Zel94, page 42] and many other scientific papers and books, it is claimed that the information about the strength of the output signal is coded in the frequency and duration of the spikes. However, it is questionable whether this is indeed true. This diploma thesis follows the argumentation of various other authors and proposes that the exact timing of single spikes also carries important information (refer to subsection 2.2.1).
There exists strong evidence that the basis for learning is formed by synaptic plasticity, i.e. the possibility that the strengths of synapses vary over time. Donald O. Hebb was the first one to address the question on how synaptic weights can be modified to store information. He postulated the nowadays called "*Hebbian*" learning in 1949 in the following way [Heb49]:

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased.

Therefore, similar activation patterns in the pre-synaptic and post-synaptic neuron strengthen a synapse, which can be formally written as the *Hebb rule* for the modification of a weight w_{AB} for a synapse from neuron A to neuron B:

$$\Delta w_{AB} = \eta \cdot V_A \cdot V_B$$

where $\eta > 0$ is the learning rate and V_A and V_B denote the respective activities of the neurons A and B. For Hebbian learning, all information has to be locally available at the synapse; therefore the information from both the pre-synaptic and the post-synaptic neurons needs to be known. At the moment there exist different formulations of the Hebb rule, depending on the type of interaction. For details, refer to [Ruf98].
In the following section, a few mathematical models describing the biological model in various levels of abstraction are listed and a short introduction into the biologically realistic Hodgkin-Huxley model is given.

2.2.3 Mathematical Model for Biological Neurons

Currently there are various mathematical models which can be used for modeling and simulating Spiking Neural Networks. These models range from the biologically very

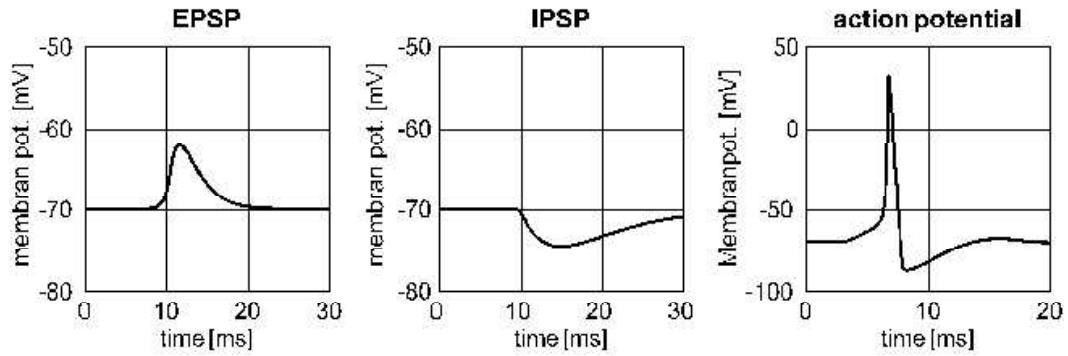


Figure 2.3: Post synaptic potentials and action potential, picture taken from http://www.cis.tugraz.at/igi/tnatschl/online/3rd_gen_ger/node1.html

realistic Hodgkin-Huxley [HH52] and compartmental models, which use differential equations to model the electrical properties of neurons via the integrate-and-fire model [Ger95, Tuc88] respectively the leaky integrate-and-fire model to the rather simple spike-and-response model [Ger98, GvH94] – all of them are using spike timings as their primary code for computation. Usually they are simulated continuously with fixed time-steps because of the representation of some mathematical functions in the model, especially the function modeling the post synaptic potential generated by synapses. The post-synaptic potentials are the changes induced to the neuron potential of their subsequent neuron (the post-synaptic neuron); they are generated by synapses whenever they receive a spike (from the pre-synaptic neuron). Fig. 2.3 shows the shape of a spike (action potential) and an excitatory and an inhibitory post synaptic potential (EPSP, IPSP).

The Hodgkin-Huxley model – or an extension thereof, the compartmental model – is used by some current simulation tools, such as the widely used GENESIS simulator [BB94]. The following description of these models is a summary of [Ruf98, chapters 2.2.1 and 9.1].

In 1952, Hodgkin and Huxley conducted research on the squid giant axon and developed their Hodgkin-Huxley model describing the initiation and propagation of action potentials. In these axons, the membrane potential V_m is dependent on the the passive ion channels (resulting in a leak conductance g_l) and active K^+ and Na^+ ion channels (with voltage-dependent conductances g_K and g_{Na}). The reversal potentials resulting from the ion concentration differences are $E_l = -54,3mV$, $E_{Na} = 50mV$ and $E_K = -77mV$, the resting membrane potential is about $E_{rest} = -65mV$. The Hodgkin-Huxley model can be specified as an equivalent electrical circuit describing the time dependencies of the active ion channels and the influence of the conductances on the membrane potential V_m. In Fig. 2.4 this equivalent circuit is shown; it can be formally described by

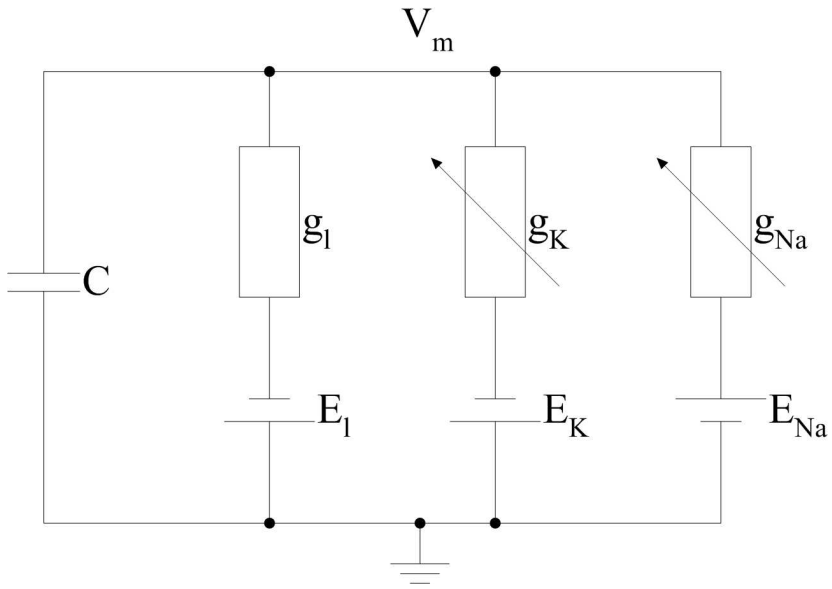


Figure 2.4: Electrical circuit describing the membrane potential (after [Ruf98]).

$$C\frac{dV_m}{dt} = g_l(E_l - V_m) + g_{Na}(E_{Na} - V_m) + g_K(E_K - V_m)$$

where C is the membrane capacitance. The ionic currents are described by the terms $g_i(E_i - V_m)$ whereas the conductance g_l is constant and g_{Na} and g_K are voltage-dependent:

$$\begin{aligned} g_{Na} &= G_{Na} \cdot m^3 \cdot h \\ g_K &= G_K \cdot n^4 \end{aligned}$$

where G_{Na} and G_K are constants for the maximum conductances. The property that ion channels can be blocked is modeled by the time- and voltage-dependent state variables $m, n, h \in [0,1]$. A typical action potential as generated by these equations is shown in Fig. 2.5, parallel to the changes of m, n and h. For details, refer to [Ruf98] and [HH52].

This model of the time-dependent functioning of the squid giant axon has been extended to allow the division of neurons into a finite number of interconnected components, so-called compartments. Although an axon can also be modeled using multiple compartments, it is normally not needed. But complex dendritic trees of some neurons (such as a GENESIS model of a Purkinje cell consisting of 4550 compartments) can be simulated biologically very realistically using the extended model. The equivalent circuit of a compartment as shown in Fig. 2.6 can be expressed by

$$C_m\frac{dV_m}{dt} = \frac{E_m - V_m}{R_m} + \sum_k((E_k - V_m)\cdot g_k) + \frac{V_m' - V_m}{R_a'} + \frac{V_m'' - V_m}{R_a} + I_{inject}$$

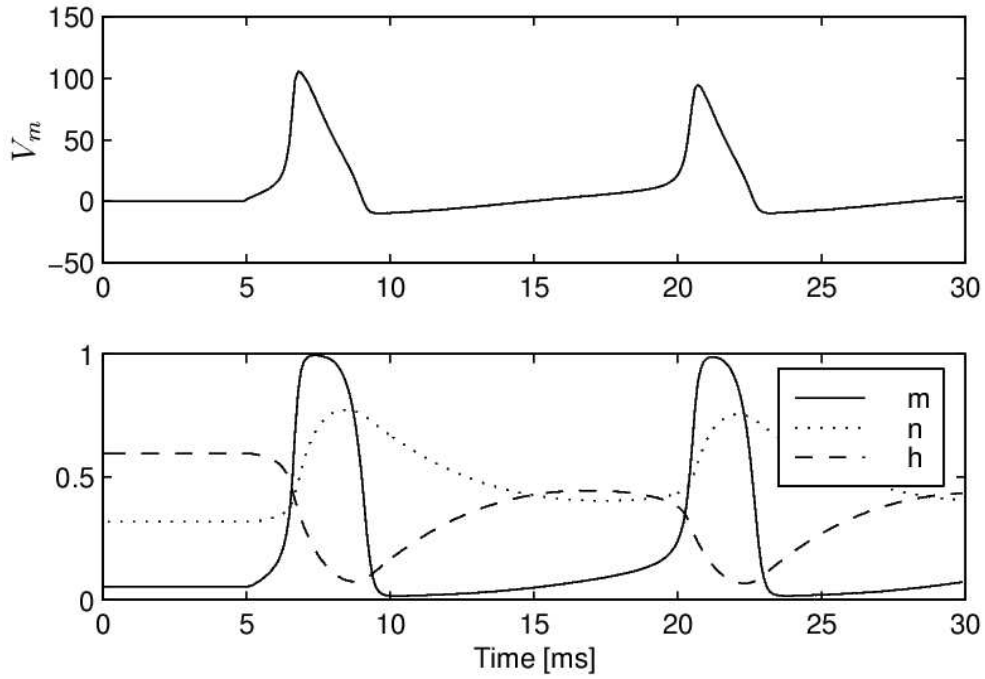


Figure 2.5: The trajectories of V_m and the state variables during the generation of an action potential (from [Ruf98]).

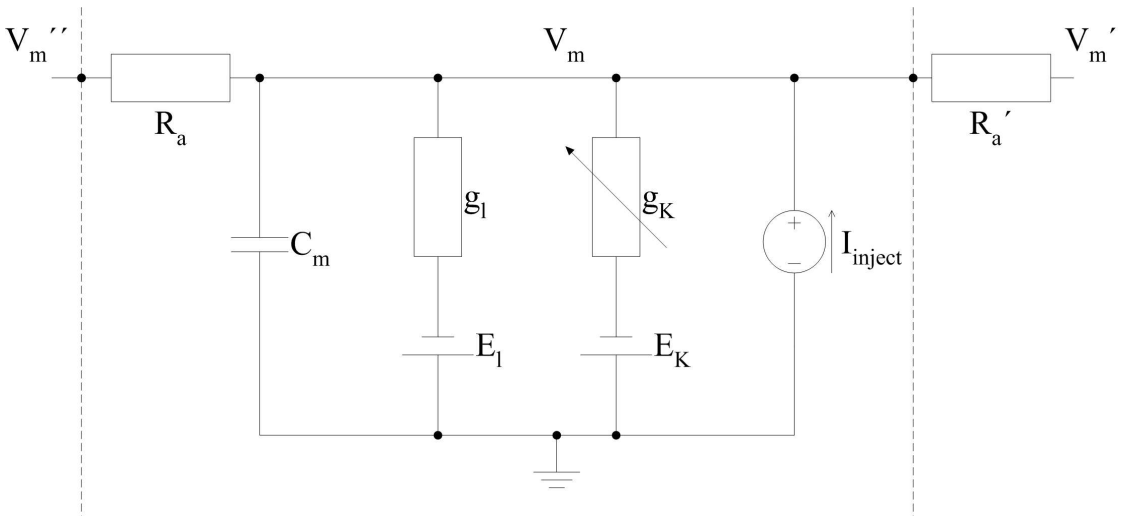


Figure 2.6: Electrical circuit for a basic compartment (after [Ruf98]).

For details the reader is again referred to [Ruf98].

As the previous description might already suggest, this biologically very realistic model is too complex for solving problems of mainly computational nature or for theoretical considerations on the computational power of different network models. Therefore, in the next chapter a simpler, more abstract model of SNNs is described

which is better suited for these tasks.

2.2.4 Mathematical Model for Spiking Neurons

In this diploma thesis the following definition of spiking neural networks, directly cited from [Maa95], is used as the basis for construction a model of SNNs that is optimized for fast simulation:
An SNN consists of:

- *a finite directed graph $\langle V,E\rangle$ (we refer to the elements of V as "neurons" and to the elements of E as "synapses")*
- *a subset $V_{in}\subseteq V$ of input neurons*
- *a subset $V_{out}\subseteq V$ of output neurons*
- *for each neuron $v\in V-V_{in}$ a threshold function $\Theta_v:\mathbb{R}^+\to\mathbb{R}\cup\{\infty\}$*
- *for each synapse $\langle u,v\rangle\in E$ a response function $\varepsilon_{u,v}:\mathbb{R}^+\to\mathbb{R}$ and a weight $w_{u,v}\in\mathbb{R}^+$*

We assume that the firing of input neurons $v\in V_{in}$ is determined from outside of the SNN, i.e. the sets $F_v\subseteq\mathbb{R}^+$ of firing times ("spike trains") for the neurons $v\in V_{in}$ are given as the input of the SNN.
For a neuron $v\in V-V_{in}$ one defines its set F_v of firing times recursively. The first element of F_v is $\inf\{t\in\mathbb{R}^+ : P_v(t)\geq\Theta_v(0)\}$, and for any $s\in F_v$ the next larger element of F_v is $\inf\{t\in\mathbb{R}^+ : t>s \text{ and } P_v(t)\geq\Theta_v(t-s)\}$, where the potential function $P_v:\mathbb{R}^+\to\mathbb{R}$ is defined by

$$P_v(t) := \sum_{\substack{u\in V\\ \langle u,v\rangle\in E}} \sum_{\substack{s\in F_u\\ s<t}} w_{u,v}\cdot\varepsilon_{u,v}(t-s)$$

The firing times ("spike trains") F_v of the output neurons $v\in V_{out}$ that result in this way are interpreted as the output of the SNN.
The complexity of a computation in an SNN is evaluated by counting each spike as a computation step.

This model represents a rather general definition of SNNs and is related to biological neurons as follows: the shape of a post synaptic potential (PSP) caused by an incoming spike from neuron u is described by the response function $\varepsilon_{u,v}$, where $\varepsilon_{u,v}(t)=0$ for $t\in[0,d_{u,v}]$ with $d_{u,v}$ modeling the delay between the generation of the action potential and the time when the resulting PSP starts to influence the potential of neuron v.

Currently the ways of simulating such SNNs differ heavily from the way ANNs are used in today's applications. This is caused by the immanent complex behavior of spiking neurons compared to the relatively simple behavior of sigmoidal neurons used in ANNs. Sigmoidal neurons usually only sum up the analog input values they receive and utilize a sigmoid function to compute their output value from this sum, while spiking neurons try to model the behavior and partly the structure of biological neurons more closely. Therefore spiking neurons are often simulated using the biologically very realistic Hodgkin-Huxley model, which uses differential equations to describe the shape of neuron potentials and generated spikes (see subsection 2.2.3). These differential equations impose the use of continuous simulation techniques, where the values of mathematical functions are computed at simulation times with fixed or variable time steps. However, with this simulation technique, the inner states of neurons which are inactive at some simulation time also need to be computed, resulting in bad scalability of the simulation in terms of large networks with many neurons. A better simulation technique – concerning scalability – is the so-called discrete event simulation where computation is only necessary for simulation elements that change their state at the current simulation time. In large networks where only a small percentage of neurons is active at any given time, discrete event simulation is expected to be significantly faster.

2.3 Discrete Event Simulation

Various parts of this chapter have been taken from the respective parts in the FWF project proposal "DEVS Simulation of Spiking Neural Networks", to which Herbert Prähofer has kindly contributed many corrections and additions. The model itself, the extensions and the description thereof have been summarized from [ZPK00].
Discrete event simulation [ZPK00, BCN01, Ban98] is a technique for modeling and simulating systems where the system behavior is abstracted to discrete events. In contrast to continuous simulation, where all system states are computed for the simulation time points (either with fixed or with variable time steps), discrete event simulation calculates the simulation time points when the system states reach specific values.
Discrete event simulation is a widely used technique in such diverse application areas as performance evaluation of manufacturing, transport, communication, and computer systems, verification of VLSI systems, and others. The great benefit of discrete event simulation compared to continuous simulation or time discrete simulation [ZPK00] is that the behavior of the systems is reduced to essential events. Simulation jumps from one event time to the next event time. Between event times, no simulation has to be done. Moreover, system parts which are in a quiet phase, i.e, parts where no events occur at the time, do not have to be simulated at all. The simulation can be restricted to the active areas of the system.

In this way, discrete event simulation has shown to have great benefits compared to continuous simulation. For example, discrete event simulation allows to simulate large VLSI systems at the gate level for which no continuous simulation is feasible. In [Moo96] the idea of discrete event simulation has been adopted for simulating ecosystems, an application area which is usually modeled with partial differential equations and simulated with computationally demanding finite difference or finite element methods. Research has shown that the discrete event simulation technique led to simulation performance improvements with a factor of 100 and more. It allowed to simulate models of a size which never had been tackled before [ZY96, ZD97].
Spiking neural networks show similar characteristics as the application areas above. As shown in the next section, SNNs can be modeled using the DEVS discrete event system formalism [ZPK00]. Spikes can naturally be modeled with event signals. The usual activity patterns observed in SNNs, where spikes spread in a restricted area of the whole network, should lead to dramatic performance improvements.

The following classic DEVS system specification is directly cited from [ZPK00], but has been slightly adapted to match the model extensions better:
A discrete event system specification *(*DEVS*) is a structure*

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

- X *is the set of input values*
- Y *is the set of output values*
- S *is a set of states*
- $\delta_{int} : S \to S$ *is the* internal transition *function*
- $\delta_{ext} : Q \times X \to S$ *is the* external transition *function, where*
  - $Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ *is the* total state *set*
  - e *is the* time elapsed *since last transition*
- $\lambda : S \to Y$ *is the output function*
- $ta : S \to \mathbb{R}^{+}_{0,\infty}$ *is the time advance function*

The interpretation of these elements is illustrated in Fig. 2.7. At any time the system is in some state, s. If no external event occurs, the system will stay in state s for time $ta(s)$. *Notice that* $ta(s)$ *could be a real number as one would expect. But it can also take on*

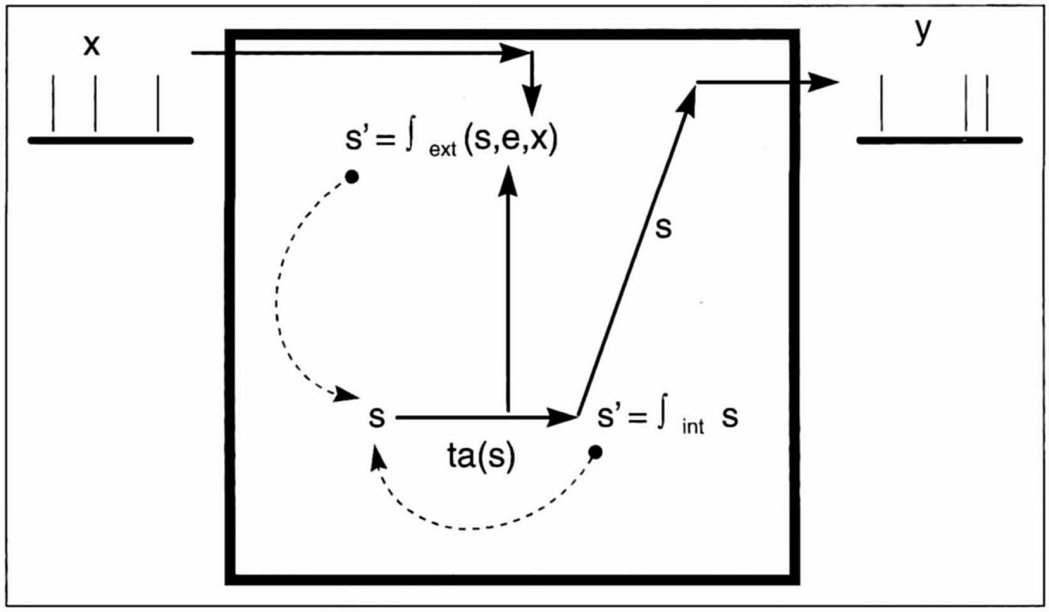


Figure 2.7: Overview over the parts of the DEVS model (form [ZPK00]).

the values 0 *and* ∞*. In the first case, the stay in state s is so short that no external events can intervene – we say that s is a* transitory *state. In the second case, the system will stay in s forever unless an external event interrupts its slumber. We say that s is a* passive *state in this case. When the resting time expires, i.e., when the elapsed time,* $e = ta(s)$*, the system outputs the value,* $\lambda(s)$*, and changes to state* $\delta_{int}(s)$*. Note that output is only possible just before internal transitions.*
If an external event $x \in X$ *occurs before this expiration time, i.e., when the system is in total state* (s,e) *with* $e \leq ta(s)$*, the system changes to state* $\delta_{ext}(s,e,x)$*. Thus, the internal transition function dictates the system's new state when no event occurred since the last transition. The external transition function dictates the system's new state when an external event occurs – this state is determined by the input, x, the current state, s, and how long the system has been in this state, e. In both cases, the system is then in some new state* s' *with some new resting time,* $ta(s')$*, and the same story continues.*

Following the above definition, the mentioned leaping from one simulation time to the next can be explained easily. An external event, such as a spike being received from the outside of the network, triggers an internal state change which advances the system from one discrete state to another one. The same happens for internal events, such as spikes that are fired by a neuron inside the network. Therefore it is never necessary to simulate the trajectories of system states between discrete events. Instead, the power of the simulation system lies in the implementation of the functions $\delta_{int}(s)$, $\delta_{ext}(s,e,x)$ and $ta(s)$ which depend heavily on the chosen state values $s \in S$. Furthermore, the sys-

tem will only produce outputs on state changes (internal or external) using the output function $\lambda(s)$.

