

---

■ < HEAD

■ > development

=====





Fakultät für Informatik

Professur für Technische Informatik

# Bachelorarbeit

Integration von Umwelt- und Sensormodellierung in die  
Netzwerksimulation

Thomas Rückert

Chemnitz, den 29. Mai 2015

**Prüfer:** Prof. Dr. Wolfram Hardt

**Betreuer:** Dipl.-Inf. Mirko Lippmann

**Thomas Rückert,**  
Integration von Umwelt- und Sensormodellierung in die Netzwerksimulation  
Bachelorarbeit, Fakultät für Informatik  
Technische Universität Chemnitz, Mai 2015

## **Abstract**

Mit dieser Arbeit soll die Simulation von Sensorknoten in einer Simulationsumgebung ermöglicht werden. Es gibt zahlreiche Umgebungen und Frameworks zum Simulieren von Netzwerken, jedoch werden in diesen lediglich die bloße Kommunikation zwischen den verschiedenen Sensor- oder Netzwerknoten untersucht. Daher soll nun die Simulationsumgebung Omnet++ und das Framework MiXiM genutzt werden und um eine Sensormodellierung erweitert werden. Zum einen sollen dafür bestehende Knoten um eine Sensorik erweitert werden. Diese soll wiederum auf generierte Umgebungsparameter zugreifen können. Es soll möglich sein ein großes Netzwerk von verschiedenen Sensorknoten, die untereinander kommunizieren, zu simulieren und dabei deren Verhalten und Energieverbrauch zu betrachten. Daher ist es ebenfalls notwendig, verschiedene statistische Werte über die Knoten bereitzustellen und diese anschließend zu visualisieren.



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>iii</b>
<b>Tabellenverzeichnis</b>	<b>v</b>
<b>1 Einleitung</b>	<b>1</b>
■< HEAD	
<b>2 Motivation</b>	<b>3</b>
<b>3 Vorberichtigungen</b>	<b>5</b>
3.1 Evaluation von Systemen . . . . .	5
3.2 Sensoren . . . . .	6
3.3 Sensorknoten . . . . .	10
3.4 Sensornetzwerke . . . . .	11
<b>4 Simulationsumgebung</b>	<b>13</b>
4.1 wichtige Simulationsumgebungen . . . . .	13
4.2 Vergleich . . . . .	15
4.3 Omnet++ . . . . .	16
4.3.1 Einleitung . . . . .	16
4.3.2 NED language . . . . .	17
4.3.3 Einige Techniken, Funktionen und wichtige Module . . . . .	20
4.4 MiXiM-Framework als Omnet++-Erweiterung . . . . .	27
4.4.1 Einleitung . . . . .	27
4.4.2 Einige wichtige Module . . . . .	27
<b>5 Implementierung</b>	<b>29</b>
5.1 Einleitung . . . . .	29
5.2 Aufbau und Struktur . . . . .	29
5.2.1 Klassenübersicht . . . . .	29
5.2.2 Übersicht NED-Module . . . . .	42
5.2.3 Simulationsparameter und omnetpp.ini . . . . .	51
5.3 Funktionsweise mit Beispielanwendungen . . . . .	53
<b>6 Zusammenfassung</b>	<b>61</b>

<b>Literatur- und Webverzeichnis</b>	<b>63</b>
=====	
<b>2 Vorbetrachtungen</b>	<b>3</b>
2.1 Evaluation von Systemen . . . . .	3
2.2 Sensoren . . . . .	4
2.3 Sensorknoten . . . . .	8
2.4 Sensornetzwerke . . . . .	9
<b>3 Simulationsumgebung</b>	<b>11</b>
3.1 wichtige Simulationsumgebungen . . . . .	11
3.2 Vergleich . . . . .	13
3.3 Omnet++ . . . . .	14
3.3.1 Einleitung . . . . .	14
3.3.2 NED language . . . . .	15
3.3.3 Einige Techniken, Funktionen und wichtige Module . . . . .	19
3.4 MiXiM-Framework als Omnet++-Erweiterung . . . . .	26
3.4.1 Einleitung . . . . .	26
3.4.2 Einige wichtige Module . . . . .	26
<b>4 Implementierung</b>	<b>29</b>
4.1 Einleitung . . . . .	29
4.2 Aufbau und Struktur . . . . .	29
4.2.1 Klassenübersicht . . . . .	29
4.2.2 Übersicht NED-Module . . . . .	42
4.2.3 Simulationsparameter und omnetpp.ini . . . . .	51
4.3 Funktionsweise mit Beispielanwendungen . . . . .	53
<b>5 Auswertung</b>	<b>61</b>
<b>6 Zusammenfassung</b>	<b>65</b>
<b>Literatur- und Webverzeichnis</b>	<b>67</b>