There are a few extensions to this classic DEVS model with the parallel DEVS model being the most advanced [ZPK00]. However, for this diploma thesis, the parallel DEVS model is not as important because the developed simulation framework is currently targeted towards sequential single-processor machines. However, a short discussion about future enhancements using parallel simulation is given in section 6.3. In the following, two simpler extension are shortly summarized which are of importance for the object oriented implementation of the simulation framework.
The first extension is the *classic DEVS with ports* model which differs from the classic model only in the definitions of the input, output and state sets:

$$M_{Ports} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

- $X = \{(p,v) \mid p \in InPorts, v \in X_p\}$ *is the set of input ports and values*
- $Y = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ *is the set of output ports and values*
- *S is the set of* sequential *states*

This model is more concrete than the classic DEVS model in the sense that it makes modeling easier by introducing the notation of finite numbers of input and output ports instead of the abstract notation of input and output sets. The second extension is the *classic DEVS coupled* model which additionally includes the means to build models from components; this allows the formation of hierarchical specifications:

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select \rangle$$

where

- $X = \{(p,v) \mid p \in InPorts, v \in X_p\}$ *is the set of input ports and values*
- $Y = \{(p,v) \mid p \in OutPorts, v \in Y_p\}$ *is the set of output ports and values*
- *D is the set of component names*
- *components are DEVS models, for each* $d \in D$,
  - $M_d = \langle X_d, Y_d, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ is a DEVS *with*
    $X_d = \{(p,v) \mid p \in InPorts_d, v \in X_p\}$,
    $Y_d = \{(p,v) \mid p \in OutPorts_d, v \in Y_p\}$

- *external input coupling connect external inputs to component inputs,*
  $EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in InPorts, d \in D, ip_d \in InPorts_d\}$
- *external output coupling connect component outputs to external outputs,*
  $EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OutPorts, d \in D, op_d \in OutPorts_d\}$
- *internal coupling connects component outputs to component inputs,*
  $IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OutPorts_a, ip_b \in InPorts_b\}$
  However, no direct feedback loops are allowed*, i.e., no output port of a component may be connected to an input port of the same component:*
  $((d, op_d), (e, ip_e)) \in IC \Rightarrow d \neq e$
- *Select:* $2^D - \{\} \rightarrow D$*, the tie-breaking function (used in classic DEVS but eliminated in parallel DEVS)*

The possibility to construct complex models hierarchically stems from the notation that components of the classic DEVS coupled model are themselves DEVS (with ports) models, which of course can again be DEVS coupled models.

Chapter 3

Model

The investigation of the potential of SNNs can be regarded as an open research issue [RWdRvSB97] – their analysis requires powerful simulation systems which allow to simulate large networks of neurons. However, no such simulators are currently available. The GENESIS simulator [BB94] is currently considered as the standard simulator for SNNs. It is a continuous simulator working at the level of continuous signals and is primarily used for conducting research on biological neural networks. With its accuracy in emulating biological neurons in detail, it is widely accepted and often used for this task. But, due its simulation execution demands, it does not allow to simulate large networks for tackling mainly technical problems. Other research on SNNs also use continuous simulation as analysis technique (e.g. [Ruf98, SM01, MN97, BB94, QC01, CBG01]). Because of the computationally expensive continuous simulation technique, there were quite some attempts at making the simulation faster [SM01, NT].
This diploma thesis proposes the adoption of discrete event simulation techniques for building an SNN simulator. Discrete event simulation abstracts from continuous signals to event occurrences and, by that, usually shows dramatic reduction in simulation execution time and/or increase of model size. SNNs, due to the event-like occurrences of spikes, lend themselves naturally for discrete event simulation.
In the following, a discrete event model for SNNs is outlined which should serve as the basis for building an SNN simulator.

The scientifically new concept of this diploma thesis – which already has been presented in [MAP+02] – is to combine the SNN and DEVS models to achieve a flexible, fast and scalable simulation of powerful SNNs.

To accomplish this task, a new model of Spiking Neural Networks was developed in cooperation with Michael Affenzeller, Herbert Prähofer, Gerhard Höfer and Alexander Fried. This novel model has been designed especially for discrete event simulation. It is based on the integrate-and-fire model and the mathematical formalization of Spik-

ing Neural Networks introduced in [Maa95] and cited in subsection 2.2.4, but does not use continuous functions for modeling post synaptic potentials and neuron potentials. Instead of this, piecewise linear functions are applied, which offer the possibility to exactly and easily calculate the firing times of neurons using their potential and threshold functions.
In the next section, piecewise linear functions as used in this diploma thesis will be thoroughly defined and the advantages of their application will be analyzed.

3.1 Piecewise linear functions

The neuron will fire whenever the value of its potential is equal to or higher than the value of its threshold. Therefore, calculating the next firing time is equivalent to calculating an intersection between two piecewise linear functions, which can be done efficiently. It has been shown in [Ruf98] that spiking neural networks using piecewise linear functions are real-time equivalent to a Turing machine, i.e. the simulation of one computation step of a Turing machine by an SNN constructed especially for this purpose only takes a fixed amount of computation steps within that SNN. Therefore, the use of piecewise linear functions as synaptic response, potential and threshold functions describing the inner neuron state should not affect the qualitative computation power of SNNs negatively. Nevertheless, simulations will make it possible to study the PSP shape's quantitative influence on the computational efficiency of SNNs. By the use of piecewise linear functions, our model makes it possible to approximate the shape of different functions as closely as needed or wanted in different applications. Additionally there is a gain in flexibility because there is no restriction to mathematical functions with a closed representation. Currently, studying the effects of different shapes of synaptic response functions is difficult because closed representations of these functions have to be found (for the numeric integration techniques – used in continuous simulations – to work). Within our new model, this restriction no longer exists, offering neuro-biologists more degrees of freedom in their studies of neural models.
For efficient handling and computations we use a special optimization: not representing the piecewise linear functions by a list of time/value tuples for the function points but by a list of time/gradient tuples for the function segments. This optimization has been suggested by Alexander Fried. Furthermore, in this thesis the functions are defined to start and end at a value of zero.
Therefore, a piecewise linear function $f(t)$ is defined as follows:

$$
\begin{aligned}
f &:= \left\langle \langle s_i, \lambda_i \rangle \mid s_i \in \mathbb{R}_0^+, \lambda_i \in \mathbb{R}^+, (\forall u, v)\ (0 < u < v < len(f))\ s_u < s_v \right\rangle \\
f(0) &:= 0 \\
f(t) &:= \sum_{i=1}^{k-1} (f_{i+1}^s - f_i^s) \cdot f_i^\lambda + (t - f_k^s) \cdot f_k^\lambda
\end{aligned}
$$

with $k = \max_i(s_i < t)$ (the index of the last gradient change before the current time t). f is a vector of tuples $\langle s_i, \lambda_i \rangle$ where s_i are unique, strictly monotonic increasing time values and λ_i are gradient values. The gradients are defined as the multiplicands in the linear function equations

$$y = \lambda_i \cdot x + d_i$$

Between the time points s_i and s_{i+1} the function has a constant gradient λ_i, $s_0 := 0$, $f(0) := 0$, $\lambda_0 := 0$ and $\lambda_{m+1} := 0$ for m being the number of gradient changes in f. The additional constraint $f(s_{m+1}) := 0$ must be satisfied to make the function bounded. Only the segments $\langle s_i, \lambda_i \rangle$ for $(i = 1, ..., m)$ need to be specified to completely define $f(t)$ for all t, because λ_{m+1} is defined to be zero and s_{m+1} (the time when the function again reaches a value of 0) can be computed from the other values. Fig. 3.2 shows the shape of such a piecewise linear function with 5 segments.
This makes it possible to represent a piecewise linear function consisting of N segments with N time/gradient tuples and also offers a wide range of possible optimizations in the simulation.
For working with piecewise linear function as defined in this section, the following notation will be used:

- f_i^s denotes the starting time of the i-th segment (the time value of the i-th tuple of the vector f)
- f_i^λ denotes the gradient of the i-th segment (the gradient value of the i-th tuple of the vector f)
- f^s denotes the set of all starting time values in f, i.e. $f^s := \{f_i^s \mid 0 \leq i \leq len(f)\}$
- f^λ denotes the set of all gradient values in f, i.e. $f^\lambda := \left\{ f_i^\lambda \mid 0 \leq i \leq len(f) \right\}$

3.2 Spiking Neurons

Furthermore, this novel model treats neurons and synapses as active elements and uses single spike events for modeling the spikes that a neuron emits; only indicating the exact firing times but ignoring the shape of spikes (which are also not modeled in the widely used integrate-and-fire model). As neurons are not coupled directly but

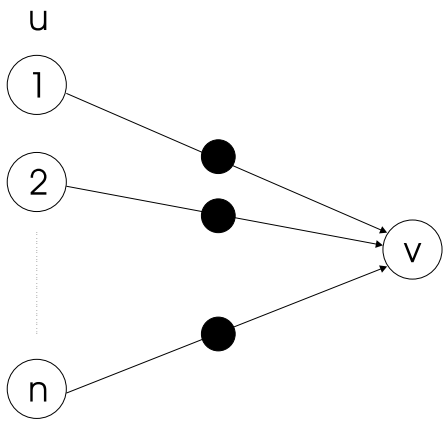


Figure 3.1: The input synapses of a neuron v receive spike events from neurons $u = 1,...,n$, therefore generating the input values of neuron v.

only via synapses to each other, it are the synapses that receive the spike events sent by the neurons. Each synapse receives spike events from only one neuron (the pre-synaptic neuron) and has only one neuron to send events to (the post-synaptic neuron). Therefore, the network forms a bipartite graph of neurons and synapses with synapses being positioned between neurons as shown in Fig. 3.1. This diploma thesis uses the notation that a neuron v receives its input from neurons $u = 1,...,n$, which send their spike events to the respective input synapses of neuron v.
Those synapses $\langle u,v \rangle$ are, upon the receipt of a spike event from their pre-synaptic neuron u, also responsible for generating the post-synaptic potentials in the form of piecewise linear functions $\varepsilon_{u,v}(t)$ and forwarding them to their post-synaptic neuron v, scaled by their specific synaptic weight $w_{u,v}$ and deferred by their synaptic delay $d_{u,v}$. Fig. 3.2 shows the input and output of a synapse which is generating an excitatory post-synaptic potential due to the receipt of a spike. After a neuron v has received a new post-synaptic potential from one of its input synapses $\langle u,v \rangle$, it merges it with its currently valid potential $P_v(t)$ and recalculates if and when the next firing occurs by calculating the time when the potential function $P_v(t)$ intersects with the threshold function $\Theta_v(t)$. The neuron threshold is modeled as a time dependent function instead of a constant value to prevent a neuron from instantly firing again because the potential $P_v(t)$ might still be higher than the threshold value. Therefore, the threshold function $\Theta_v(t)$ is defined to be infinite for some time $\tau_{v,ref}$ (the so-called absolute refraction period of the neuron) after the neuron v has fired, effectively preventing it from firing until the threshold has a finite value again. After this absolute refraction period ends at relative time $\tau_{v,ref}$, the neuron enters the relative refraction period, during which the threshold function rapidly approaches its constant value $\Theta_v(0)$, which by definition is reached at relative time $\tau_{v,end}$. In Fig. 3.4 the firing of a neuron v due to the intersection of the potential and the threshold functions is shown.

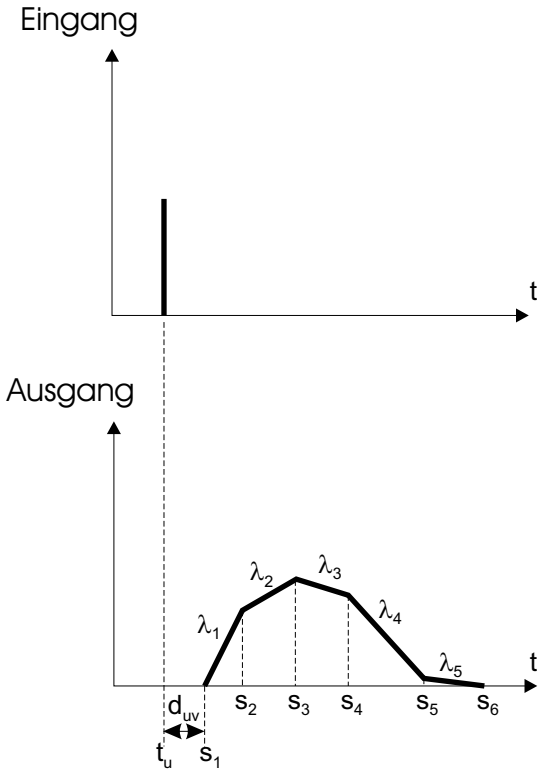


Figure 3.2: Synapse functioning: The synapse generates an excitatory post-synaptic potential (EPSP) after the receipt of a spike event.

Following the definition in section 3.1, the synapse response function $\varepsilon_{u,v}(t)$ can now be written as

$$\varepsilon_{u,v}(t) := \sum_{i=1}^{k_{u,v}-1} (\varepsilon^s_{u,v,i+1} - \varepsilon^s_{u,v,i}) \cdot \varepsilon^\lambda_{u,v,i} + (t - \varepsilon^s_{u,v,k_{u,v}}) \cdot \varepsilon^\lambda_{u,v,k_{u,v}}$$

and $\varepsilon_{u,v}(0) := 0$, whereas $k_{u,v} := \max_i(\varepsilon^s_{u,v,i} < t)$. Therefore the current gradient of $\varepsilon_{u,v}(t)$ is denoted by $\varepsilon^\lambda_{u,v,k_{u,v}}$. If, after each gradient change, the function value $\varepsilon_{u,v}(\varepsilon^s_{u,v,k_{u,v}})$ is calculated for the simulation time directly before this change, then the recursive form

$$\varepsilon_{u,v}(t) = \varepsilon_{u,v}(\varepsilon^s_{u,v,k_{u,v}}) + (t - \varepsilon^s_{u,v,k_{u,v}}) \cdot \varepsilon^\lambda_{u,v,k_{u,v}}$$

can be used.

The assumption that $\varepsilon_{u,v}(t) = 0$ for $t \in [0, d_{u,v}]$, i.e. the initial synaptic delay between the receipt of a spike from neuron u until the post-synaptic potential starts changing the potential of neuron v, can be modeled easily by choosing $\varepsilon^s_{u,v,1} = d_{u,v}$ ($\varepsilon^\lambda_{u,v,0}$, the gradient within the interval $[0, \varepsilon^s_{u,v,1}]$ is 0 by definition).

The potential of a neuron can now be defined as

$$P_v(t) := \sum_{\substack{u \in V \\ \langle u,v \rangle \in E}} \sum_{\substack{s_u \in F_u \\ s_u < t}} w_{u,v} \cdot \varepsilon_{u,v}(t - s_u)$$

with $P_v(0) := 0$, where V specifies the set of neurons, E the set of synapses and F_u the set of firing times of the neuron u. Therefore the potential $P_v(t)$ is defined as the linear sum of all incoming PSPs caused by spikes from the neurons u, whereas multiple PSPs from a single synapse $\langle u,v \rangle$, started at times $s_u \in F_u$, can be active (the response function $\varepsilon_{u,v}(t - s_u)$ has not reached 0 again) concurrently.
Inserting the definition of piecewise linear functions in $P_v(t)$ yields:

$$\begin{aligned} P_v(t) \quad := \quad & \sum_{\substack{u \in V \\ \langle u,v \rangle \in E}} \sum_{\substack{s_u \in F_u \\ s_u < t}} w_{u,v} \cdot \\ & \left(\sum_{i=1}^{k_{u,v}-1} (\varepsilon^s_{u,v,i+1} - \varepsilon^s_{u,v,i}) \cdot \varepsilon^\lambda_{u,v,i} + (t - s_u - \varepsilon^s_{u,v,k_{u,v}}) \cdot \varepsilon^\lambda_{u,v,k_{u,v}} \right) \end{aligned}$$

However, computing this formula directly each time an intersection with the threshold function has to be calculated would cause too much computational effort. Therefore, it is one of the key points where optimization is necessary. In this diploma thesis, the following optimization is proposed: P_v is kept as a state variable which is initially empty.

$$P_v(t) := \langle\rangle \text{ for } (\forall u)\ (u \in V, \langle u,v \rangle \in E)\ F_u = \emptyset$$

When a neuron of the input set of v, i.e. a Neuron u for $u \in V$ and $\langle u,v \rangle \in E$, fires, then the new response function $\varepsilon_{u,v}(t - s_u)$ (post-synaptic potential) generated by the synapse $\langle u,v \rangle$ is merged with P_v; merging is simply done by adding the response functions to the potential recursively:

$$P_v'(t) := P_v(t) + \varepsilon_{u,v}(t - s_u)$$

As these functions are both piecewise linear as defined in section 3.1, the sum of two piecewise linear functions needs to be calculated:

$$\begin{aligned} P_v' \quad &= \quad P_v \oplus \varepsilon_{u,v} \\ P_v \oplus \varepsilon_{u,v} \quad &:= \quad \Big\langle \langle s_i, \lambda_i \rangle \mid s_i \in P_v^s \cup \varepsilon^s_{u,v}, \\ & \qquad (\forall j,k)\ (0 < j < k < len(P_v \oplus \varepsilon_{u,v}))\ s_j < s_k, \\ & \qquad \lambda_i = \left.\frac{dP_v}{dt}\right|_{t \to s_i^+} + \left.\frac{d\varepsilon_{u,v}}{dt}\right|_{t \to s_i^+} \Big\rangle \end{aligned}$$

with $\frac{df}{dt}\Big| t \to s_i^+$ denoting the gradient of the function f at time s_i, but using the "right" gradient (with values infinitesimally larger than s_i) if f is not continuous at s_i. This merge process can be done with linear complexity for arbitrary previous states of P_v and response functions – assumed that both functions are valid according to the definition and constraint given in section 3.1; calculating the gradient values as

$$\lambda_i = P^{\lambda}_{v,j} + \varepsilon^{\lambda}_{u,v,k}$$

$$j = \max_l(P^s_{v,l} \leq s_i)$$

$$k = \max_l(\varepsilon^s_{u,v,l} \leq s_i)$$

in the above definition of the sum enables an efficient implementation. After the merge process, the new neuron potential is – in any case – again a piecewise linear function following all constraints given in section 3.1 (e.g. the values $P'^s_{v,i}$ are unique and strictly monotonic increasing and $P'_v(P'^s_{v,m+1}) := 0$).
Fig. 3.3 shows an example where a new response function $\varepsilon_{u,v}(t - s_u)$ is merged with the current potential function P_v, generating the new potential function P'_v.

Due to the special internal representation of and constraints on piecewise linear functions in the current implementation, it is possible to use an extended and very fast variant of the merge-sort algorithm for combining a newly received post-synaptic potential with the current neuron potential (see section 4.2) which completely eliminates the need to sum up over the neuron inputs as it has to be done in the standard integrate-and-fire model.

The threshold function $\Theta_v(t)$ is also defined as a piecewise linear function as

$$\Theta_v(t) := \begin{cases} \infty & if \quad \max_i(\Theta^s_{v,i}) \leq t < \max_i(\Theta^s_{v,i}) + \tau_{v,ref} \\ \Theta_{v,ref} & \\ -\sum_{i=1}^{l_v-1}(\Theta^s_{v,i+1} - \Theta^s_{v,i}) \cdot \Theta^{\lambda}_{v,i} & if \quad \max_i(\Theta^s_{v,i}) + \tau_{v,ref} \leq t \\ +(t - \Theta^s_{v,l}) \cdot \Theta^{\lambda}_{v,l} & \quad < \max_i(\Theta^s_{v,i}) + \tau_{v,ref} + \tau_{v,end} \\ \Theta_v(0) & otherwise \end{cases}$$

where $\Theta_v(0)$ is some constant value, $\Theta^s_{v,i}$ specify the firing times of neuron v, $\Theta_{v,ref}$ is the initial value of $\Theta_v(\max_i(\Theta^s_{v,i}) + \tau_{v,ref})$ after the absolute refraction period with $\Theta_v(t) = \infty$, l_v denotes the number of gradient changes and $\Theta^{\lambda}_{v,i}$ are the gradients within the intervals $[\Theta^s_{v,i}, \Theta^s_{v,i+1}]$ (see Fig. 3.4).
The goal of discrete event simulation is now to calculate the simulation times s_v (in Fig. 3.4 specified as t_v for better readability) at which the neuron v will fire.

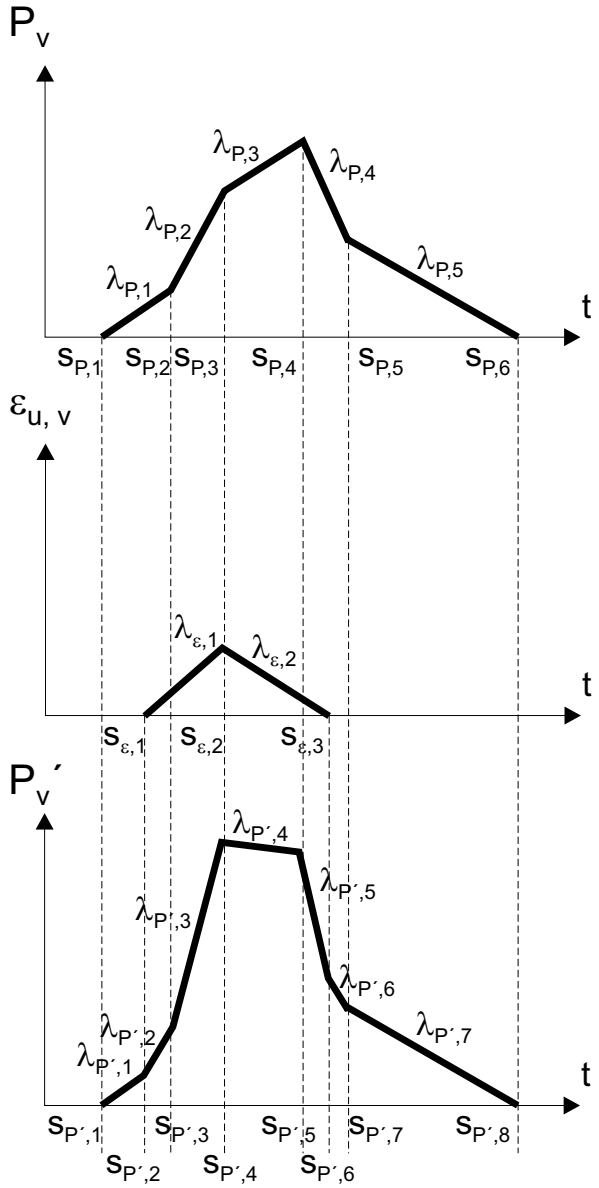


Figure 3.3: Merging a new post-synaptic potential with a neuron's potential function.

3.3 Calculation of firing times

To calculate the simulation time points when a neuron v will fire, it is necessary to calculate the intersections between the neuron's potential and threshold functions. As both the potential and threshold functions are piecewise linear, this calculation can be reduced to calculating the intersection between linear functions. A linear function is normally given as

$$y(t) = k \cdot (t - e) + d$$

For uniquely defining such a function, only one of the constants d and e would be

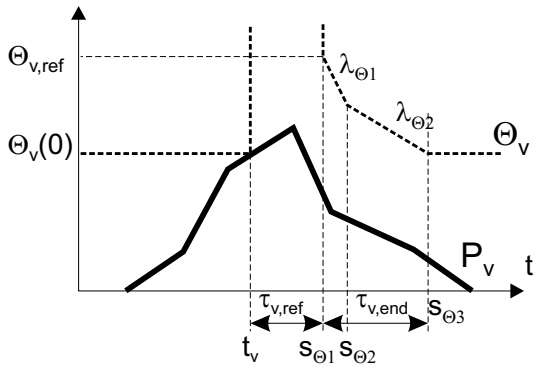


Figure 3.4: Neuron potential intersecting with the threshold function.