■> development

# Abbildungsverzeichnis

2.1	Abstraktes Sensor Modell . . . . .	5
2.2	Deutschlandkarte der mittleren Temperatur zwischen 1961 und 1990 . . . . .	7
2.3	Deutschlandkarte: Beispielverteilung der Luftfeuchtigkeit . . . . .	8
2.4	Abstraktes Sensorsnoten Modell . . . . .	10
3.1	GUI bei der Ausführung einer Simulation . . . . .	15
3.2	Oberfläche der Entwicklungsumgebung mit Beispiel für die NED-Integration .	17
3.3	Zustandswechsel Automat falsch . . . . .	23
3.4	Zustandswechsel Automat richtig . . . . .	23
3.5	Beispiel cMessage als Event . . . . .	24
4.1	Beispielstatisik Relativer Batterieladezustand . . . . .	38
4.2	SimpleClasses: Member . . . . .	40
4.3	ExtendedMessage: Vererbung . . . . .	41
4.4	allgemeines Modell eines Sensors . . . . .	46
4.5	Aufbau des Compoundmoduls SensorNode . . . . .	48
4.6	Die verschiedenen Events im Beispiel SensorExample . . . . .	55
4.7	Beispiel für den Nachrichtenverlauf beim Einlesen von Sensordaten .	56
4.8	Powermodi im Beispiel SleepVsNoSleep . . . . .	57
4.9	Ladezustand im Beispiel SleepVsNoSleep . . . . .	58
4.10	Powermodi im Beispiel SleepOutOfSync . . . . .	59

■< HEAD

5.1	Beispielstatisik Relativer Batterieladezustand . . . . .	38
5.2	SimpleClasses: Member . . . . .	41
5.3	ExtendedMessage: Vererbung . . . . .	42
5.4	allgemeines Modell eines Sensors . . . . .	46
5.5	Aufbau des Compoundmoduls SensorNode . . . . .	48
5.6	Die verschiedenen Events im Beispiel SensorExample . . . . .	55
5.7	Beispiel für den Nachrichtenverlauf beim Einlesen von Sensordaten .	56
5.8	Powermodi im Beispiel SleepVsNoSleep . . . . .	57
5.9	Ladezustand im Beispiel SleepVsNoSleep . . . . .	57
5.10	Powermodi im Beispiel SleepOutOfSync . . . . .	58

===== ■> development



## **Tabellenverzeichnis**

3.1	Übersicht Simulatoren . . . . .	14
3.2	NED Schlüsselbegriffe . . . . .	16
3.3	Übersicht über einige Funktionen von cMessage . . . . .	21
4.1	Klassenübersicht . . . . .	30



# 1 Einleitung

Die Verwendung von Sensoren steigt in der heutigen Zeit mehr und mehr an. Sensorknoten unterscheiden sich im Kern nicht von herkömmlichen Computern und sind zusätzlich mit Sensoren und oft mit Batterien und Funkmodulen ausgestattet. Da Computerbauteile bei gleicher Leistung immer kleiner werden, ist es nicht verwunderlich dass auch Sensorknoten immer kleiner werden. Mit diesen kleinen Knoten ist es möglich ganze Netzwerke von Sensoren zu erschaffen, die miteinander kommunizieren. So können beispielsweise die Umgebungsparameter großer Naturflächen detailliert untersucht werden, ohne dass eine große Forschungsstation aufgebaut werden müsste. Stattdessen kann man viele kleine Sensorknoten in der Umwelt verteilen, die miteinander in Kontakt stehen.

**Motivation** Diese Sensorknoten werden sinnvoller Weise in drahtlosen Sensornetzwerken organisiert. Das Netzwerk dient dabei dem Sammeln von Daten, also dem Senden von den gemessenen Daten zu einer Datensenke. Da der Energiehaushalt bei Sensorknoten meist sehr sensibel ist, kommt neben einem Funktransceiver oft auch ein Wake-up-Receiver zum Einsatz.