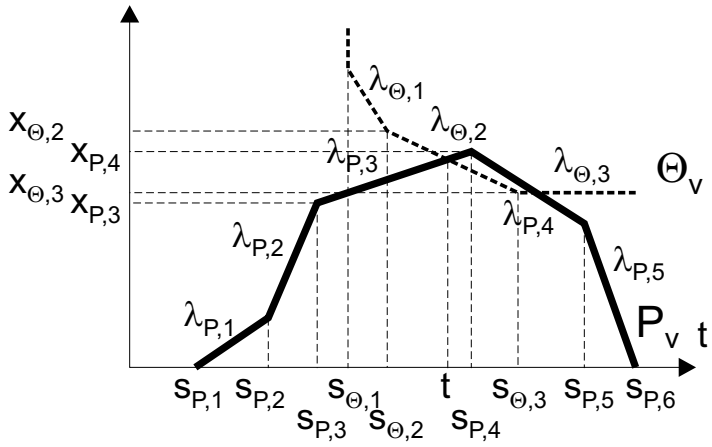


Figure 3.5: Calculating the intersection between a neuron's potential and threshold functions.

necessary. But for simplifying the application of these equations to the intersection algorithm, both are used. To calculate the intersection between two such functions y_1 and y_2, one can use

$$d_1 + k_1 \cdot (t - e_1) = d_2 + k_2 \cdot (t - e_2)$$

or, after making t explicit:

$$t = \frac{d_1 - d_2 - k_1 \cdot e_1 + k_2 \cdot e_2}{k_2 - k_1}$$

When calculating the intersection between two piecewise linear function, it is sufficient to calculate the intersections between each of the segments. The intersection of the piecewise linear functions is found when the calculated time t is inside the time-frame in which both segments are valid. An example of this can be seen in Fig. 3.5; in this example the intersection between the neuron potential P_v and the neuron threshold Θ_v occurs in segment 3 of P_v and segment 2 of Θ_v. Thus, when calculating the intersection time t between those segments with

$$t = \frac{x_{P,3} - x_{\Theta,2} - \lambda_{P,3} \cdot s_{P,3} + \lambda_{\Theta,2} \cdot s_{\Theta,2}}{\lambda_{\Theta,2} - \lambda_{P,3}}$$

the condition $x_{P,3} \leq t < x_{P,4} \wedge \lambda_{\Theta,2} \leq t < \lambda_{\Theta,3}$ is met. This means that the intersection time t is within the definition frame of the segments (the time interval in which both segments are valid) and therefore it is indeed an intersection between P_v and Θ_v.
When P_v and Θ_v are piecewise linear functions according to the definition in 3.1 (the important restriction is that the tuple vectors need to be sorted), then the intersection time can be calculated in linear time complexity.

3.4 Network inputs and outputs

For an SNN to prove useful in practical applications, there is a need for well-known coding schemes at the inputs and outputs of the network. The temporal spike coding scheme, transporting information within the neural network on the exact firing times of spikes, seems to offer many benefits in terms of computation power and compact coding of information. Nevertheless, at the moment there exist practically no sensors or actors utilizing this coding scheme, so it cannot be used for interacting with the real world. Therefore, input and output converters which transform the external information coding to spike trains at the SNN input and the spike trains to the external coding at the SNN output are needed. Effectively, this transforms the SNN into a black box whose inner operations do not have to be known.
Additionally, these converters offer more flexibility because the external and internal information coding schemes are decoupled. This enables the use of different spike coding schemes, such as temporal coding, rate coding, population coding or much more complex coding schemes (like fractal coding) without being forced to change the external representation of information. It also eases the use of different external information codings, e.g. when adding additional sensors or actors for interaction with the environment.

Currently, input and output converters between real numbered vectors outside the SNN and two inner coding schemes have been specified: temporal coding and rate coding. The third known coding scheme that is used in biological models, population coding (refer to subsection 2.2.1), is currently not specified in detail. The following definitions have, as far as applicable, been adapted mainly from [Ruf98] because the input and output converters are independent of the inner simulation technique; their only purpose is to convert between real numbered values and simulation times.

Temporal coding is defined as follows: The input neurons u fires at simulation times

$$s_u = T_{input} - d \cdot x_u$$

where x_u specifies the real numbered value that should be coded, T_{input} a reference time marking the input value zero, and d a common scaling factor. It has been shown that this coding scheme can be used to easily detect equality or near-equality of values (see [Maa97b]), simulate McCulloch-Pitts neurons (see [Maa99b]), compute radial basis functions (RBFs, see [Hop95]) and weighted sums (see [Ruf98]) as well as simulating sigmoidal neurons (also [Ruf98]) and approximating any continuous function (see [Ruf98], [Maa99b] and [Maa97a]).
Rate coding is defined by

$$\Delta s_u = \frac{1}{x_u}$$

The neuron u fires periodically with frequency x_u whereas x_u is the real numbered value that should be coded. By definition, the input neuron u starts to fire at time

$$s_{u,0} = \frac{1}{x_u}$$

for the first time and then fires at the recursively defined times

$$s_{u,i+1} = s_{u,i} + \Delta s_u.$$

This definition uniquely defines the firing times of an input neuron u for a constant input value x_u. It is not defined and dependent on the implementation how the exact firing times are calculated when the input value x_u changes during the simulation. Usually Δs_u will be modified immediately when x_u changes, but the next firing time $s_{u,i+1}$ will be kept – the next firing event will be executed as it has been scheduled. Only the firing time after the next, $s_{u,i+2}$, will be influenced by the changed firing rate as $s_{u,i+2} = s_{u+1} + \Delta s_u$.

The network output converters work accordingly by receiving spike events at their inputs and computing real numbered values for their outputs. Temporal coded output converters calculate the output value with

$$x_u = \frac{T_{input} - s_u}{d}$$

whereas the constant scaling factor d should have the same value as used in the input converter. If a reference spike at time T_{input} is available, marking the exact time point with real numbered value 0, then this output coding is the exact inversion of the input coding, allowing the network to use absolute input and output values. However, if such a reference value is not available, then the output converter must take the smallest value in the respective output cycle (the time of the spike that has been received latest) as T_{input}; this results in some constant error for all output values (scaled correctly but not having an absolute reference point). This problem is inherent to the coding scheme, which is based on relative times (time differences).

Rate coded output converters can simply use the last two received spikes to calculate the output value (the firing rate) with

$$x_u = \frac{1}{\Delta s_u} = \frac{1}{s_{u,i+1} - s_{u,i}}.$$

3.5 Learning

Learning the synaptic weight values in this model can be done as in every other implementation of an SNN simulation. E.g. in [Ruf98] a few methods for supervised and unsupervised learning in temporal coding are explained. One such learning rule for supervised learning can be conducted with only two spikes at each synapse $\langle u, v \rangle$: the pre-synaptic neuron u fires at time t_u and the post-synaptic neuron is forced to fire at time t_0 by additional PSPs (from auxiliary neurons), which is the time it should fire for the given input pattern. Then the synaptic weight can be changed according to

$$\Delta w_{u,v} = \eta \cdot (t_0 - t_u)$$

with a following normalization of the weight vector of neuron v after each learning cycle. However, since there are auxiliary neurons needed to make this learning possible, it is questionable whether there is any biological motivation for this method. SNNs naturally imply the use of unsupervised learning techniques due to their inherent event oriented nature.
Therefore, the simulations conducted for this diploma thesis do not use any form of supervised learning. Instead, a simulation of an SOM (Self Organizing Map, see [Koh95]) has been written – but the basic structure needed for arbitrary learning algorithms has been implemented, making the use of other learning methods easy (see section 4.4). A simple, unsupervised learning rule would be to apply

$$\Delta w_{u,v} = \eta \cdot (s_u - w_{u,v})$$

to the winner neuron v (the neuron that fires first in the competitive layer), where s_u is the firing time of the pre-synaptic neuron u in the current cycle and η is the current learning rate [Ruf98]. But to extend this method to also exhibit self-organizing behavior, neighborhood information must be introduced. One way to do so, also described in [Ruf98] and implemented for the simulation described in section 5.4, is the following, enhanced learning rule:

$$\Delta w_{u,v} = \eta \cdot \frac{T_{out} - t_v}{T_{out}} \cdot (s_u - w_{u,v})$$

where t_v is the firing time of the v-th competitive neuron (the post-synaptic neuron of the respective synapse) and T_{out} is the reference time marking the end of the learning

cycle. It is important that no competitive neuron will fire after T_{out} – every neuron that fires in the current learning cycle must fire before this time; this can be satisfied by setting T_{out} large enough. The rule is applied to all neurons in the competitive layer that have fired before time T_{out}. Then, the neighborhood function needed for self-organizing behavior is realized by the factor $(T_{out} - t_j)/T_{out}$. Furthermore, the learning rate η as well as the lateral weights $\widetilde{w}_{i,j}$ between the competitive neurons are slowly decreased, thus also decreasing the size of the neighborhood as the learning advances.
Another unsupervised learning rule is described in [NR98], which emulates RBF networks in temporal coding. However, for this diploma thesis no simulations about RBF networks were conducted.

Chapter 4

Framework

In order to be able to prove that the concepts proposed in this diploma thesis indeed work for practical simulations, a Java framework for DEVS simulation of SNNs has been developed and the core parts concerning the structure of the Neural Networks and handling of events including calculating firing times of neurons are complete. Using discrete event simulation, an enormous increase in simulation speed and flexibility is expected when compared to the currently used continuous simulation techniques. It should be possible to simulate large networks of spiking neurons and thus allowing studies of advanced networks holding a lot more information than networks with only a small number of neurons. Additionally, it is expected that the simulation speed-up gained by discrete event simulation will increase with the size of a network, because typically the percentage of concurrently active neurons will decrease as the number of neurons in the network increases. With this framework it will be possible to simulate and visualize Spiking Neural Networks with piecewise linear functions and to experiment effectively with new signal forms and learning methods in the context of computational theory as well as from the biological point of view. Although the use of other function types is possible in the core parts of the framework, the current implementation only covers piecewise linear ones (as defined in section 3.1), which seem to be sufficient for reproducing most computational effects of biological neural networks (refer to section 3.1). If, at some time, it turns out that other function types, e.g. polynomial functions or approaches based on B-splines [CS47, CS96] could be beneficial, adding these types to the framework can easily be carried out; the base classes of the framework have been defined on an abstract level that is independent of the used functions. Nevertheless, piecewise linear functions also allow an approximation of all function shapes as closely as needed or wanted, even if there are more efficient approximations for some function types.

In the following subsections, the simulation framework will be described. First of all, the core parts of the framework for the event handling in neurons and synapses is explained, followed by an overview of the additional parts providing added function-

ality. In general, the implementation will not be explained in great detail, because the Javadoc documentation has been generated for exactly this purpose. Instead, in this chapter mainly the features of the different components will be described, but elaborating the description for the core algorithms that actually implement the formal modal described in chapter 3. In the algorithm descriptions, merely the important parts will be shown, omitting debugging statements and maintenance code.
For details of the implementation such as method parameters, pre- and post-conditions and exceptions, the reader is referred to the Javadoc documentation which is contained on the CD-ROM (submitted accompanyingly).

4.1 Architecture

The simulation framework is based on MOSAIC (*MOdeling and Simulation by Assembling Interactive Components*), developed at the Institute for Systems Research at the Johannes Kepler University Linz. It is a framework for building discrete event simulation systems following the DEVS model and using an object oriented approach based on JavaBeans. This approach allows to build complex simulation systems hierarchically. Furthermore, JavaBeans define a clean application programming interface (API) for the use of the basic components. Because of the use of Java as implementation language, the whole simulation system is completely platform independent and can run on any computer which a Java virtual machine (JVM) is available for. During the development of this simulation framework, both Microsoft Windows 2000 with JVM 1.3.1 and JVM 1.4 as well as Debian GNU/Linux 3.0 with JVM 1.3.1 and JVM 1.4 have been used. Therefore, the system runs without any problems on these reference platforms. It will probably run on any platform with a JVM, but this has not been verified so far.

4.1.1 Requirements

Currently only a JVM version >= 1.2 is needed to compile and run the simulation, but this may change in the future due to the possible use of newer language features (such as assertions available since JVM 1.4 or templates planned for JVM 1.5).
Additionally, a few external Java libraries are needed for various parts of the simulation framework; but all of them are available for free:

- Apache Ant: Apache Ant is a Java-based build tool that enables to build complex software packages platform-independently. In this simulation framework, it is used for controlling the whole build process (explained in subsection 4.1.2).
  Required version: >= 1.4 including the additional tasks package
  Available from: http://jakarta.apache.org/ant/index.html

License: "Apache Software License, Version 1.1" (Open Source)
Ant is not distributed with the simulation framework because it is needed to build the distribution.

- JUnit: JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is used to conduct automatic regression tests of software components that can run automatically. In this simulation framework, it is used to test important parts automatically before creating the distribution files (explained in subsection 4.1.3).
  Required version: >= 3.7
  Available from: http://www.junit.org/index.htm
  License: "IBM Common Public License Version 0.5" (Open Source)
  JUnit is distributed with the simulation framework in the source tree under libraries/junit-3.7.jar

- JAI: The Java Advanced Imaging API provides a set of object-oriented interfaces that support a simple, high-level programming model which allows developers to manipulate images easily. In this simulation framework, it is only used for loading and storing images as network inputs and outputs (explained in section 4.3).
  Required version: >= 1.1.1
  Available from: http://java.sun.com/products/java-media/jai/index.html
  License: "Sun Microsystems, Inc. Binary Code License Agreement JAVA ADVANCED IMAGING API, VERSION 1.1.1" and "DEVELOPMENT TOOLS JAVA ADVANCED IMAGING, VERSION 1.1.1 SUPPLEMENTAL LICENSE TERMS"
  JAI is distributed with the simulation framework in the source tree under libraries/jai_core.jar and libraries/jai_coded.jar. Distribution in binary form is allowed by the "SUPPLEMENTAL LICENSE TERMS", paragraph 2.

4.1.2 Build process

Since this simulation framework is a complex software system consisting of many interacting components, the build process of the distribution files (source, documentation and binary) is not trivial. Therefore, to enable the system to be built easily, an automated build process has been created.
The build process uses Apache Ant as the build tool. This program parses the file *build.xml* to get the build instructions. As this is an XML file, it can be read with any ASCII editor. Normally, it is enough to call the Ant program from within the distribution source directory; it will automatically load the build file and execute the default task, `dist`, which in turn depends on the other needed tasks. The distribution source directory contains the following content:

[build/]
build.xml
concurrent/
[dist/]
libraries/
mosaic/
sim/

The directories *build/* and *dist/* are created during the automatic build process and are therefore not contained in the source distribution. As mentioned above, the file *build.xml* is the main build file for Ant, which contains all the instructions that are needed to create the documentation and binary distributions from the extracted source distribution. In the directory *libraries/* are only those Java libraries that are distributed with the simulation framework (described in subsection 4.1.1). The other directories, *concurrent/*, *mosaic/* and *sim/* contain the source code of the whole simulation framework, including MOSAIC.
When Ant is executed from within this source directory without further parameters, it automatically creates the directories *build/*, containing the intermediate build files, and *dist/*, containing the created source, documentation and binary distribution files. This behavior is defined by the various build tasks in *build.xml*, which are listed in table 4.1.

4.1.3 Automatic component testing

Due to the support of the JUnit regression testing framework, building new automatic component tests is simple. A new test class only needs to be derived from `junit.framework.TestCase`, which is demonstrated by the implemented `ImageConverterTest` class. Currently the framework is set up to contain unit test classes in the package `mosaic.sim.neuron.tests`, correspondingly in the source directory *mosaic/sim/neuron/tests*. Although at this stage, the `ImageConverterTest` class is the only one that has been implemented, other ones will be added in the future, with tests for the core algorithms (like merge and intersection of piecewise linear functions, see section 4.2 for an explanation) being added next.
The main build file *build.xml* contains the build task `test` to start the automatic regression tests. Newly added tests will automatically be included in the testing process, they only need to be put into the mentioned directory.

4.1.4 Basic structure

The most basic structure of the whole simulation framework is a bipartite graph formed of `Neuron` and `Synapse` objects. All of the simulation dynamics builds upon this topology, which is illustrated in Fig. 4.1.

Task name	Depends on	Description
init		Creates an internal time stamp and the directory *build/*.
compile	init	Compiles the whole Java source from the current directory (contained in the directories *concurrent/*, *mosaic/* and *sim/*) and puts the compiled class files into *build/*.
test	compile	Starts the automatic regression tests on the compiled classes and stops the build process when an error occurs.
dist	compile	Creates the directory *dist/*, generates the JavaDoc HTML documentation and generates the files *dist/DEVSNeuron-doc-<date>.zip*, *dist/DEVSNeuron-src-<date>.zip* and *dist/DEVSNeuron-bin-<date>.jar*. The file *DEVSNeuron-doc-<timestamp>.zip* contains the JavaDoc HTML documentation for the simulation framework, the file *DEVSNeuron-src-<date>.zip* contains the complete source distribution with all files that are needed to build itself and the file *DEVSNeuron-bin-<date>.jar* contains all compiled class and auxiliary files needed to run the simulations (only the binary distribution file is needed to run the simulations or develop additional simulations).
clean		Removes the directories *build/* and *dist/*.

Table 4.1: Build tasks in the automated framework building process.

Following the methodology of MOSAIC, the core parts are active elements which directly or indirectly use the MOSAIC base class `Timer`. This class enables components to become active in the simulation by triggering time events; moreover, it allows the simple addition of visualization elements based on the Java Swing technology. In the present simulation framework, three types of events are used:

- Time events: These events are triggered when a certain simulation time is reached. Setting such an event to be executed at some defined, *future* simulation time is called *scheduling*. As these event types are already provided by the MOSAIC framework, its functionality is used. Whenever a time event is to be scheduled for activation, a method called `activateAt(double absoluteTime)` or `activateIn(double relativeTime)` is called (see algorithm 1 for an example). These two methods are implemented in `Timer`, sup-

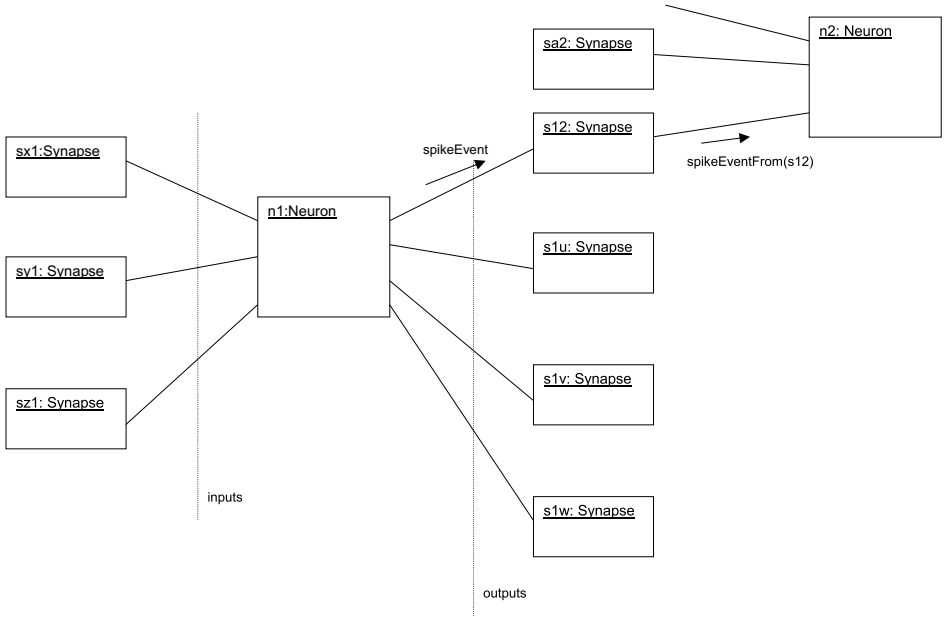


Figure 4.1: The general structure of the simulation framework is a bipartite graph of `Neuron` and `Synapse` objects.

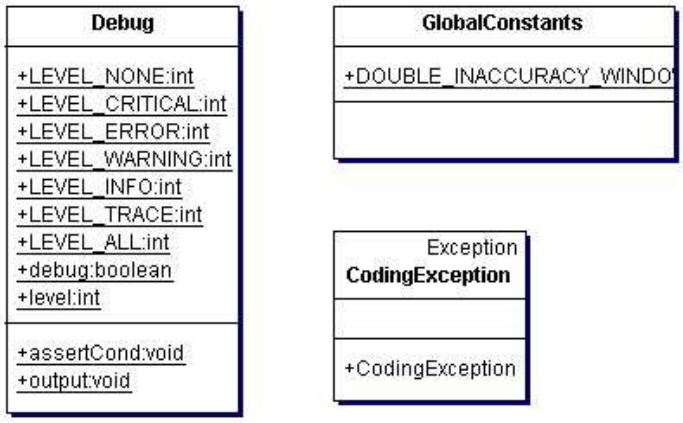


Figure 4.2: Auxiliary classes in the core framework.

ported by `BasicActiveModel` as well as `ActiveVariable` (refer to section 4.2) and are therefore available in each derived class.

- Internal events: These events are mostly subsequent events which are caused by time events or state change events. In the implementation, internal events are simple method calls that are usually done for a list of similar objects. One of the most important events in the simulation framework is the spike event, which is sent by `Neuron` objects to a list of `Synapse` objects; this event is an internal

event (refer to algorithm 2), but is caused by a time event which activates that `Neuron` object.

- State change events: These events are triggered when certain system states change their value. Sending and listening for such events is also supported by MOSAIC, the created simulation framework is using this support. Although state change events are mostly used for the visualization (by visualization components listening to state changes of those components they represent), they are also used for triggering the recalculation of firing times after the potential or threshold functions of a `Neuron` object have changed.

The use of these events in the various parts of the simulation will be described in more detail in the following sections. However, there are a few global auxiliary classes that are used in almost all following parts:

GlobalConstants This class defines some constants that are global parameters for the simulation framework, e.g. the inaccuracy window used for comparing two floating point numbers. (Due to rounding errors, it is not reasonable to compare floating point numbers for equality. Instead, the absolute difference has to be smaller than some small value ε, called inaccuracy window in this diploma thesis.)

CodingException This class represents an exception in the used spike coding scheme, which can be e.g. temporal or rate coding. Whenever a situation is detected that is not allowed within the definition of the used coding scheme, this exception is thrown by the input or output converters.

Debug This class is used for debugging purposes, mainly for printing debug messages dependent on a trace level and for assertions (although they are now available as a language construct with a JVM >= 1.4, but the assertions provided by this class are more powerful in logging error messages).

4.2 Neurons and Synapses

The core framework mainly consists of the classes `Neuron` and `Synapse`, which are in the center of the simulation. As can be seen in Fig. 4.3, the connection between `Neuron` and `Synapse` objects lies within the interfaces `NeuronInput` and `NeuronOutput`, which will be described below. Now, the main idea is that `Neuron` objects fire spike events to `Synapse` objects by calling the respective method; therefore those spike events are internal events. Upon receiving a spike event, the `Synapse` objects will then in turn invoke a method of their post-synaptic `Neuron` object to state that an event

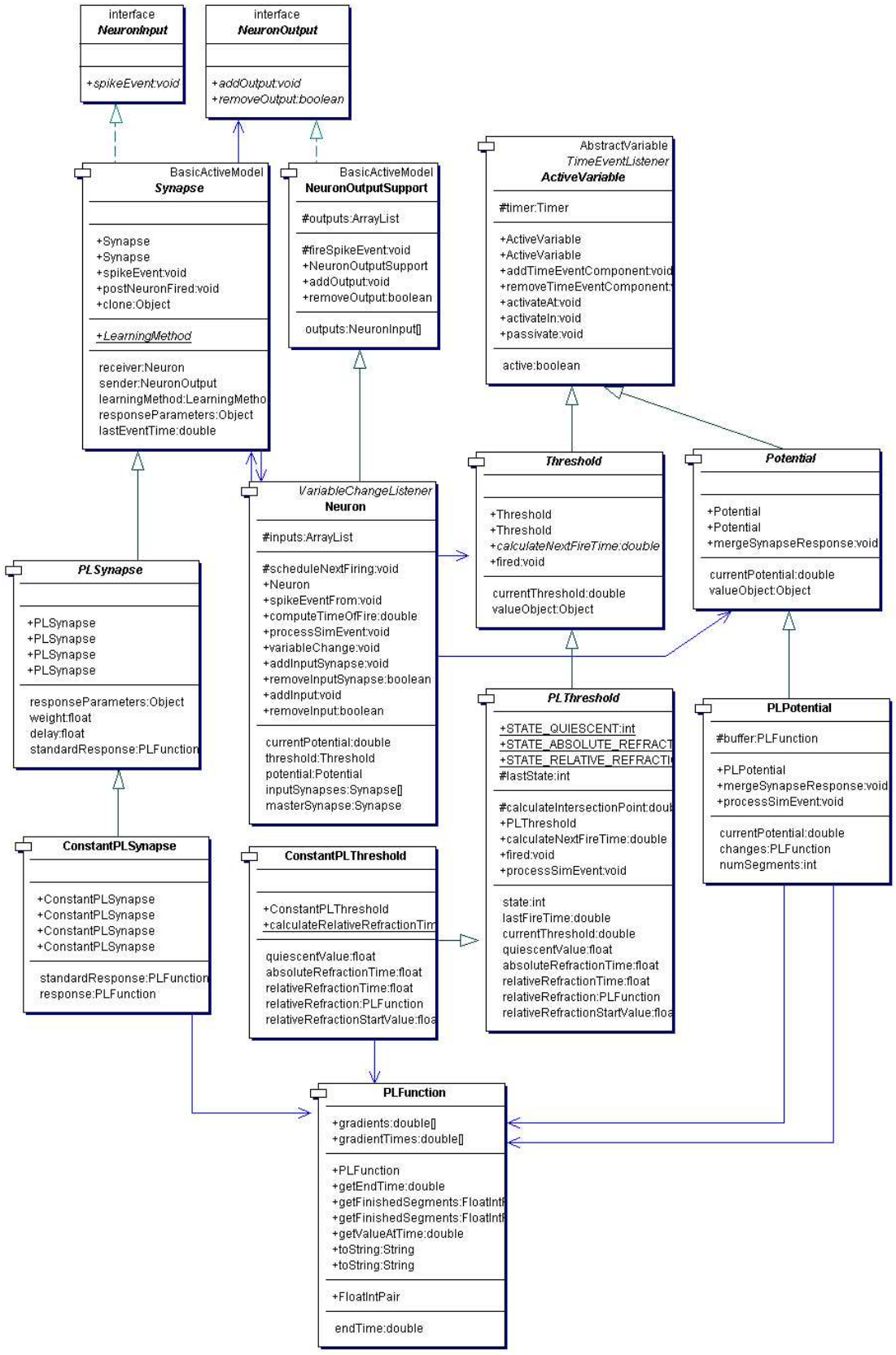


Figure 4.3: Main classes for event handling in neurons and synapses.

has been received. After this, the `Neuron` object will start handling the incoming firing event (see algorithm 1) which involves the calling of another method of the input `Synapse` object to get the parameters of the generated synapse response function. Due to this incoming event, the `Neuron` object might then generate a time event for its own activation, scheduled for the time when the next firing event is due. As the time event is then triggered at some future simulation time, the `Neuron` object will get reactivated and will in turn emit a spike event (see algorithm 2).
In the following, the interfaces and classes that implement this concept in the core parts of the simulation framework are explained. It is important to note that the process itself is completely independent of the used function types, be them piecewise linear, spline based or continuous – the general idea is merely to use an event based approach.

NeuronInput This interface specifies the capability of an object to act like an input of a neuron, i.e. the ability to receive spike events. It will usually be directly connected to an object with the capability of sending spike events, i.e. an object that implements the `NeuronOutput` interface. The only method that is defined by this interface and therefore must be implemented by objects which need to receive spike events is

```
public void spikeEvent(NeuronOutput sender)
```

Therefore, a spike event does not carry any other information than the sender object, which is biologically motivated.

NeuronOutput This interface specifies the capability of an object to act like the output part of a neuron, i.e. the ability to send spike events. Objects implementing the `NeuronInput` interface will register to this event source and all objects implementing `NeuronOutput` must call the `spikeEvent()` method of all currently registered objects whenever the object emits a spike event. This interface defines the usual `addOutput` and `removeOutput` methods as used by Java event handling.

NeuronOutputSupport This is a helper class for objects that want to emit spike events. It implements everything required by the `NeuronOutput` interface and also offers the method `fireSpikeEvent` for sending a spike event to all registered `NeuronInput` listeners. Because not only `Neuron` objects will emit spike events, but typically also input converters (see section 4.3) and other "sensory" classes, the functionality of sending a spike event to a list of registered listeners has been abstracted into this class.

Neuron This is the core class of the framework, administrating the handling of spike events and holding the `Potential` and `Threshold` objects. It represents a neuron, which is able to fire spike events (thus it implements the `NeuronOutput` interface) and receive spikes indirectly via `Synapse` objects. For a simple handling of the input and output `Synapse` objects, this class uses `ArrayList` members, which can easily be manipulated using the respective member functions. Additionally, more sophisticated methods for manipulating inputs and outputs are available, such as the method `addInput` which adds a given `NeuronOutput` object to the list of inputs; to accomplish this, a new `Synapse` object is generated from a template object and the bi-directional connections on both sides of the `Synapse` object (the given pre-synaptic `NeuronOutput` object and the current post-synaptic `Neuron` object) are set up. Thus, this single method call will generate four new links to completely connect the bi-directional, bipartite graph formed of `Neuron` and `Synapse` objects.

Furthermore, each `Neuron` object contains a `Potential` and a `Threshold` object for maintaining its internal state as defined in the formal model in section 3.2. The `Neuron` object itself does not have any other state information than the input and output object lists; all of the calculation according to the formal model is done in the embedded `Potential` and `Threshold` objects. However, it is responsible for administrating the flow of spike events through the simulation framework, as shown in algorithms 1 and 2.

Note that the `Neuron` object itself is completely independent of the function types.

Synapse This class represents synapses, which are the connections between neurons in the biological model. When a `Synapse` object – which implements the `NeuronInput` interface – receives a spike event from any `NeuronOutput` capable object, it will forward it to the connected "*post-synaptic*", i.e. the *receiver* `Neuron` object (see algorithm 3), which will then query the `Synapse` for the response parameters. These parameters define how the neuron potential changes because of the incoming spike. This class is declared abstract because it does not define any response function type, but those function types must be defined by using a derived class and overriding the method `getResponseParameters`. However, `Synapse` already implements the maintenance code needed to connect with a *sender* `NeuronOutput` object (which will mostly be a `Neuron` object) and a *receiver* `Neuron` object. Furthermore, it defines a framework for implementing learning methods, which is described in section 4.4.

PLFunction This class represents a piecewise linear function in a finite time-frame with a finite number of linear segments as defined in section 3.1. It stores

This method is called by the input `Synapse` object after receiving a spike event – thus it is the entry point in the Neuron object for handling internal events.

```
public void spikeEventFrom(Synapse sy) {
    potential.mergeSynapseResponse(sy.getResponseParameters());
    scheduleNextFiring();
}
```

As can be seen easily, the `Neuron` object just passes the synapse response parameters (that are obtained from the `Synapse` object which sent the event) to its `Potential` object. This `Potential` object is then responsible for updating its state (see algorithm 5). The method `scheduleNextFiring` then uses the newly calculated future `Potential` state to calculate and schedule the next time when this `Neuron` object will fire.

```
protected void scheduleNextFiring() {
    double dt = computeTimeOfFire();
    if (dt == Double.POSITIVE_INFINITY)
        passivate();
    else
        activateAt(dt);
}
```

In this method, the real calculation is again delegated to another object, in this case the `Threshold` object by calling the method `computeTimeOfFire` which only calls the respective method of the `Threshold` object (see below). After that, two cases are distinguished. Either the calculation resulted in a next firing time (in which case the return value is non-infinite) and the respective time event for activating the `Neuron` object is scheduled, or there is no next firing time with the current `Potential` and `Threshold` states (the return value is infinite) and the time event is canceled.

```
public double computeTimeOfFire() {
    return threshold.calculateNextFireTime(potential);
}
```

Then the `Neuron` object has completed handling the incoming event at the current simulation time.

Algorithm 1: Handling an incoming event in a `Neuron` object.

these segments in simple arrays of `double` values. The linear segments are described by their gradients and their starting times and the whole function is defined to start at value 0 and time 0 with gradient 0. All `gradients[i]` are valid between the times `gradientTimes[i]` and `gradientTimes[i+1]`. Therefore, the whole function has a value of 0 in the interval $[0, \texttt{gradientTimes}[0]]$. The last segment, starting at time `gradientTimes[numGradients]` with `gradient[numGradients]` is defined to end at the time when the function reaches the value 0. This time can be retrieved by using the method `getEndTime()`. Additional methods are provided for operations on piecewise linear functions, such as `getFinishedSegments` and `getValueAtTime`.

PLSynapse This class is derived from `Synapse`, implementing synapses with piecewise linear functions. This class is still abstract because the shape of the piece-

When a time event has been scheduled by `scheduleNextFiringTime`, the `Neuron` object will get reactivated at the scheduled simulation time by the MOSAIC framework. This reactivation is done by calling the method `processSimEvent`, which is then responsible for generating a firing event.

```
public void processSimEvent(SimEvent e) {
    fireSpikeEvent();
    threshold.fired();
    for (int i=0; i<inputs.size(); i++)
        ((Synapse) inputs.get(i)).postNeuronFired();
    scheduleNextFiring();
}
```

First of all, the spike event is fired by using the respective method of the base class `NeuronOutputSupport`. Then, the `Threshold` object is notified of the firing as it may change its state due to this (this notification is also an internal event), followed by the input `Synapse` objects also being notified (this is important for learning algorithms as explained in section 4.4). In a final step, a possibly new time event is scheduled as the `Potential` and `Threshold` states might already generate another spike in the future (if no spike is to be emitted, `scheduleNextFiring` will not schedule a time event).

Algorithm 2: Generating a spike event in a `Neuron` object.

When a `Synapse` object receives a spike event (due to implementing the `NeuronInput` interface), it will forward this event to its associated *receiver* `Neuron` object.

```
public void spikeEvent(NeuronOutput sender) {
    lastEventTime = SimApplication.getCurrentSimApplication().getTime();
    receiver.spikeEventFrom(this);
}
```

After the time of the incoming spike event is recorded, which is important for learning algorithms, the internal event is simply forwarded to the associated receiver `Neuron` object.

Algorithm 3: Forwarding a spike event in a `Synapse` object.

wise linear function is not defined, but queried using the abstract function

```
protected PLFunction getStandardResponse()
```

But those synaptic parameters that are usually used at each synapse, the weight and delay, are directly implemented in this class; they are applied to the piecewise linear function returned by `getStandardResponse` before returning it in the overridden method `getResponseParameters` (see algorithm 4).

ConstantPLSynapse This class represents a synapse using piecewise linear functions, but with a constant shape for all synapse objects in the simulation. It merely overloads the method `getStandardReponse` from `PLSynapse` to return a piecewise linear function that is kept as a static member in `ConstantPLSynapse`.

The method `getResponseParameters` is responsible for returning the response function that a `Synapse` generates in response to a spike event.

```
public Object getResponseParameters() {
    PLFunction standardResponse = getStandardResponse();
    PLFunction response = new PLFunction(standardResponse.gradients.length);
    for (int i=0; i<response.gradients.length; i++)
        response.gradients[i] = standardResponse.gradients[i] * weight;
    for (int i=0; i<response.gradients.length; i++)
        response.gradientTimes[i] = standardResponse.gradientTimes[i] +
          delay;
    return response;
}
```

First of all the standard shape of the response function is retrieved from the abstract method `getStandardResponse`. Then this response is simply scaled by the `weight` parameter and shifted by the `delay` parameter before returning it.

Algorithm 4: Calculating the current synapse response function from synaptic parameters.

ActiveVariable This class implements an active variable that has support for sending out `VariableChangeEvent`s and for sending itself a `TimeEvent`, therefore being an active variable container. Active variables are able to generate the third type of event: the state change events. This class has been implemented because `BasicActiveModel` does not support sending `VariableChangeEvent`s.

Potential This class represent the potential of a neuron. It is abstract and does not implement any behavior, but just defines the interface that the `Neuron` class uses to interact with its associated potential. The only methods defined by this class and needed to be implemented in overriding classes are

```
public double getCurrentPotential()
public void mergeSynapseResponse(Object change)
```

(see algorithm 1).

PLPotential This class represents a neuron potential described by a piecewise linear function. It does for the `Potential` object what `PLSynapse` does for `Synapse`: implementing a specific behavior of the abstract functionality. To accomplish this, `PLPotential` has a state member variable `changes`, which is a `PLFunction` and contains the currently known potential function (mostly future segments, but might also have some past segments). Although currently the arrays for the used `PLFunction` objects are recreated dynamically on any change, the class is fully prepared for using fixed-sized arrays in case of performance problems with dynamic memory allocation. The real functionality of this class is inside the method `mergeSynapseResponse`, whose primary working is described in algorithm 5, which is an implementation of the definition in

section 3.2. Additionally, the `processSimEvent` method is used for cleanup to prevent unbounded growth of the `PLFunction` kept in `changes`.

```
public void mergeSynapseResponse(Object change) {
    PLFunction newChange = (PLFunction) change;
    int i = 0, j = 0, k = 0;
    boolean changesActive = false, newChangeActive = false;
    buffer = new PLFunction(numSegments + newChange.gradients.length);
    while (i < numSegments || j < newChange.gradients.length) {
        if (j >= newChange.gradients.length || (i < numSegments &&
          (newChange.gradientTimes[j] + getTime()) -
          changes.gradientTimes[i] >
          GlobalConstants.DOUBLE_INACCURACY_WINDOW)) {
            changesActive = true;
            buffer.gradients[k] = changes.gradients[i] +
              (newChangeActive ? newChange.gradients[j-1] : 0);
            buffer.gradientTimes[k] = changes.gradientTimes[i];
            i++;
        }
        else if (i >= numSegments || (j < newChange.gradients.length &&
          changes.gradientTimes[i] -
          (newChange.gradientTimes[j] + getTime()) >
          GlobalConstants.DOUBLE_INACCURACY_WINDOW)) {
            newChangeActive = true;
            buffer.gradients[k] = newChange.gradients[j] +
              (changesActive ? changes.gradients[i-1] : 0);
            buffer.gradientTimes[k] = newChange.gradientTimes[j] + getTime();
            j++;
        }
        else {
            changesActive = newChangeActive = true;
            buffer.gradients[k] = changes.gradients[i] +
              newChange.gradients[j];
            buffer.gradientTimes[k] = changes.gradientTimes[i];
            i++; j++;
        }
        k++;
    }
    numSegments = k;
    PLFunction swap = buffer;
    buffer = changes;
    changes = swap;

    signalVariableChange();
}
```

After the definition of needed local variables and some maintenance code not shown here, the variant of the merge-sort algorithm immediately begins. The first `if`-branch is entered whenever a segment of the old `changes` is to be copied to the new changes, either because there are no more `newChange` segments or this segment starts before (concerning simulation time) the next segment from `newChange`. The second `if`-branch is entered whenever a segment from `newChange` is to be copied. The last `if`-branch is entered whenever the segments in `changes` and `newChange` begin at (nearly, as defined by the inaccuracy window) the same time. As can be seen easily, this method has a **linear time complexity**.
After this sorting has been completed, the newly created `buffer` function is swapped to be available as member variable `changes` and a state change event is sent to all listeners (e.g. visualization objects). For a detailed description of all conditions the reader is referred to the source code.

Algorithm 5: Merging a synapse response with the current neuron potential.

Threshold This class represents a Neuron threshold. It is abstract and does not implement any behavior, but just defines the interface that the Neuron object uses to interact with its associated threshold. The only abstract methods defined by this class and needed to be implemented in overriding classes are

```
public double getCurrentThreshold()
public double calculateNextFireTime(Potential
potential)
```

(see algorithm 1), but `fired` (see algorithm 2) should also be overridden to gain control over the behavior when the associated `Neuron` object has fired.

PLThreshold This class implements a threshold with piecewise linear functions, but is still declared abstract because the exact shape of the threshold in the relative refraction phase is not defined. However, it implements the calculation of the next firing time by intersecting a piecewise linear potential and a piecewise linear threshold function. The implemented method `calculateNextFireTime` uses a state member variable with three different states: quiescent, absolute refraction and relative refraction. Normally the threshold is in the quiescent state in which it has a constant threshold value. Upon invocation of the method `fired` (which is called by the associated Neuron object when it emits a spike event, see algorithm 2), the absolute refraction state is entered in which no firing is possible (`calculateNextFireTime` always returns an infinite value). After some time, the relative refraction state is entered, in which the threshold is a piecewise linear function that ends after some more time, returning to the quiescent state; these state transitions are carried out using time events and the method `processSimEvent`.

In algorithm 6 the calculation of an intersection between arbitrary piecewise linear functions, as defined in section 3.3, is shown. The implementation of `calculateNextFireTime` merely uses this method in the quiescent (with a piecewise linear function with one segment) and relative refraction states – the other code in `calculateNextFireTime` is dedicated to maintenance.

4.3 Network inputs and outputs

In the current simulation framework, input and output converters for the temporal coding scheme and an input converter for the rate coding scheme have been implemented. To enable an abstraction of the coder and decoder, interfaces for vector and matrix input and output have also been defined. In the following, the network input classes will be described.

The method `calculateIntersectionPoint` calculates the intersection between two arbitrary piecewise linear functions.

```
protected double calculateIntersectionPoint(
   PLFunction f1, double startValue1, double startTime1, int numSegments1,
   PLFunction f2, double startValue2, double startTime2, int numSegments2) {
     int i1, i2;
     double x1 = startValue1, x2 = startValue2, k1, k2, t1, t2,
       t, fireTime = 0;
     boolean foundIntersectionWithin = false, foundIntersectionBefore = false;
     PLFunction.FloatIntPair ret1 = f1.getFinishedSegments(
       startValue1, numSegments1, getTime() - startTime1, false);
     PLFunction.FloatIntPair ret2 = f2.getFinishedSegments(
       startValue2, numSegments2, getTime() - startTime2, false);
     i1 = ret1.i; x1 = ret1.f;
     i2 = ret2.i; x2 = ret2.f;
     while (! foundIntersectionWithin &&
       i1 < numSegments1-1 && i2 < numSegments2-1) {
         k1 = f1.gradients[i1]; k2 = f2.gradients[i2];
         t1 = f1.gradientTimes[i1] + startTime1;
         t2 = f2.gradientTimes[i2] + startTime2;
         if (k1 != k2) {
             t = (x1 - x2 - k1*t1 + k2*t2) / (k2 - k1);
             if (t < t1 || t < t2)
                 foundIntersectionBefore = true;
             else if (t <= f1.gradientTimes[i1+1] ||
               t <= f2.gradientTimes[i2+1]) {
                 fireTime = t;
                 foundIntersectionWithin = true;
             }
         }
         if (f1.gradientTimes[i1+1] + startTime1 >
           f2.gradientTimes[i2+1] + startTime2)
             i2++;
         else if (f1.gradientTimes[i1+1] + startTime1 <
           f2.gradientTimes[i2+1] + startTime2)
             i1++;
         else { i1++; i2++; }
     }
     if (foundIntersectionWithin)
         return fireTime;
     else {
         if (foundIntersectionBefore)
             return Double.NEGATIVE_INFINITY;
         else
             return Double.POSITIVE_INFINITY;
     }
}
```

After skipping segments that are entirely in the past, this algorithm principally just tries to calculate an intersection time between each of the segments in `f1` and each of the segments in `f2`. However, since both functions are sorted by definition, it is enough to advance the segments in both functions instead of looping over them in nested loops – therefore this algorithm also features **linear time complexity**. For those segments that are intersected, the exact definition from section 3.3 is used.

After calculating the intersection time of two segments, there are just 3 possibilities: either the intersection time is before the scope of one (or both) of the segments, it is within the scope of both segments or it is after the scope of one (or both) of the segments. The first case is a special one: when no real intersection is found, then the method return negative infinity, indicating that the given functions have an intersection before the current simulation time. The second case is the "normal" case where there is an intersection between two future segments, i.e. an intersection between the given functions within their scope. If no intersection (either case 1 or case 2) is found until the end of both given functions, positive infinity is returned.

Algorithm 6: Calculating the intersection between piecewise linear functions.

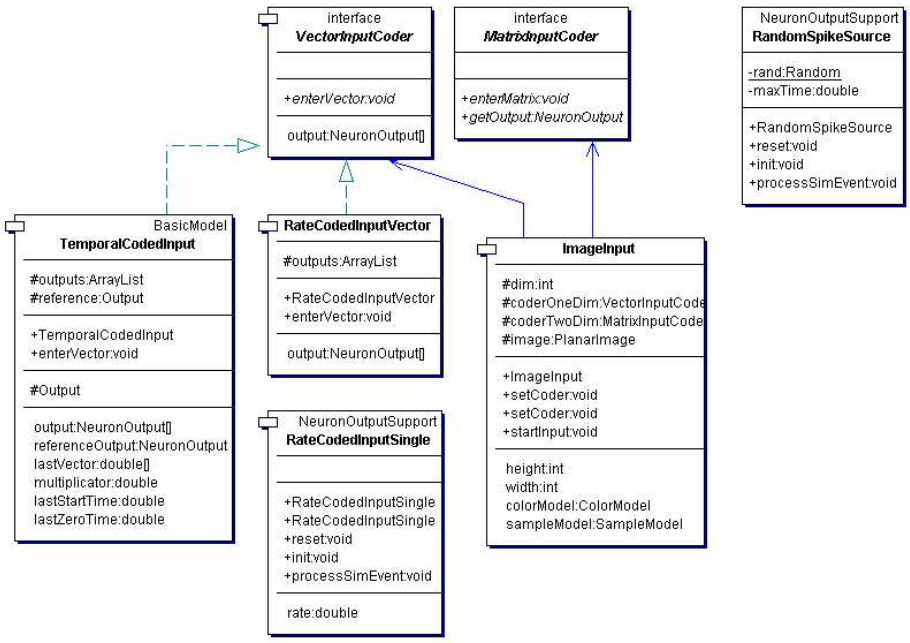


Figure 4.4: Network input converter classes.