Diese Netze müssen vor dem Praxiseinsatz getestet werden, wobei die Betrachtung des Energiezustands über lange Zeiträume bei verschiedenen Routingverfahren von Bedeutung ist und wie sich die Knoten in großen Netzen von tausenden Knoten verhalten.

Das Ziel dieser Arbeit ist es, neben der Simulation von den Netzwerken das Verhalten der Sensorik abbilden zu können. Diesen sollen Umgebungsparameter bereitgestellt werden. Dabei ist wiederum die Betrachtung des Energiehaushalts von entscheidender Bedeutung. Für Nutzung großer Netzwerke sollen Möglichkeiten bereitgestellt werden, die statistischen Daten von den Simulationen zu visualisieren, damit diese bestmöglich ausgewertet werden können.

Zuletzt soll anhand von Tests und Beispielanwendungen gezeigt werden, wie die Implementierung der Sensormodelle genutzt werden könnte.



## 2 Vorberichtigungen

Im Folgenden werden die verwendeten Technologien betrachtet. Als Versionsverwaltungssoftware wurde Git[4] auf der Plattform Github[5] verwendet, worauf nicht weiter eingegangen wird. Zum Erstellen der Simulation wurde Omnet++[12] mithilfe des MiXiM-Frameworks[8] benutzt.

### 2.1 Evaluation von Systemen

Im Entwicklungsprozess eines jeden Systems müssen neue Teile oder Module evaluiert werden. Für das Testen gibt es verschiedene Möglichkeiten, die einem zur Verfügung stehen. Im frühen Stadium der Entwicklung bietet das Abschätzen von gewissen Parametern eine kostengünstige Variante zum Bestimmen von Designparametern. Das kann noch vor einer ersten Implementierung und daher auch ohne Messungen durchgeführt werden. Allerdings sind diese Ergebnisse oftmals sehr grob und es lässt sich auch nicht jeder Wert so einfach bestimmen.

Es ist daher notwendig genauere Tests durchzuführen. Wenn man ein komplett implementiertes und produziertes System anschließend testen möchte, kann das sehr teuer werden, sollten viele Fehler auftreten oder schlimmer, wenn man zum Beispiel merkt, dass das gewählte Modell das gewünschte Ziel nicht befriedigend erfüllen kann und daher viel bisherige Arbeit verworfen werden müsste.

Man kann diesen Problemen zuvor kommen, indem man noch vor der ersten Implementierung eines Systems Simulationen und Emulationen erstellt und Prototypen anfertigt. Das senkt die Kosten mitunter erheblich und es lassen sich beinahe alle Parameter des zukünftigen Produkts überprüfen, auch wenn es das Testen des fertigen Produkts nicht komplett ersetzen kann.

**Simulation** Eine Simulation ist ein Modell eines Systems, welches dieses passend abbildet. Mit diesem Modell kann herausgefunden werden, was im realen System später umsetzbar ist. Der Zustand eines solchen Modells ändert sich im Laufe der (Simulations-)Zeit. Daher führt das Modell Zustandsübergänge durch, welche als Events bezeichnet werden. Man kann Systeme nach den Zeitpunkten, an denen Zustandswechsel möglich sind, in analoge und diskrete unterteilen. Wie der Name vermuten lässt, können im analogen Fall zu jeder Zeit Zustandsübergänge stattfinden,

im diskreten dagegen nur zu bestimmten Zeitpunkten.

Eine Simulation eignet sich zu jedem Zeitpunkt des Produktentwurfs sehr gut. Große Vorteile sind, dass auf diese Weise schon sehr früh getestet werden kann und sich Simulationen sehr kostengünstig erstellen lassen, da keine besondere Hardware notwendig ist.

**Emulationen** Eine Emulation ist die Implementierung eines Systems, welche den kompletten Funktionsumfang des Entwurfs abdeckt. Diese kann mit einer Hardwarebeschreibungssprache wie VHDL definiert werden und auf einem FPGA oder innerhalb eines Netzwerks von FPGAs ausgeführt werden. Man spricht daher von einer homogenen Hardwareplattform.