VectorInputCoder This is an interface for delivering a one-dimensional vector of real numbered values to the network. It must be implemented by a class with a defined spike coding scheme (the so-called coder). It only defines the methods

```
public void enterVector(double[] vector)
public NeuronOutput getOutput(int index)
```

The method `enterVector` is designated for use from "outside" of the network to send real-numbered vector input events to the network, while `getOutput` can be used from "inside" of the network to obtain access to the spike event outputs of the coder.

MatrixInputCoder This is an interface for delivering a two-dimensional matrix of real numbered values to the network. It must be implemented by a class with a defined spike coding scheme (the so-called coder). It only defines the methods

```
public void enterMatrix(double[][] matrix)
public NeuronOutput getOutput(int dim1, int dim2)
```

As with `VectorInputCoder`, `enterMatrix` is for the "outside", while `getOutput` is for the "inside" of the network.

TemporalCodedInput This class implements a spike source using temporal coding by implementing the interface `VectorInputCoder`. It is a converter between analog values at its input and spike events at its output. The coding is done as follows: First of all, the number of inputs (i.e. the length of the input vector) must be equal to the number of outputs (inner `NeuronOutputSupport` classes). Thus, the analog vector submitted to the class via the method `enterVector` is converted to exactly one spike per output. The higher the analog input number, the earlier the spike on the associated output will be fired. Therefore the output associated to the analog input with the highest value will fire immediately, the other outputs will fire later, determined by the formula defined in section 3.4 (see algorithm 7).

This class can optionally use a reference output that fires at the latest time, thus representing the analog input value 0 and marking the end of the spike series emitted by this class.

The implementation of `enterVector` in `TemporalCodedInput` computes single spike events for each output depending on the given real numbered vector.

```
public void enterVector(double[] vector) throws CodingException {
    double maxValue=0;
    for (int i=0; i<vector.length; i++) {
       if (vector[i] > maxValue)
           maxValue = vector[i];
    }
    if (reference != null)
        reference.fireIn(multiplicator * maxValue);
    for (int i=0; i<vector.length; i++) {
        if (vector[i] != Double.NEGATIVE_INFINITY)
            ((Output) outputs.get(i)).fireIn(multiplicator *
              (maxValue - vector[i]));
    }
}
```

As defined in section 3.4, the spike firing times depend on the maximum value that is present in the input vector. After finding the maximum, the spike times are computed according to that formula and the firing of the respective output spikes is scheduled. Furthermore, if a reference output is used, it is scheduled to fire at the time that marks an input value of 0.

Algorithm 7: Temporal coding of an input vector.

RateCodedInputSingle This class implements a single spike source using rate coding. It is a converter between a real numbered value at its input and spike events at its output. The coding is done as follows: the method `getRate` and `setRate` can be used to set the firing rate of the current input, which continuously generates spike events with the computed interval (see algorithm 8). The rate is computed as defined in section 3.4.

RateCodedInputVector This class implements a spike source using rate coding. It

Firing spike events continuously can be done by using time events and regenerating new time events when they are triggered. Therefore the method `processSimEvent`, which is called when time events are triggered, is responsible for sending rate coded spike events.

```
public void processSimEvent(SimEvent e) {
    fireSpikeEvent();
    if (rate != 0.0) {
        double nextTime = 1 / rate;
        activateIn(nextTime);
    }
}
```

Each time a time event is triggered, the according spike event is immediately fired, followed by scheduling a new time event (dependent on the currently set firing rate) for firing the next spike event.

Algorithm 8: Rate coding of an input vector.

does for a vector what `RateCodedInputSingle` does for a single real numbered value. In fact, it uses `RateCodedInputSingle` objects for doing the real computation when receiving an input vector. This class implements `VectorInput`, as does `TemporalCodedInput`. Therefore, these coding schemes should be easily interchangeable.

ImageInput This class reads an image and converts it to either a one-dimensional vector of dimension $height \cdot width$ or a two-dimensional matrix of dimension $(height, width)$. For feeding a one-dimensional vector to the network, this class uses the interface `VectorInput`; for feeding a two-dimensional matrix, it uses `MatrixInput`. Objects implementing these interfaces have to be given to the constructor when initializing. To start the real input, reading the image and generating the vector or matrix, the method `startInput` needs to be called.
This class needs the JAI (Java Advanced Imaging) library.

RandomSpikeSource This class implements a random spike source with a configurable maximum interval between two emitted spikes. After being initialized with `init`, it just fires spike events at random intervals $[0, maxTime[$. Therefore it can be used for conducting experiments on noise stability.

In order to be able to interpret the outputs of an SNN, output converters need to be implemented; these converters clearly depend on the spike coding scheme used inside the network and the desired output coding. To make them as flexible as input converters, interfaces have also been defined for abstracting from the spike coding scheme. In the following, the currently implemented output converters will be described.

VectorOutputListener This interface must be implemented by objects that want to receive the real numbered, one-dimensional vectors computed by the network.

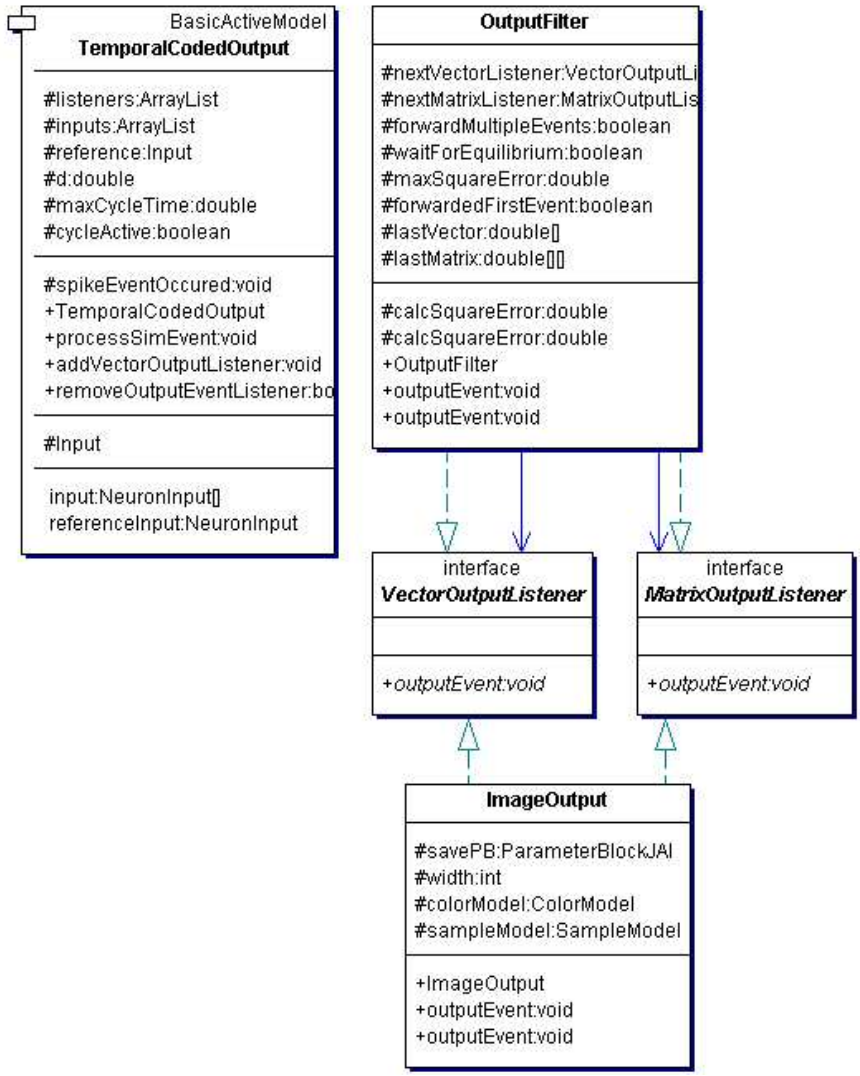


Figure 4.5: Network output converter classes.

The object receives an event whenever an output cycle is complete and the real numbered output values have been computed; these output events are generated by some spike decoder. The only method defined by this interface is

```
public void outputEvent(double[] vector)
```

MatrixOutputListener This interface must be implemented by objects that want to receive the real numbered, two-dimensional matrices computed by the network. As in `VectorOutputListener`, the only defined method is

```
public void outputEvent(double[][] matrix)
```

TemporalCodedOutput This class is a converter between spike events at its inputs and real numbered vectors – using `VectorOutputListener` objects – at its output. The spike events have to be coded temporally, e.g. from a `TemporalCodedInput` object, and are received by objects of an anonymous inner class implementing `NeuronInput`. These objects will, upon receipt of a spike event, record the reception time and notify the `TemporalCodedOuput` object. When the output cycle is complete, i.e. either each input object has received a spike, the cycle time has ended or the reference input has received a spike, the object triggers itself to decode the spike timing and send the output event (see algorithm 9)

For decoding temporally coded spike events, it is again the method `processSimEvent` in which the computation takes place. It is called when the output cycle has been completed and performs the computation of the real numbered output vector.

```
public void processSimEvent(SimEvent e) {
    double outVector[] = new double[inputs.size()];
    for (int i=0; i<inputs.size(); i++) {
        Input inp = (Input) inputs.get(i);
        if (inp.hasReceivedSpike()) {
            outVector[i] = (getTime() - inp.getReceivedTime()) / multiplicator;
            inp.reset();
        }
        else
            outVector[i] = Double.NEGATIVE_INFINITY;
    }
    for (int i=0; i<listeners.size(); i++)
        ((VectorOutputListener) listeners.get(i)).outputEvent(outVector);
}
```

The output vector is generated using the formula defined in section 3.4 and is simply sent to all registered listeners that implement the `VectorOutputListener` interface.

Algorithm 9: Temporal decoding of an output vector.

OutputFilter This class is a decorator for output listeners, enabling the filtering of output events:

- First of all, it can pass only the first event and filter out all subsequent ones.
- Secondly, events can be forwarded when they have reached an equilibrium, i.e. they do not change anymore within a certain error range.

The constructor initializes the filter. When `forwardMultipleEvents` is `true`, then all events will be forwarded (depending on the value of `waitForEquilibrium`). If it is `false`, only the first event (also depending on `waitForEquilibrium`) will be forwarded.

When `waitForEquilibrium` is `true`, then an event will be forwarded only when the difference to the previously received event is within a certain, de-

fined margin. E.g. when `waitForEquilibrium` and `forwardMultipleEvents` are both `true`, then the filter discards all events that change too much and forwards the first one that has a small difference (as given by `maxSquareError`). After this forwarding, the filter resets and waits for the next equilibrium.

The "error", i.e. the difference between two received events, is calculated as the squared difference between the vectors or matrices. When the error is below the given `maxSquareError`, then the state is regarded as being an equilibrium. It is important to note that the error is divided by the number of elements in the vector or matrix for normalization.

ImageOutput This class receives vector or matrix output events from some spike decoder and transforms the values into an image, which is immediately saved in the file given to the constructor. It also needs the JAI library.

All of the described network input and output classes form a framework that can be used for many purposes and which can be extended easily. Due to the use of abstract interfaces, the inner spike coding schemes are easily interchangeable, alleviating research on different schemes.

4.4 Learning

In the context of SNNs, there are currently supervised as well as unsupervised learning methods mentioned in various papers (see section 3.5). SNNs are, in the opinion of the author, naturally aimed at unsupervised learning methods. Although the emulation of typical supervised learning algorithms such as the backtracking algorithm is possible within these networks, it does not seem beneficial to use such algorithms. For solving problems with backtracking learning, ANNs will currently offer faster convergence. However, SNNs – especially in combination with discrete event simulation – allow more dynamics in the learning process, e.g. by making the creation and removal of synapses simple and straight-forward (also refer to section 6.2). Other ways of determining network parameters such as it is done for Hopfield networks can be implemented without learning during the simulation (refer to section 5.3).
Therefore, in the current simulation framework, there is no support for supervised learning algorithms; it could be added easily by introducing global learning objects with access to all `Synapse` objects in the simulation. On the contrary, unsupervised learning methods can mostly be implemented with locally available information. In the following, the approach taken in this diploma thesis will be described. Since learning is mostly done for synaptic parameters, the class `Synapse` allows two ways to implement specific learning algorithms:

1. By overriding `Synapse` or a subclass thereof, the method `postNeuronFired` can be overridden. In this method, learning on solely locally available information (i.e. all synaptic parameters, the time of the last pre-synaptic spike event and the time of the last post-synaptic firing) can be done each time the post-synaptic `Neuron` object fires a spike event.

2. By creating an object that implements the interface `Synapse.LearningMethod` and registering it with every `Synapse` object that should use this learning behavior. This object can, depending on the specific implementation, have access to as much information as needed for the learning algorithm (for a short discussion of global vs. local learning refer to section 6.4).

Although `Synapse` can, by definition, not exhibit any support for the first possibility, the second one is supported by the inner interface `Synapse.LearningMethod` and the default implementation of `postNeuronFired`:

Synapse.LearningMethod This interface is used for external learning algorithms that are non-local to the `Synapse` object, the only defined method is

```
public void changeParameters(Synapse sender)
```

As can be seen in algorithm 2, a `Neuron` object will notify all of its input `Synapse` objects of its firing by calling `postNeuronFired`. The default implementation of `postNeuronFired` in `Synapse` calls the method `changeParameters` of `LearningMethod`, if it has been registered previously (and `postNeuronFired` was not overridden). Obviously, since the abstract class `Synapse` does not define any synaptic parameters, the used `LearningMethod` object will depend on the overriding class.

For usage of this interface the reader is referred to the example simulation described in section 5.4. The main reason for implementing an abstract interface is the flexibility that is offered by this approach. Since Java does not support multiple inheritance, overriding `Synapse` for implementing learning algorithms in `postNeuronFired` cannot be done in a reusable way – e.g. it would be impossible to use the same learning algorithm for `ConstantPLSynapse` objects and `PLSynapse` objects with different response function shapes without having duplicate code. On the other hand, learning algorithms contained in objects which implement `Synapse.LearningMethod` can be used for arbitrary classes derived from `Synapse`. Even if this approach is at compile-time not type-safe for the input parameter `sender`, run-time type information (*RTTI*) can be used to ensure at least run-time safety – as it is done in the example simulation described in section 5.4.

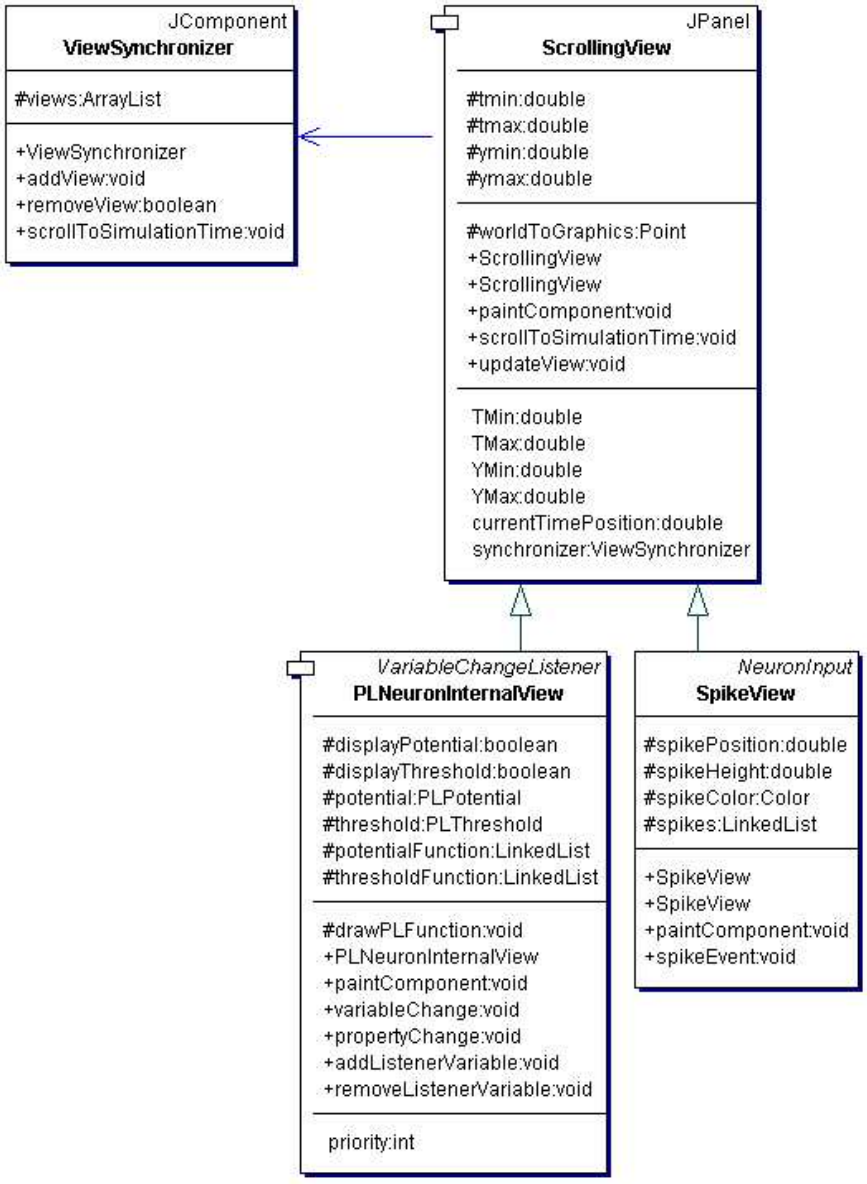


Figure 4.6: Classes handling the visualization of simulation parts.

4.5 Visualization

Visualization is performed using Java Swing components (to display the trajectories) and internal events as well as state change events (to communicate changes in simulation states).

ScrollingView This class implements a horizontally scrolling view for 2D trajectories. The view uses world coordinates in a virtual window, defined by minimum and maximum values, which are mapped to graphics coordinates in the resizable simulation window. It can also synchronize with other `ScrollingView` objects

so that multiple views always show the same simulation time-frame.

ViewSynchronizer This class is able to synchronize multiple `ScrollingView` objects so that they always show the same current simulation time.

SpikeView This class implements a scrolling spike view where spikes are simply drawn as vertical lines marking the simulation time points when the spikes occurred. It can directly receive spike events (by implementing `NeuronInput`) and will also record those spike events for drawing and logging.

PLNeuronInternalView This class implements a view for the potential and threshold functions of a `Neuron`, thus its internal states. It listens to state change events of `Potential` and `Threshold` objects and keeps a compact, internal history of the past states as far as needed.

These visualization classes not only display the spike and state trajectories, but also show the flexibility of the simulation framework. To allow this powerful simulation, no changes to the core simulation classes were needed at all; instead, the visualization classes are only registered as additional event listeners. In the same way, more visualization, logging or statistical components could be added easily.

Chapter 5

Experimental Results

In this chapter, a few experimental results obtained by simulations are described. All of these simulations were created using the framework described in the previous chapter. Since the simulation framework is based on MOSAIC, the basic structure of simulation applications also stems from this. Therefore, all simulation applications have to be derived from `SimApplication` and include a component derived from `ModelPanel`. This panel will then contain all of the simulation components, including `Neuron` and `Synapse` objects, which are normally not visible, and visualization components.
In the following sections, the basic topologies used in the simulated SNNs are shown and the basic behaviors of the different simulation experiments are explained. A comparison with simulation results from the GENESIS simulator (confer section 2.2.3) can only be given for the first simulation example because no existing, freely available GENESIS simulations of the other network types could be found; creating the examples in GENESIS would have been outside the scope of this diploma thesis.

5.1 Simple recurrent network

This example simulation resembles an example that is distributed with the standard GENESIS simulator [BB94]. It is available under the examples directory

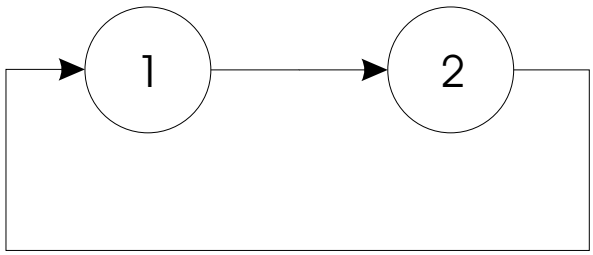
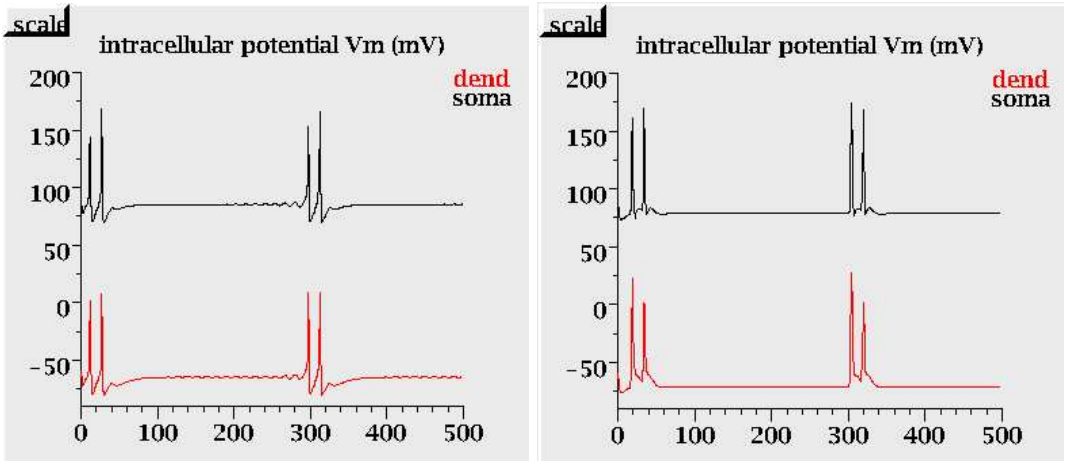


Figure 5.1: Topology of a simulation of a simple recurrent network acting as an oscillator.



(a) Cell 1 (b) Cell 2

Figure 5.2: GENESIS simulation results of the simple recurrent network.

(`/usr/share/genesis/Scripts/` on a Debian GNU/Linux system) as `MultiCell` and described by the author of that simulation as follows:

> *This is a simulation of two simplified, but realistic neurons in a feedback configuration. Each cell is composed of two compartments corresponding to a soma and a dendrite. Each compartment is composed of two variable conductance ionic channels for sodium (Na) and potassium (K). The dendritic channels are synaptically activated while the somatic channels are voltage dependent with Hodgkin-Huxley kinetics.*

There is also a more detailed description of the simulation model available in the documentation, for details, the reader is referred to the file `MultiCell.doc`. Fig. 5.1 shows the principal topology of the model, with two neurons being fully connected in a feedback loop. In this configuration, the synaptic delays are of utter importance for system stability; without these delays the neuron potentials would quickly diverge to infinite values. The results of the GENESIS simulation of this biologically realistic model are shown in Fig. 5.2. As can be seen, the whole network forms an oscillator in which both neurons periodically fire two temporally close spikes. Cell 1 fires at some small time before cell 2 and the duration of the oscillation period is much longer than the time between the dual, adjacent spikes.

The discrete event simulation pendant is implemented in the class `mosaic.sim.neuron.simulations.genesis.MultiCell` and can be launched by calling

```
java -classpath DEVSNeuron-bin-<date>.jar
mosaic.sim.neuron.simulations.genesis.MultiCell
```

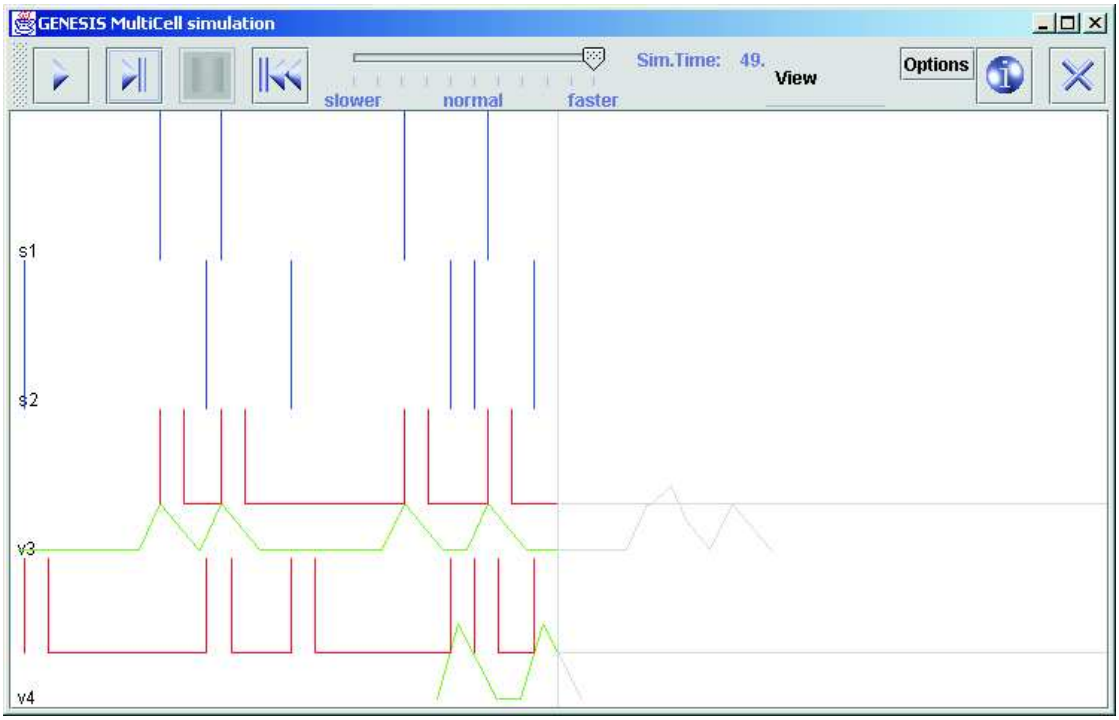


Figure 5.3: Discrete event simulation results of the simple recurrent network.

To achieve comparable simulation results, careful parameter tuning was required because currently there is no rule for deriving the parameters of the piecewise linear simulation model from the parameters in the compartmental model. Moreover, it turned out that the values of the delay of the first synapse (from cell 1 to cell 2) and the weight of the second synapse (from cell 2 to cell 1) were critical for the stability of the overall simulation. Therefore it might be argued that an SNN simulated using the model described in chapter 3 is not stable with regard to its simulation parameters [Pic00b]. It might or might not be stable with regard to its input values when the simulation parameters are chosen appropriately – as there are no input values, this stability criterion is not applicable to the present example .
Nevertheless it was possible to achieve comparable results in the firing pattern of both neurons. Fig. 5.3 shows the firing pattern of the simulation. As can be seen, the firing pattern of the GENESIS simulation is reproduced closely enough to prove that for simulating this simple example, the abstractions and simplifications in the discrete event model do not influence the qualitative properties of the network. It would also be possible to emulate the firing behavior more closely by changing the synaptic delays, but this does not influence the qualitative results; currently the delay of the synapse from cell 2 to cell 1 is about 5 times higher than the delay of the synapse from cell 1 to cell 2.

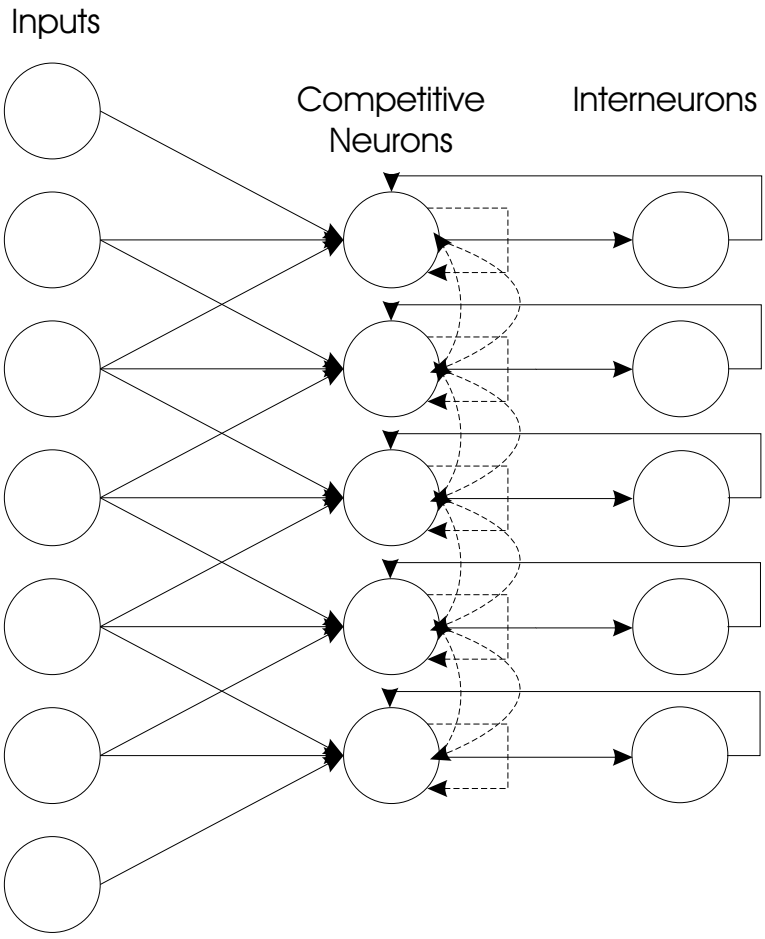


Figure 5.4: Topology of a simulation of a Cuneate-Based Network.

5.2 Cuneate-Based Network

This example simulation implements a *Cuneate-Based Network* (CBN) as described in [SBMC01] and [SMB01]. A CBN models the cuneate nucleus, which is a part of the sensory system in the brain stem. It is capable of spatial and temporal filtering of spike trains, i.e. it has edge-detector like capabilities for amplifying the "edges" in the input signal, can detect the "strongest" of those edges and will furthermore suppress signals that do not change over time. The basic coding scheme of this network type is rate coding; therefore this simulation shows that the simulation framework is capable of computations in this coding scheme.

The topology of a CBN, as shown in Fig. 5.4, is composed of two interconnected layers of neurons: the first layer is a competitive layer to which the inputs are connected using restricted input windows for each competitive neuron (in the example the input window has a size of 3) and the second layer implements interneurons. Obviously, the number of interneurons corresponds to the number of competitive neurons as the interneurons

form a feedback configuration with the competitive neurons.
All of the above mentioned features of a CBN can be explained by this special topology:

- edge detection: The edge-detection is performed by the neurons in the competitive layer using the given input window and appropriately chosen weights of the respective synapses. In the example simulation, a variation of the laplacian and sobel edge extractor operators [Cas95] has been chosen for computing those weights.

- spatial filtering: Due to the inhibitive lateral connections in the competitive layer, only those neurons that detect the "strongest" edges will actually fire; this behavior is comparable to the implementation of the winner-takes-all principle of SOM networks in SNNs (refer to section 5.4). However, in the current simulation of a CBN, only a restricted number of competitive neighbors is connected instead of the whole competitive layer being fully connected.

- temporal filtering: Finally, the interneurons with their inhibitive feedback configuration form a kind of temporal filtering which is able to extract temporal changes in the input signal. It is important to note that this temporal filter does not prevent the competitive neurons from firing but only lowers their firing rate whenever their firing pattern is stable, i.e. the rate-coded input signal does not change over time. Thus, a stable, low output firing rate of the CBN can be considered to not carry any relevant, new information.

For a more detailed description of the network features, the reader is referred to [SBMC01] and [SMB01].
In Fig. 5.5, 5.6 and 5.7 the discrete event simulation of a CBN is shown, which is implemented in the class `mosaic.sim.neuron.simulations.CBN` and can be launched by calling

```
java -classpath DEVSNeuron-bin-<date>.jar
mosaic.sim.neuron.simulations.CBN
```

The first 9 views on the very top of the simulation windows show the inputs of the network, which are simply rate coded with a constant frequency; inputs 0 to 2 (labeled *spikeview input0* to *spikeview input2*) have a frequency of 0.1 spikes per simulation unit (*spu*), 3 to 5 (labeled *spikeview input3* to *spikeview input5*) a frequency of 0.2 spu and 6 to 8 (labeled *spikeview input6* to *spikeview input8*) a frequency of 1.0 spu. Therefore, there is one "edge" between input 2 and 3 and another between 5 and 6 with the last one being "stronger". The last 7 views (labeled *spikeview output0* to *spikeview output6*) show the outputs of the network in various phases of the computation. In

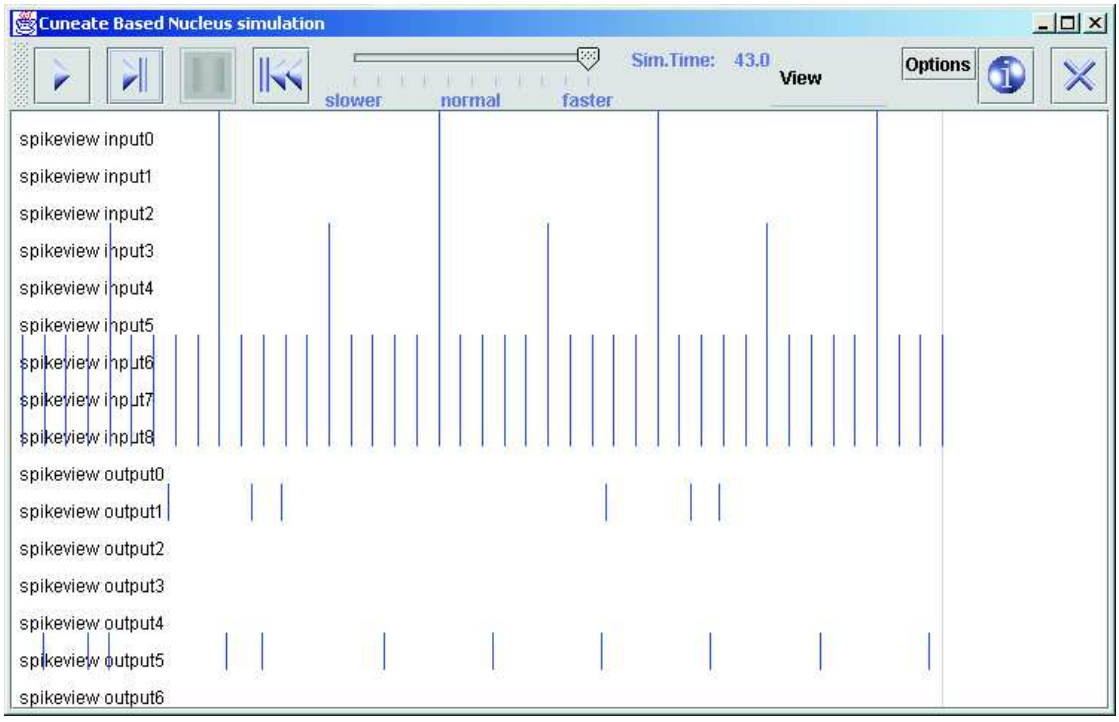


Figure 5.5: Discrete event simulation results of a CBN – initial phase.

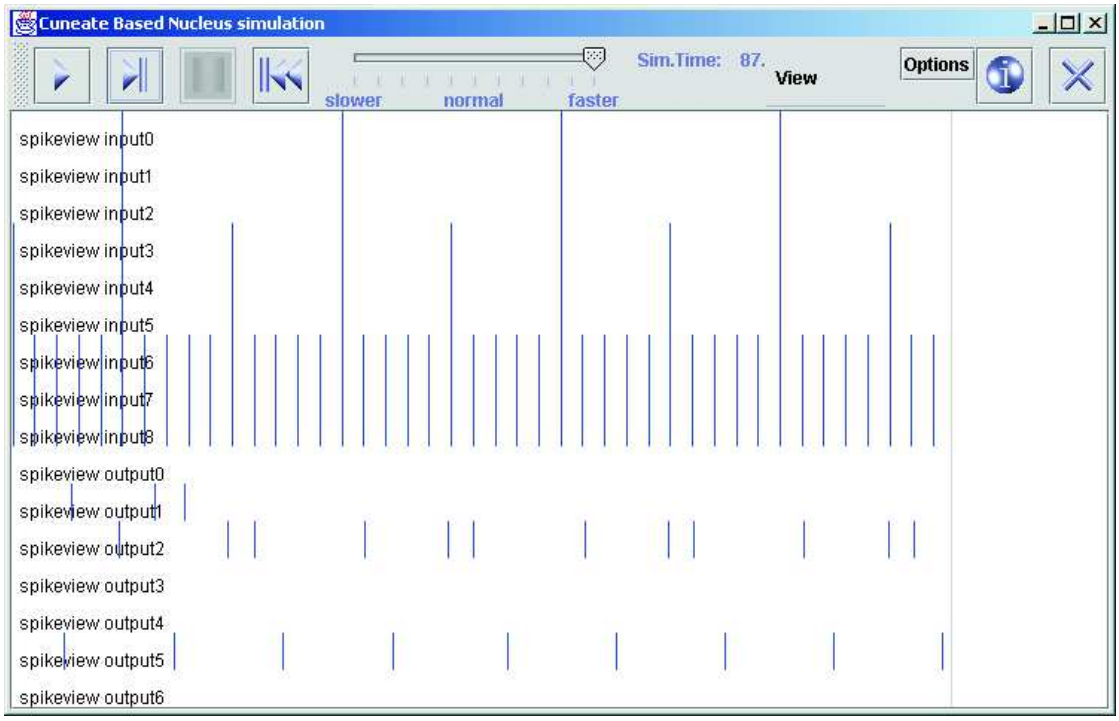


Figure 5.6: Discrete event simulation results of a CBN – intermittent phase.

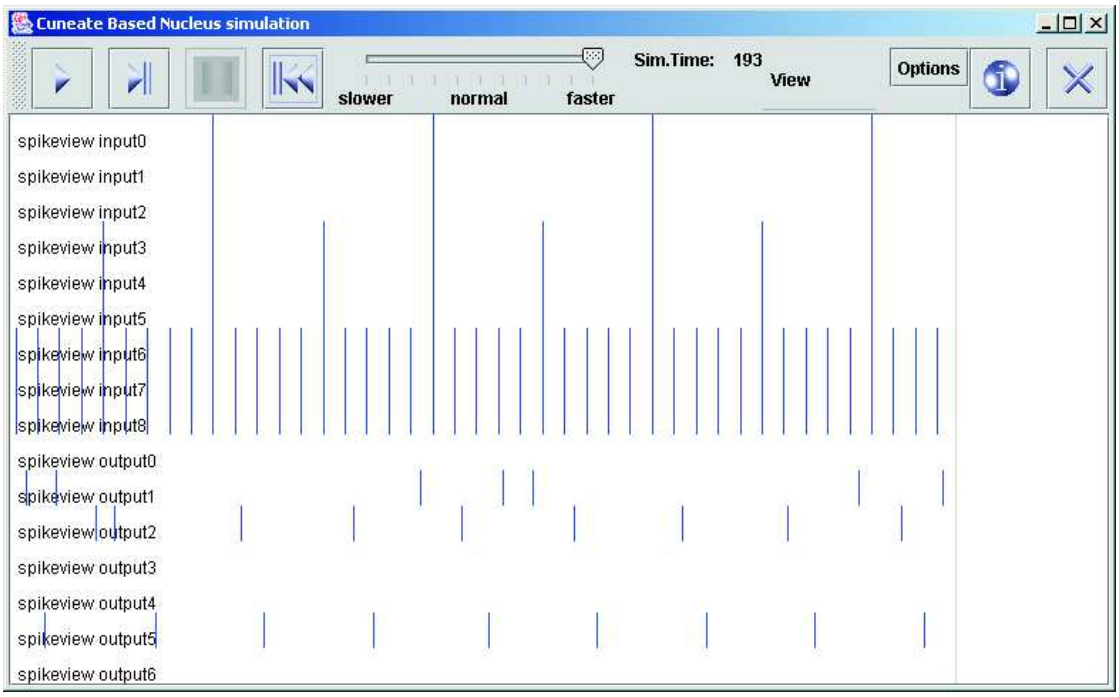


Figure 5.7: Discrete event simulation results of a CBN – equilibrium phase.

Fig. 5.5 the initial phase of the network is shown directly after applying the input. As can be seen, the network immediately reacts as an edge extractor, with only outputs 1 and 5 (those detecting the edges) being active. Due to the network also acting as a spatial filter, the outputs which are close to 1 and 5 are prevented from firing. Output 5 shows more activity than output 1 because of the higher input firing rates – but the distance is too large to be covered by the restricted width of the competitive field. After this initial phase, the interneurons start influencing the network, causing the output pattern to fluctuate (see Fig. 5.6). However, since the input pattern does not change over time, the network settles down and a stable pattern emerges (see Fig. 5.7).
The presented example shows how well discrete event simulation of SNNs is able to reproduce the behavior of a biologically inspired, reasonably complex network utilizing not only a feed-forward model with synapse weights but also synaptic delays as computational elements.

5.3 Hopfield network

In this simulation example, a Hopfield network with graded response [Hop84] is emulated by a SNN as described in [MN97]. For the internal representation of neuron input and output values, temporal coding is used (refer to section 3.4) – therefore the temporal input coder is used for providing the network with input patterns (refer to section 4.3). The simulation is implemented in the class

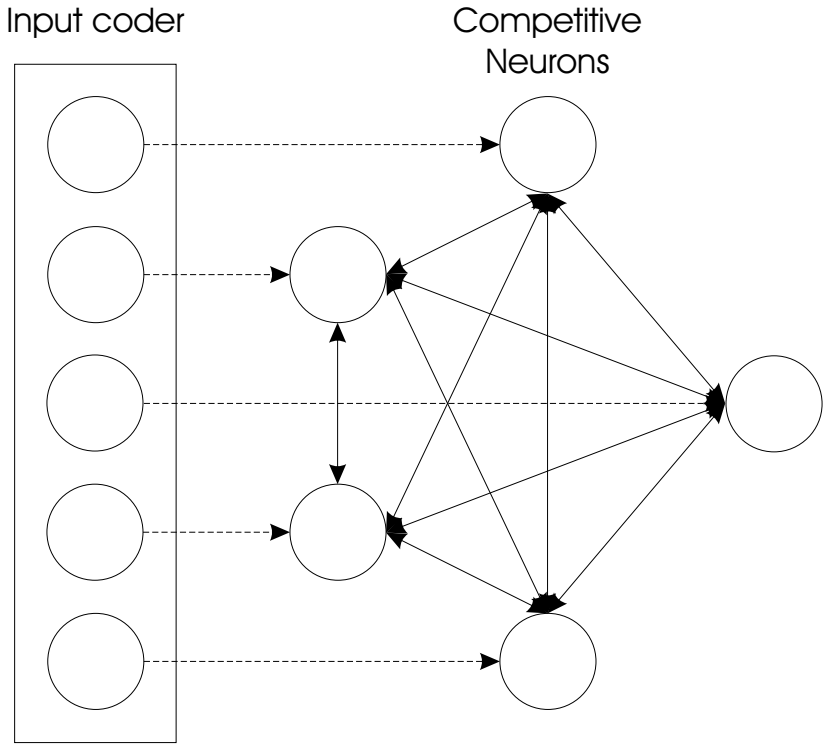


Figure 5.8: Topology of a simulation of a Hopfield network.

Training pattern 0	$\langle 1,0,0,0,0,0,0,0,0,1\rangle$
Training pattern 1	$\langle 0,1,0,0,0,0,0,0,1,0\rangle$
Training pattern 2	$\langle 0,0,1,0,0,0,0,1,0,0\rangle$
Training pattern 3	$\langle 0,0,0,1,0,0,1,0,0,0\rangle$
Training pattern 4	$\langle 0,0,0,0,1,1,0,0,0,0\rangle$
Input pattern	$\langle 0.7,0.3,0.2,NF,NF,NF,NF,NF,0.1,0.3\rangle$

Table 5.1: Training and input patterns for the Hopfield network simulation.

`mosaic.sim.neuron.simulations.Hopfield` and can be run by calling

```
java -classpath DEVSNeuron-bin-<date>.jar
mosaic.sim.neuron.simulations.Hopfield
```

In the Hopfield network model, all neurons that take part in the computation are fully connected to each other. Fig. 5.8 shows how each neuron is connected to all others and receives exactly one input from outside the network. The input classes are parts of a single input coder – in this case an object of class `TemporalInputCoder` (refer to section 4.3) – and therefore provide consistent input from a real numbered vector that is given from the outside of the network. As defined in the coding scheme, the firing times of single spike events sent by the inputs to the neurons in the computational layer encode the values of the input vector. As the network is inherently recurrent, the computation started by input spike events will run infinitely; the output of the network

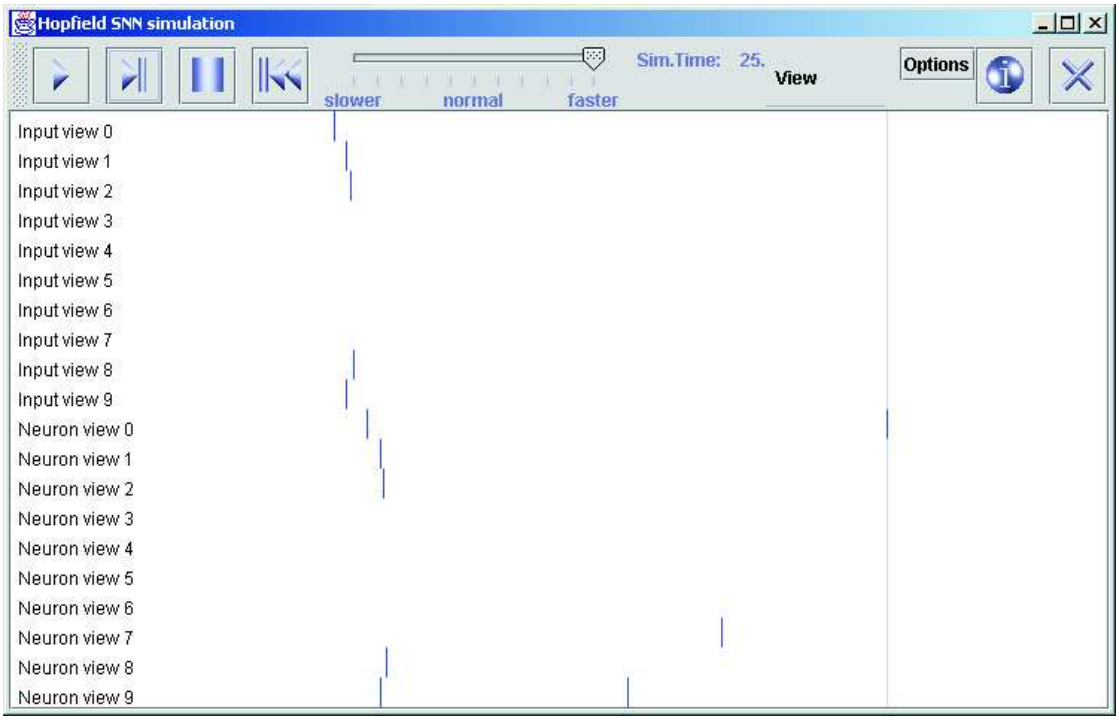


Figure 5.9: Discrete event simulation results of a Hopfield network – initial phase.

is therefore defined as the stable states that the neurons converge to. With SNNs, this corresponds to a stable, repetitive firing pattern of the neurons. For detecting if a firing pattern is stable, the class `OutputFilter` can be used.

In the current prototype implementation, there is a fixed set of training patterns for computing the weights (see table 5.1) in the initialization phase. Then a single, also fixed input pattern, which is a mixture of the first two input patterns with some noise added, is given as input to start the network computation (see table 5.1; *NF* means *non-firing*, i.e. the corresponding input neuron will not fire at all). As can be seen, the input patterns are orthogonal to each other and should therefore form a very stable base for fix-points in the weight set. The input as shown in table 5.1 clearly shows an affinity to the first training pattern and should therefore cause the network to converge to a stable output of training pattern 0.
In Fig. 5.9 the input spike pattern is shown in views *Inputview 0* to *Inputview 9*. Currently the emerging, stable firing pattern shown in Fig. 5.10 is only based on the large synaptic delays and not on the behavior that is expected from a Hopfield type network – to converge to a stable output pattern that resembles one of the training patterns. At the time of this writing, it is unknown to the author why the network does not show the expected behavior since the instructions available in [MN97] and [MN98] have been followed during creating the simulation. However, in future work it might be beneficial to contact the authors of those papers and clarify any problems in implementing the

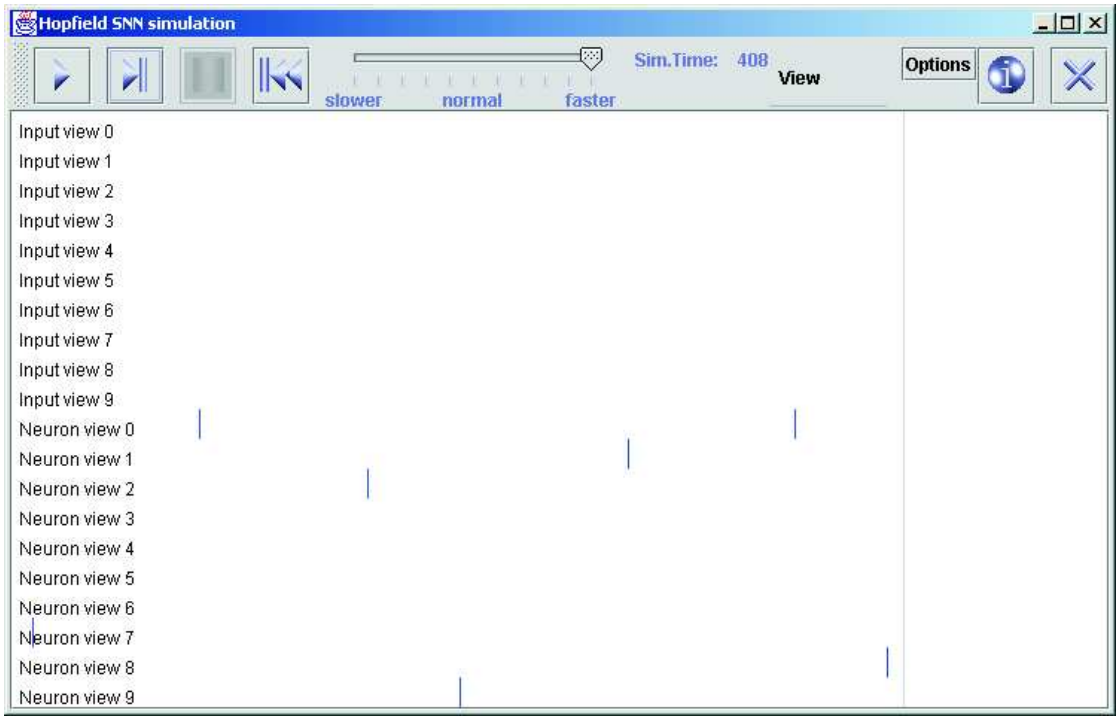


Figure 5.10: Discrete event simulation results of a Hopfield network – equilibrium phase.

theoretical concepts in specific simulations.

5.4 Self Organizing Map

This example is a simulation of a Self Organizing Map (SOM) as introduced by Kohonen [Koh95], but as a SNN in temporal coding. Temporal coding has – in this case – the enormous advantage that the winner neuron in the competitive layer can be computed fast and locally. Other SOM simulations with SNNs that use the rate coding scheme depend on the firing rate of the competitive neurons for determining the winner neuron in each training cycle. However, this implies that the global learning algorithm has to wait until each neuron has fired a few times for calculating the firing rate. Therefore, temporal coding offers significant advantages in this application.
The simulation is implemented in the class `mosaic.sim.neuron.simulations.SOM` and can be run by calling

```
java -classpath DEVSNeuron-bin-<date>.jar
mosaic.sim.neuron.simulations.SOM
```

In this example, the hot-spots provided by the simulation framework for the implementation of learning algorithms are used for the first time.

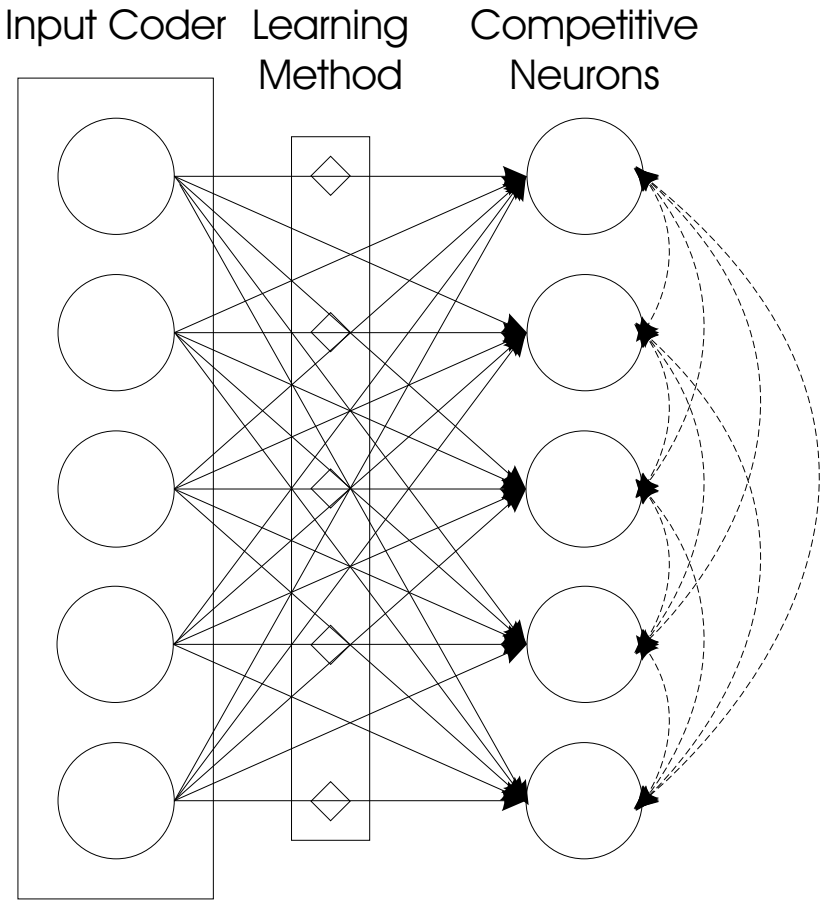


Figure 5.11: Topology of a simulation of a Self Organizing Map network.

The underlying model uses the standard SOM topology, which is shown in Fig. 5.11. As can be seen, there is a layer of competitive neurons which are fully connected to each other via inhibitive lateral connections and fully connected to the input layer via adaptive synapses (that are subject to learning). Synaptic delays are not used in this network model, only the weights are parameters of the network. Although the weights in the inhibitive lateral connections are also changed during the learning process, they are just decreased as the learning process advances. The calculation of these lateral weights as well as the learning rule for the weights of the input synapses are described in [Ruf98, chapter 7 and section 11.1]; an adapted learning rule is cited in section 3.5. Basically, this simulation can be divided into two phases:

1. Initialization phase: First of all, the simulation tries to load a training set that has been used in a previous run. If it finds a file called `SOM_training_set.dat`, it will load the input vectors used for training from this file. If this file does not exist, a new set of training vectors is randomly generated. As described in [Ruf98], the size of the training set – composed of one-dimensional input patterns – is 10 and the pattern values are uniformly distributed over $[0.1, 0.9]$.

After the training set has been initialized, the synaptic weights are set: For the input weights (the synapses connecting the input layer with the competitive layer), the mean values of the training set are used, random noise in $[-1,1]$ is added to each value and the weight vectors for each competitive neuron are normalized to a constant value λ. As with the input vectors, the simulation tries to locate a file named `SOM_computed_weights.dat` to load previously computed weights from. In contrast, the lateral weights are initialized as $\widetilde{w}_{k,j} = 2 \cdot \left(0.25 - m_{k,j}/m_{max}\right)$, where $m_{k,j}$ are the elements from the so-called neighborhood matrix and m_{max} is the largest element of the neighborhood matrix. The neighborhood matrix is defined as $m_{k,j} = |k-j|$, resulting in $m_{max} = N-1$ where N is the number of competitive neurons. Comparable to the handling of input vectors and input weights, the lateral weights are stored alongside the learning rate in a file named `SOM_learning_parameters.dat`. This way, the simulation can be restarted after it has been stopped.
Therefore, the input weights, which are subject to learning in the next phase, get initialized with nearly equal values for all competitive neurons but with some random noise (which is important for the learning algorithm to work) while the lateral weights, which are only decreased during the learning phase, are initialized with strictly deterministic values that form the neighborhood.

2. Learning phase: In this phase the network is constantly fed with input vectors from the training set and the learning rule is applied in each training cycle. To start the training cycles, time events are used again, but on a higher level than for scheduling spike events. Therefore, the training cycle is performed in the method `processSimEvent`, which is responsible for handling time events. Each time this method is called by the MOSAIC framework, the neurons in the competitive layer are reset, the learning rate and the lateral synaptic weights (the neighborhood) are decreased and a random input vector from the training set is fed into the input coder.
The input coder will then generate spike events according to the values in the input vector and send these events to the competitive neurons. Assuming that the input weights have been properly initialized, there will always fire at least one neuron in the competitive layer. When a neuron in the competitive layer fires due to the incoming spike events, it will influence all others via the inhibitive lateral synapses, delaying their firing with its own spike. Therefore the firing of the first neuron, the so-called winner neuron, will delay or possibly prevent the firing of the other neurons dependent on the size of the neighborhood (i.e. the weights of the lateral synapses).
For each neuron which fires, a learning method is applied to all connected input synapses at the time of the firing (refer to section 4.4 for details on how this is

supported by the simulation framework). In the SOM example simulation, one global object is used for implementing the learning algorithm. This object implements the interface `Synapse.LearningMethod` and can therefore change the synaptic parameters of the calling synapse in its method `changeParameters` – the current implementation changes only the weights because synaptic delays are not used in the SOM simulation. For computing the change of the weight value, the adapted learning rule stated in section 3.5 is used, but with a few changes to use locally available information: the weight w of a calling synapse is changed by $\Delta w = \eta \cdot (T_{out} - t)/T_{out} \cdot (s - w)$ where η is the current, global learning rate, t is the current simulation time (i.e. the time when the post-synaptic neuron fired), T_{out} is some time in the future when all neurons are guaranteed to have fired and s is the value that the pre-synaptic input neuron represented (i.e. the respective input value). Only the values η and T_{out} are global, all others including s, which can be derived from the time the pre-synaptic neuron fired, are obtained from locally available information. Therefore, a learning algorithm that relies solely on information that is available locally at each synapse might be constructed in the future – the current algorithm is a step in that direction.

To achieve the expected results, careful and long parameter tuning was necessary. It turned out that the rates for decreasing the learning rate and neighborhood as well as the starting values for the lateral weights (forming the neighborhood) were critical for the overall performance. In contrast, the parameters in the learning rule for changing the input weights did not seem to have such a major influence – maybe the range from which they can be chosen from is larger than those of the neighborhood parameters.
With the current parameter values, the simulation shows the expected behavior: the competitive neurons specialize on specific input patterns and after enough training cycles the inhibitive lateral connections prevent neurons other than the winner neuron from firing. In the simulation source code, the parameters are defined as constants and can therefore be examined and changed very easily. Fig. 5.12 shows that, before training is conducted, the neurons react strongly to the random input pattern, with all of the competitive neurons firing at some time. As described in [Ruf98, chapter 7 and section 11.1], about 4000 training cycles, with one random input pattern per training cycle, were necessary. On a standard desktop workstation with an AMD Athlon 1700+ CPU it took about 4000 seconds to compute these learning cycles, so that on average one learning cycle can be done in one second. However, the first 100 learning cycles took over 250 seconds and later cycles significantly less; this is a direct consequence from using discrete event simulation. Due to the inhibitive lateral connections, other competitive neurons are prevented from firing in the later stages of the training. Therefore those neurons that do not fire anymore do not need to be taken into account, allowing the training cycles to be handled faster. Fig. 5.13 shows the network behavior after

Figure 5.12: Discrete event simulation results of a SOM network – before training.

Figure 5.13: Discrete event simulation results of a SOM network – after training.

about 4000 training cycles – one cycle is defined between two adjacent reference input spikes, painted in green in the simulation window. As can be seen, now only one or very few neurons respond to a given input pattern, which is the expected behavior.

5.5 Comparisons

Since the new discrete event model for simulating SNNs uses a different approach in simulation than the currently used models, its characteristics might be quite different. One characteristic is obviously shared with the underlying technique of discrete event simulation: that the running time is dependent on the activity of the system respectively on the activity of parts of the system.
Current SNN simulation tools normally base on continuous simulation. This means that the state of every neuron and every synapse in the neural network needs to be computed at certain simulation time steps (which normally have a fixed width), independently of their current activity. Although the concept is very well suited for biologically realistic simulations and is straight-forward to implement, it might not be the best technique for simulating large networks when focusing primarily on computation. The reason is that large networks are expected to have less concurrently active neurons, thus in principle enabling good scalability. However, due to the nature of the computational model, continuous simulation can, when not being implemented in a highly optimized way, only scale linearly with the number of neurons and synapses. This leads to the tremendous advantage that for SNNs that have a low percentage of concurrently active neurons, a significant simulation speed-up can be expected when using discrete event simulation instead of continuous simulation. This can be achieved while still computing the neuron potentials as accurately as possible (only restricted by the accuracy of the underlying computer architecture and programming environment).
Currently the example simulations described in the above sections do not allow a direct comparison with continuous simulation, mainly because they were created especially for testing and verifying the developed simulation framework (see chapter 4); simulations with a defined input and output set and a clearly defined problem still need to be created. Therefore, one of the main goals of future work should be to rigorously compare the characteristics of both simulation models qualitatively as well as quantitatively. For representing continuous simulation, the freely available GENESIS simulator [BB94] will be used. Currently many scientific groups use specific simulation programs written especially for the respective focus of research [Ruf98, SM01, MN97, QC01, CBG01]. There are only a few standard simulation tools that are general enough to be used in different simulation situations. One of them is GENESIS, a simulation tool developed at the University of Berkeley, California and used by neuro-biologists as well as computer scientists for conducting research on bi-

ologically realistic neural networks. Since its main goal is to be biologically realistic, neurons and synapses are typically modeled with numerous compartments and simulated according to the Hodgkin-Huxley equations (refer to section 2.2.3). GENESIS seems to be a good representative for simulation tools using continuous simulation because it is widely used and accepted. For representing the technique of discrete event simulation, the simulation framework developed in this diploma thesis should be extended and optimized to suit the needs of conducting the rigorous comparison. The resulting simulation system should also be general, being a discrete event simulation pendant to GENESIS.
Therefore, simulation results obtained with GENESIS should be compared to simulation results obtained with the created simulation system:

- For qualitative comparison, the computational characteristics of the simulated SNNs will be the determinative factor, since this diploma thesis and possible future work mainly focus on making SNNs more usable in practical technical applications. These computational characteristics might involve the ability to do spatio-temporal filtering, to reproduce some trained patterns or to classify patterns. A sub-goal will be to specifically select some network types and applications that allow sensible qualitative comparisons.

- For quantitative comparison, the simulation speed will be the determinative factor. By modeling a specific network structure both in GENESIS and the new simulation framework – presumed that the qualitative features are comparable – the simulation speed will be measured in terms of advanced simulation time in some fixed real time; a second method will be to determine the largest number of neurons and synapses in some network structure that still can be simulated in real-time. These two measuring units allow a numerical comparison of the different simulation techniques.
  Although larger networks are expected to typically have a lower percentage of concurrently active neurons, thus allowing large network simulations to scale better than linearly, there will probably be cases where continuous simulation is better suited than discrete event simulation. The comparison of the simulation techniques will allow to make statements about the usability of discrete event simulation in different cases and application domains and might allow to give recommendations on the use of simulation techniques for given tasks. Probably there will also exist some critical parameters that influence the simulation speed of event simulated SNNs drastically. Another sub-goal of the comparison will be to find some of those critical parameters.

Summarizing it can be said that a rigorous comparison between continuous and event based simulation of SNNs still has to be conducted to fully prove the advantages of the

simulation model proposed in this diploma thesis; however, as it has been presented in this section, a reasonable comparison cannot be done easily and is therefore outside the scope of this diploma thesis.

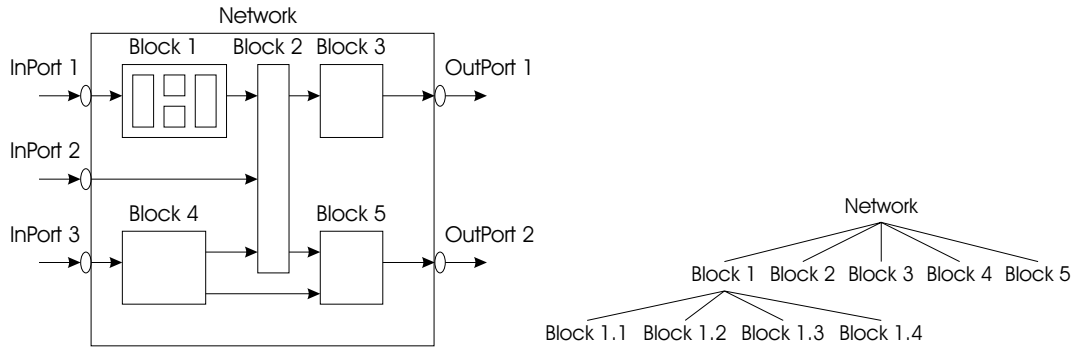
Chapter 6

Advanced Techniques

In this chapter a few, more advanced simulation techniques will be shortly discussed. These techniques are currently not used or implemented in the simulation framework, but some are at least partly supported by already defined interfaces. This chapter therefore lists some of the possible future enhancements or directions of development; for few of them, it still has to be decided if they are needed to make future simulations more powerful or if other, different techniques might be more advantageous. However, currently the methods described in the following sections seem to offer many benefits for making simulations faster, more flexible or more powerful. It is important to note that the first three techniques are completely independent of each other, making it possible to implement any of them without interfering with the others at all.

6.1 Building blocks

One possibility to ease the creation of large, complex SNNs is to construct them hierarchically: parts of the SNN can be composed of a number of simpler parts that are connected to each other via clearly defined inputs and outputs. These compositions can again form components of a larger network block themselves, also having defined input and / or output interfaces. The main advantage of building blocks, besides making it possible to handle the complexity of large SNNs with well-known system theoretical methods, is that these building blocks are reusable. Such blocks could be parameterized and thus be seen as black boxes with defined input and output behavior (e.g. a visual 2D filter modeled after the cuneate nucleus in the human brain).
As sketched in Fig. 6.1, a SNN can be composed of different blocks that are connected via their input and output ports, with different components handling different parts of the whole network's inputs and outputs. However, the same network can also be seen in a hierarchical view, showing how the components form a tree structure with the network being the root. This tree structure resembles a *holarchy* built from holons – entities that are a whole for themselves and a part of some other whole at the same



(a) Schematic black-boxes view

(b) Hierarchical (holon) view

Figure 6.1: Networks composed hierarchically of simpler, reusable components.

time (cf. [Pic98, Pic99]); the subtle difference between a hierarchy and a holarchy is that a holarchy does not allow the inclusion of parts from a higher level in parts of a lower, e.g. a molecule contains atoms but not vice versa.
In the current simulation framework, this concept has been used to build the network input and output converters as described in section 4.3. Those converters offer a well-defined input or output interface of the whole SNN, effectively transforming it into a black box to outside components. Although the example simulations which implement various network topologies (see chapter 5) are currently not implemented as building blocks that can already be integrated into other SNNs, care has been taken to make them as general as possible – most of the parameters that control the topology and behavior of the networks have been implemented either as parameters or as constants. Thus it should be simple to convert those example simulations into parameterizable, black-box building blocks as soon as interfaces for constructing holarchies have been defined and are supported by the simulation framework.

6.2 Dynamic topology

Within the current simulation framework, which is implemented in a clean, object-oriented manner, there is the possibility for handling the number of neurons as well as connections, i.e. the synapses, within an SNN dynamically. Because of the inherent event based nature of the simulation model, neurons are completely independent of the number of input and output synapses which are connected to them, even at run-time. In fact, neurons only maintain a dynamically managed list of synapses from which they receive input (for providing those synapses with post-synaptic firing events used for learning purposes, see section 4.4) and to which they should send their firing events; there is no other information necessary, and therefore available, in the neurons. Both of

those lists can be changed easily during run-time. Of course, it is also possible to add and remove whole neurons during run-time because they can be connected to others on the fly by creating appropriate synapses.
Therefore, the simulation framework lends itself naturally to techniques like pruning and growing [LDS+90, PH95], i.e. changing the network size and connectivity during learning. These techniques can now be implemented easily in specific learning algorithms – possibly even without weight normalization which is biologically controvertible but used by some current algorithms [LO01].
Taking the ideas from the previous section into account, the technique of dynamic connections can be generalized to work at any layer of the hierarchy: on the layer of neurons and synapses, i.e. inside a building block, it can be used to create and remove synapses due to learning rules; on the layer of building blocks, it could be used to dynamically connect the inputs and outputs of different building blocks to each other, e.g. because of blocks with different parameters being varyingly successful in solving a given sub-problem.
With the current simulation framework, which builds on event simulation, dynamical interaction between system components during run-time becomes possible and can be conducted easily. These new possibilities might well lead to the development of new learning algorithms that supersede current ones in quality and speed.

6.3 Parallel simulation

As the simulation of neural networks, especially of the more complex SNNs, requires high computational power to run in real time even with careful optimizations, a single processor machine might be too slow to tackle large problems. Although available processor speed currently seems to abide by Moore's Law [Sch97], there will always be problems that are too complex for single processor machines in terms of execution time or memory consumption. Therefore, at the moment the only way to approach such problem areas seems to be the use of parallel systems, i.e. systems composed of more than one interconnected processors; in this diploma thesis the term *parallel simulation* is used to describe a simulation running on a parallel computer system. *Parallel simulation* puts the reduction of execution time in the main point of view as opposed to connecting geographically dispersed simulations, which is referred to by *distributed simulation*.

As has been outlined in previous chapters, SNNs – as a model of neural networks – are parallel in their innermost nature. Therefore they nearly demand the application of parallel simulation. But, and this could be seen as a disadvantage of discrete event simulation, a causal chain emerges due to the events sent from one simulation component to the other [ZPK00]. To satisfy this causality, events have to be processed in a strict

order which is partly sequential, imposing a limit on the degree of possible parallelism. With continuous simulation, causality is always satisfied when the parallel processors run the simulation with the same speed; but with discrete event simulation, care has to be taken to assure correct parallel execution.
In the last two decades, a few approaches to parallel simulation have been developed, which can be categorized as follows:

- *conservative*: Introduced in the late 1970s [CM96, CM81], *conservative parallel discrete event simulation* principally strictly avoids causality violations. Therefore, events that constitute dependencies between different components are executed in the order of their timestamps.

- *optimistic*: In contrast to conservative approaches, *optimistic parallel discrete event simulation* allows the execution of events that might possibly violate causality; but it supports a rollback mechanism for annihilating the effects of events which were executed ahead of time and caused a causality violation. One implementation of this approach is the *Time-Warp* algorithm [JS85, JBW$^+$87].

Based on these principles, the underlying DEVS simulation system – currently MOSAIC – could be extended to a parallel discrete event simulation (PDES) system. When the object-oriented concept is followed closely, only minimal changes should be necessary to adapt the current simulation framework for utilizing parallel simulation. However, modifications on a higher level, namely on the level of specific simulations, might become necessary to minimize the dependencies between simulation components and therefore achieve a significant speed-up in the parallel simulation. The use of closed, black-box building blocks with restricted sets of input and output ports might help in achieving low coupling between and high coherence in those blocks. This as well as the use of local learning algorithms (see next section) minimize the dependencies and are thus advantageous for efficient parallel simulation.

6.4 Local learning

For various topologies of neural networks there exist different learning algorithms which are more or less capable of tuning the network parameters to achieve a desired behavior. However, many of them depend on at least partially global information such as some error value from other layers (cf. [Zel94, chapter 8]) or a “winner” among some group of neurons (cf. [Zel94, chapter 15]). Such learning techniques, in this diploma thesis called *global learning*, are disadvantageous because of mainly two problems:

- It is very difficult, if not impossible, to achieve reasonable speed-up due to the use of parallel simulation as described in the previous section; gathering all the information needed for learning almost always requires synchronization between the parallel processors. Therefore the simulation speed that can be achieved is drastically limited with the learning algorithm representing the bottleneck.

- The use of such global information, which is not available locally at those components where learning happens (normally the synapses) is biologically questionable (a good summary can be found in [KK00]).

Therefore it might be much better to develop *local learning* algorithms that utilize solely information which is locally available to the component that is modified by learning, e.g. the synapse. In the current simulation framework, a `PLSynapse` object (a `Synapse` object with piecewise linear response function) has the following information locally available: its own weight and delay parameters, the time of the last firing of the pre-synaptic Neuron object and the last firing time of the post-synaptic `Neuron` object (which is biologically motivated by backpropagating action potentials).
Particularly in the context of parallel simulation, local learning algorithms might offer enormous increases in simulation speed because of less interdependencies between simulation components. The adapted version of a SOM learning algorithm presented in section 5.4 is a first approach to a truly local version of SOM learning. Currently it seems that unsupervised learning methods are generally better suited for local learning because supervised methods always need the desired output value or state in addition to the output produced by the network. This desired output must be non-local by definition – it is given from the outside of the network by some "trainer".
But there is also some drawback of local learning algorithms: They might collide with the utilization of dynamic synapses (refer to section 6.2). Currently it is unknown to the author if it is possible to locally determine the necessity for creating or removing simulation components, such as neurons or synapses – this should be a topic of future research.

Chapter 7

Conclusion and Outlook

In this diploma thesis, a new model for simulating SNNs efficiently has been introduced, backed by a prototype simulation framework which already implements it. After explaining the currently know theoretical foundations in chapter 2, the model has been presented in chapter 3. The most important aspect, which differentiates this new model from other ones, is the use of discrete event simulation (cf. section 2.3). To enable the efficient simulation by calculating the neuron firing times in advance, the model utilizes piecewise linear functions; a formal syntax for computing with piecewise linear functions has been introduced (cf. section 3.1) and algorithms for merging – used for calculating the effect of a synapse response on the post-synaptic neuron's potential – as well as intersecting two functions – used for calculating firing times – have been described and implemented (cf. sections 3.2 respectively 3.3). These algorithms form the basis of the implemented prototype simulation framework; it is built around neurons and synapses which communicate via internal system events (cf. chapter 4). As could be seen, the basic structure of the framework is very abstract, allowing many different features to be implemented. One such possible enhancement would be to use other function types in addition to the currently used piecewise linear ones – maybe polynomial or spline based functions might be beneficial to solve some given problems efficiently.

Moreover, a few example simulations were performed to show the current features of the simulation framework (cf. chapter 5). Starting with a very simple feedback oscillator configuration, the main strengths of the framework were shown. This first example was chosen to show that simple simulations can be built easily but automatically utilize the full power of event based simulation. A simulation of a biologically inspired filtering network, a CBN, was presented as the second example, showing how the firing rate coding can be used to feed input into an SNN. To show that SNNs can indeed emulate standard ANNs, the third example shows the simulation of a Hopfield type network in temporal coding. This example clearly benefits from the discrete event simulation, since the recurrence of the Hopfield network is inherent, caused by the fully connected,

feedback topology; it does not have to be implemented especially but the SNN uses standard neurons and synapses without any modification. There is also no need for "steps" in the simulation, usually supervised globally. Finally, the last example is a more complex one, simulating a SOM network in temporal coding. It uses the framework's capabilities for implementing learning algorithms by defining a partially global learning mechanism – but the aim should be to use local learning algorithms.

Although at the moment there are no direct comparisons to the use of continuous simulation, the potential of this new technique is clearly visible. Due to the application of discrete event simulation techniques it should be reasonable to simulate large-scale neural networks with thousands of neurons on standard workstations, also facilitating the investigation in the influence of different function types on the computational power of spiking neurons. This expectation is based on the fact that the application of discrete event simulation for ecosystems showed a speed-up of about 100 (!) [Moo96]. This significant increase in simulation speed can – in addition to the better scalability – be used to drastically increase the network size. For more sophisticated simulations, the number of neurons is expected to be the determinative factor. Furthermore, the additional degrees of freedom due to the freely definable shape of all functions in the system will allow to conduct experiments on the importance of various aspects of the model on the computational power. Without being restricted to finding a closed mathematical representation, this freedom in modeling function shapes can allow completely new computational elements to be built out of spiking neurons. One aim of future work should definitely be to create simulations that are currently not possible with continuous simulation. However, future research has to show that the assumptions that were made hold true and that effects of biological neural networks (such as synchronization between quasi-chaotically firing neurons) are reproducible with this model.
Finally, in chapter 6 a few possible directions for future developments were suggested. These enhancements might help in mastering the complexity of large neural networks, enabling the use of new learning algorithms or achieving additional simulation speed-up. More advantageous developments might come from research on information encoding – intuitively it seems that much more complex coding schemes are used in biology. Although empirical studies suggest that all of the different coding schemes that have been discussed in this diploma thesis (temporal coding, rate coding and population coding) are used in some areas of the human brain [RWdRvSB97], they might as well be used concurrently. When thinking about the fast information processing that the visual sensory system is capable of, together with being *the* human sense that also provides more detail information than any other sense, exactly one idea comes to mind: that spike coding resembles some sort of *fractal coding*, the first spikes carrying course information but transmitting it quickly and following spikes delivering more and more details.

Concluding it can be said that this diploma thesis can only be a small step into the future adoption of SNNs in practical applications. Although some may have the opinion that future research on neural networks will further differentiate the foci of computer scientists and neuro-biologists [Zel94, page 574], the author believes that the contrary may be true: SNNs might offer advantages for applications in computer science and for research in neurobiology, allowing both communities to share insights. Allowing a fast simulation of large networks by applying discrete event simulation is hopefully a step into this direction.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ANN	Artificial Neural Network
API	Application Programming Interface
ART	Adaptive Resonance Theory
ASCII	American Standard Code for Information Interchange
CBN	Cuneate-Based Network
EPSP	Excitatory Post-Synaptic Potential
HTML	Hypertext Markup Language
IFN	Integrate-and-Fire Neuron
IPSP	Inhibitory Post-Synaptic Potential
JVM	Java Virtual Machine
LIFN	Leaky Integrate-and-Fire Neuron
MIT	Massachusetts Institute of Technology
PSP	Post-Synaptic Potential
RBF	Radial Basis Function
SNN	Spiking Neural Network
SOM	Self Organizing Map
XML	Extensible Markup Language

Bibliography

[Arb89] M. Arbib. *The Metamorphical Brain 2*. John Wiley and Sons, New York, 1989.

[Ban98] J. Banks, editor. *Handbook of Simulation*. Wiley-Intersience, 1998.

[BB94] James M. Bower and David Beeman. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. Springer Verlag, New York, 1994.

[BCN01] J. Banks, J.S. Carson, and B.L. Nelson. *Discrete-Event System Simulation*. Prentice Hall, 2001.

[Cas95] Kenneth R. Castleman. *Digital Image Processing*. Prentice Hall, 1995.

[CBG01] Santi Chillemi, Michele Barbi, and Angelo Di Garbo. Synchronisation mechanisms in neuronal networks. In José Mira and Alberto Prieto, editors, *Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2001*, Lecture Notes in Computer Science, pages 87–94. Springer, 2001.

[CM81] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Comm. ACM*, 24(11):198–205, 1981.

[CM96] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed systems. *IEEE Trans on Sim*, 13(2):55–102, 1996.

[CS47] H. B. Curry and I. J. Schoenberg. On spline distributions and their limits: The polya distribution functions. *Bull. Amer. Math. Soc.*, 53:1114, 1947.

[CS96] H. B. Curry and I. J. Schoenberg. On polya frequency functions iv: the fundamental spline functions and their limits. *J. Analyse Math.*, 17:71–107, 1996.

[Gel89] E. Gelenbe. Random neural networks with negative and positive signals and product form solution. *Neural Computation*, 1(4):502–511, 1989.

[Gel90] E. Gelenbe. Stability of the random neural network model. *Neural Computation*, 2(2):239–247, 1990.

[Ger95] W. Gerstner. Time structure of the activity of neural network models. *Phys. Rev.*, 51:738–758, 1995.

[Ger98] W. Gerstner. *Spiking neurons*. MIT Press, Cambridge, 1998.

[GML99] E. Gelenbe, Z. Mao, and Y. Li. Function approximation with spiked random networks. *IEEE Trans. on Neural Networks*, 10(1):3–9, 1999.

[GvH94] W. Gerstner and J.L. van Hemmen. How to describe neuronal activity: spikes, rates or assemblies ? *Advances in Neural Information Processing Systems*, 6:463–470, 1994.

[Heb49] D.O. Hebb. *Organization of Behaviour*. Wiley, New York, 1949.

[Hen02] Rolf D. Henkel. Stereosehen und das zyklopische Auge. *Spektrum der Wissenschaft*, pages 10–16, April 2002.

[HH52] A.L. Hodgkin and A.F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[Hop84] J.J. Hopfield. Neurons with graded responses have collective computational properties like those of two-state neurons. In *Proceedings of the National Academy of Science USA*, volume 81, pages 3088–3092, 1984.

[Hop95] J.J. Hopfield. Pattern recognition computation using action potential timing for stimulus representation. *Nature*, 376:33–36, 1995.

[HT87] J. J. Hopfield and D. W. Tank. Neural computation by concentrating information in time. In *Proc. Na-*

tional Academy of Sciences, volume 84, pages 1896–1900, 1987.

[JA93] K.T. Judd and K. Aihara. Pulse propagation networks: A neural network model that uses temporal coding by action potentials. *Neural Networks*, 6:203–215, 1993.

[JBW+87] D. Jefferson, B. Beckmann, F. Wieland, L. Blume, and M. Diloreto. Distributed simulation and the time warp operating system. In *Proc. of the 11th ACM Symposium on Operating System Principles*, pages 77–93. ACM, 1987.

[JS85] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism. In *Proc. SCS Distr. Sim. Conf.*, pages 63–69, 1985.

[KK00] K.P. Körding and P. König. A learning rule for local decorrelation and dynamic recruitement. *Neural Networks*, 13:1–9, 2000.

[Koh95] Tuevo Kohonen. *Self-Organizing Maps*, volume 30 of *Series in Information Sciences*. Springer, Berlin, Heidelberg, New York, 1995.

[Kol57] A.N. Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Doklady Akademii*, 114:953–956, 1957.

[KSJ91] E.R. Kandel, J.H. Schwartz, and T.M. Jessel. *Principles of Neural Science, Third Edition*. Elsevier, 1991.

[LDS+90] Y. LeCun, J. Denker, S. Solla, R. E. Howard, and L. D. Jackel. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems II*, San Mateo, CA, 1990. Morgan Kauffman.

[LO01] Aleksandar Lazarevic and Zoran Obradovic. The effective pruning of neural network ensembles. *Proc. IEEE/INNS International Conference on Neural Neural Networks*, pages 796–801, 2001.

[Maa95] Wolfgang Maass. On the computational complexity of networks of spiking neurons. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Informa-*

tion Processing Systems, volume 7, pages 183–190. MIT Press, 1995.

[Maa96] Wolfgang Maass. Lower bounds for the computational power of networks of spiking neurons. *Neural Computation*, 8(1):1–40, 1996.

[Maa97a] Wolfgang Maass. Fast sigmoidal networks via spiking neurons. *Neural Computation*, 9:279–304, 1997.

[Maa97b] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural Networks*, 10:1659–1671, 1997.

[Maa99a] Wolfgang Maass. *Computing with spiking neurons*, pages 55–85. MIT Press, Cambridge, 1999.

[Maa99b] Wolfgang Maass. *Paradigms for computing with Spiking Neurons, Models of Neural Networks*, volume 4. Springer, Berlin, 1999.

[MAP+02] Rene Mayrhofer, Michael Affenzeller, Herbert Prähofer, Gerhard Höfer, and Alexander Fried. DEVS simulation of spiking neural networks. In Robert Trappl, editor, *Proceedings of the EMCSR 2002*, pages 573–578, 2002.

[MN97] Wolfgang Maass and Thomas Natschläger. Networks of spiking neurons can emulate arbitrary hopfield nets in temporal coding. *Network: Computation in Neural Systems*, 8(4):355–372, 1997.

[MN98] Wolfgang Maass and Thomas Natschläger. Emulation of hopfield networks with spiking neurons in temporal coding. In J. M. Bower, editor, *Computational Neuroscience: Trends in Research*, pages 221–226. Plenum Press, 1998.

[Moo96] Y. Moon. *High Performance Simulation Based Optimization Environment: Modelling Spatially Distributed Large Scale Ecosystems*. PhD thesis, The University of Arizona, Tucson, Arizona, 1996.

[MT75] M. D. Mesarovic and Y. Takahara. General systems theory: Mathematical foundations. *Mathematics in Science and Engineering*, 113, 1975.

[NR98] T. Natschläger and B. Ruf. Spatial and temporal pattern analysis via spiking neurons. *Network: Computation in Neural Systems*, 9(3):319–332, 1998.

[NT] Duane Q. Nykamp and Daniel Tranchina. Fast neural network simulations with population density methods.

[PH95] Yang J. Parekh, R. G. and V. Honavar. Constructive Neural Network Learning Algorithms for Multi-Category Pattern Classification. Technical report, Department of Computer Science, Iowa State University, Ames, Iowa, 1995.

[Pic98] Franz Pichler. Searching for Arthur Koestler's holons - a systemtheoretical perspective, 1998.

[Pic99] Franz Pichler. Holarchische System-Architekturen nach Arthur Koestler. In *Newsletter der Deutschen Gesellschaft für Systemforschung*, volume 8, pages 20–26. 1999.

[Pic00a] Franz Pichler. The CAST project: Experiences and future perspectives. *Lecture Notes in Computer Science*, pages 3–7, 2000.

[Pic00b] Franz Pichler. Notizblätter zur Lehrveranstaltung Systemtheorie 1, Ausgabe WS 2000/01, 2000.

[PS90] Franz Pichler and Heinz Schwärtzel. *CAST Computerunterstützte Systemtheorie: Aufbau und Anwendung von Systemtheorie-Methodenbanken*. Springer-Verlag, 1990.

[QC01] Gerardo Quero and Carolina Chang. Sequence learning in mobile robots using avalanche neural networks. In José Mira and Alberto Prieto, editors, *Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2001*, Lecture Notes in Computer Science, pages 508–515. Springer, 2001.

[RS98] Berthold Ruf and Michael Schmitt. Self-organization of spiking neurons using action potential timing. *IEEE Transactions on Neural Networks*, 9:3:575–578, 1998.

[Ruf98] Berthold Ruf. *Computing and Learning with Spiking Neurons - Theory and Simulations*. PhD thesis, Graz University of Technology, 1998.

[RWdRvSB97] Fred Rieke, David Warland, Rob de Ruyter von Steveninck, and William Bialek. *SPIKES - Exploring the Neural Code*. Bradford Books/MIT Press, Cambridge, MA, 1997.

[SBMC01] Eduardo Sánchez, Senén Barro, Jorge Mariño, and Antonio Canedo. A realistic computational model of the local circuitry of the cuneate nucleus. In José Mira and Alberto Prieto, editors, *Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2001*, Lecture Notes in Computer Science, pages 21–29. Springer, 2001.

[Sch97] R. R. Schaller. Moore's law: Past, present, and future. *IEEE Spectrum*, 43(6):52–59, 1997.

[SM01] Manuel A. Sánchez-Montanés. Strategies for the optimization of large scale networks of integrate and fire neurons. In José Mira and Alberto Prieto, editors, *Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2001*, Lecture Notes in Computer Science, pages 117–125. Springer, 2001.

[SMB01] Eduardo Sánchez, Manuel Mucientes, and Senén Barro. A cuneate-based network and its application as a spatio-temporal filter in mobile robotics. In José Mira and Alberto Prieto, editors, *Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2001*, Lecture Notes in Computer Science, pages 418–425. Springer, 2001.

[SO01] S. Snyders and C.W. Omlin. Inductive bias in recurrent neural networks. In José Mira and Alberto Prieto, editors, *Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks, IWANN 2001*, Lecture Notes in Computer Science, pages 339–346. Springer, 2001.

[TFM96] S.J. Thorpe, D. Fize, and C. Marlot. Speed of processing in the human visual system. *Nature*, 381:520–522, 1996.

[Tuc88] H.C. Tuckwell. *Introduction to Theoretical Neurobiology*, volume vol. 1 and 2. Cambridge University Press, Cambridge, 1988.

[Wat94] L. Watts. Event-driven simulation of networks of spiking neurons. *Advances in Neural Information Processing Systems*, 6:927–934, 1994.

[ZD97] B.P. Zeigler and D.Kim. Orders of magnitude speedup with DEVS representation and high-performance computation. In *SPIE Aerosense 97*, Orlando, FL, 1997.

[Zel94] Andreas Zell. *Simulation neuronaler Netze*. R. Oldenbourg Verlag München Wien, 1994.

[ZPK00] Bernhard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, second edition*. Academic Press, 2000.

[ZY96] B.P. Zeigler and Y.Moon. DEVS representation and aggregation of spatially distributed systems: Speed versus error tradeoffs. In *Trans of SCS*, volume 13(5), pages 179–190, 1996.

CURRICULUM VITAE

PERSONAL DATA

Name	Rene Michael Mayrhofer
Date of Birth	April 30th 1979
Place of Birth	Graz, Austria
Nationality	Austrian
Marital status	Single

EDUCATION

1985–1989	Elementary School: Volksschule Behamberg, Austria
1989–1993	Secondary School: Bundesrealgymnasium Steyr, Austria
1993–1998	Technical High School: HTBLA for Electrical Engineering and Computer Science, Steyr, Austria
1998–2002	Studies in Computer Science at the Johannes Kepler University of Linz, Austria

POSITIONS

July 1995	HTML design at Profactor GmbH Steyr
July 1996	System administration and installation of Oracle databases at RiS GmbH Steyr
August 1996	Migration of network infrastructure (Novell based to Microsoft Windows NT) at Profactor GmbH Steyr
July 1997	System and network administration at RiS GmbH Steyr
August–September 1997	OLE programming at Profactor GmbH Steyr
August 1998	Foundation of ViaNova DI Johannes Guger KEG
July–September 1998	Development of parts of CAPP Knowledge (SAP R/3 extension), customization of CAPP Knowledge for BMW GmbH Steyr
since March 1999	Development and project management of Gibraltar (Linux based firewall product) see http://www.gibraltar.at/
July–September 1999	Development of parts of CAPP Knowledge
February 2000	Development of parts of CAPP Knowledge
March–June 2001	Tutor at the University of Linz
June–September 2001	Working student at BMW, Munich (research and development center)

PUBLICATIONS

1. R. Mayrhofer, M. Affenzeller, H. Prähofer, G. Höfer, A. Fried
"DEVS Simulation of Spiking Neural Networks", *Sixteenth European Meeting on Cybernetics and Systems Research 2002*, pages 573–578, 2002.

Eidesstattliche Erklärung

Ich erkläre an Eides Statt, dass ich die vorliegende Diplomarbeit mit dem Titel "A New Approach to a Fast Simulation of Spiking Neural Networks" selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, 31. Juli 2002 Rene Mayrhofer