**(Rapid) Prototyping** Ein Prototyp ist ebenfalls eine Implementierung eines Systems, die den kompletten Funktionsumfang des Entwurfs abdeckt, allerdings geringere Anforderungen und Timing, Größe und Kosten stellt. Prototypen sind zum Beispiel oftmals wesentlich größer als das Endprodukt. Wenn er alle sonstigen Anforderungen zur Genüge erfüllt, so kann er in das finale Design umgewandelt werden und verliert bei diesem Prozess alle Grenzen des Prototyps. Im Gegensatz zur Emulation kommt eine heterogene Hardwareplattform zum Einsatz. So können etwa fertige Prozessoren, Speicher, weiterhin FPGAs oder spezielle Chips wie ein ASIC zum Einsatz kommen.

Für die Arbeit ist die Entscheidung auf eine Simulation gefallen. Man kann sich dabei auf die wesentlichen Funktionen konzentrieren und grundsätzliche Überlegungen über die genaue Umsetzung von gewissen Bauteilen zunächst außer Acht lassen. So ist die genaue Implementierung der einzelnen Komponenten des Sensors für diese Arbeit beispielsweise nicht so relevant, da eher das gesamte Verhalten der Sensorknoten und deren Energiemanagement von Bedeutung sind. Ein Modell muss stets nur so genau spezifiziert werden wie nötig und so gut wie nie mit der Komplexität der Realität übereinstimmen.

Der Fokus der Arbeit liegt daher auch auf dem Verwalten von Umweltparametern, der Erfassung dieser durch Sensoren auf vielen verschiedenen Sensorknoten. Wie genau die einzelnen Bauteile technisch aufgebaut sind, ist für die theoretische Betrachtung natürlich von Bedeutung, soll jedoch nicht bis ins kleinste Detail simuliert werden.

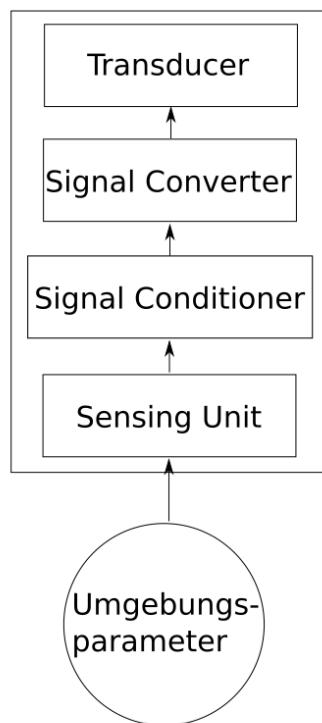
### 2.2 Sensoren

Den Hauptgegenstand in der Simulation bilden Sensoren, welche auf Sensorknoten angebracht sind. Ein solcher Knoten kann dann wiederum ein oder mehrere Sensoren

besitzen.

Sensoren sind das technische Gegenstück zu den menschlichen Sinnen, denn sie können physikalische oder chemische Eigenschaften wahrnehmen. Dabei arbeiten Sensoren allerdings noch wesentlich genauer, denn es lassen sich Messgrößen quantitativ exakt bestimmen.

Abbildung 2.1: Abstraktes Sensor Modell



Allgemein ist ein Sensor wie in Abbildung 2.1 aufgebaut. Die Sensing Unit, zu deutsch Aufnehmer, ist dabei das Herzstück, denn es ist das Bauteil, in dem der gesuchte Wert aus der Umgebung aufgenommen wird. Je nach Sensor ist dieser Wert jedoch nicht direkt interpretierbar. Das Signal wird daher zunächst aufbereitet. Zum einen kann das bedeuten, falls der Messimpuls nur sehr kurz war, diesen zu verlängern, sodass folgende Bauteile verwertbare Eingaben bekommen können. Zum anderen kann es nötig sein, dass ein analoges in ein digitales Signal umgewandelt werden muss. Diese Aufgaben werden durch Signalformer und Signalwandler übernommen.

Zuletzt muss das Signal noch durch einen Messumformer aus einem bloßen digitalen Wert ohne direkte Bedeutung in einen Messwert in der benötigten Messeinheit umge-

wandelt werden. Dieser Wert kann dann über eine Schnittstelle nach außen gegeben werden und von anderen Teilen auf einem Sensorknoten verarbeitet werden.

### Beispiele für Sensoren

Im folgenden Abschnitt werden ein paar verschiedene Beispiele für Sensoren vorgestellt. Es wird für die in der Implementierung umgesetzten Sensoren für Temperatur, Helligkeit, Luftdruck und Luftfeuchtigkeit jeweils ein Beispiel erläutert.

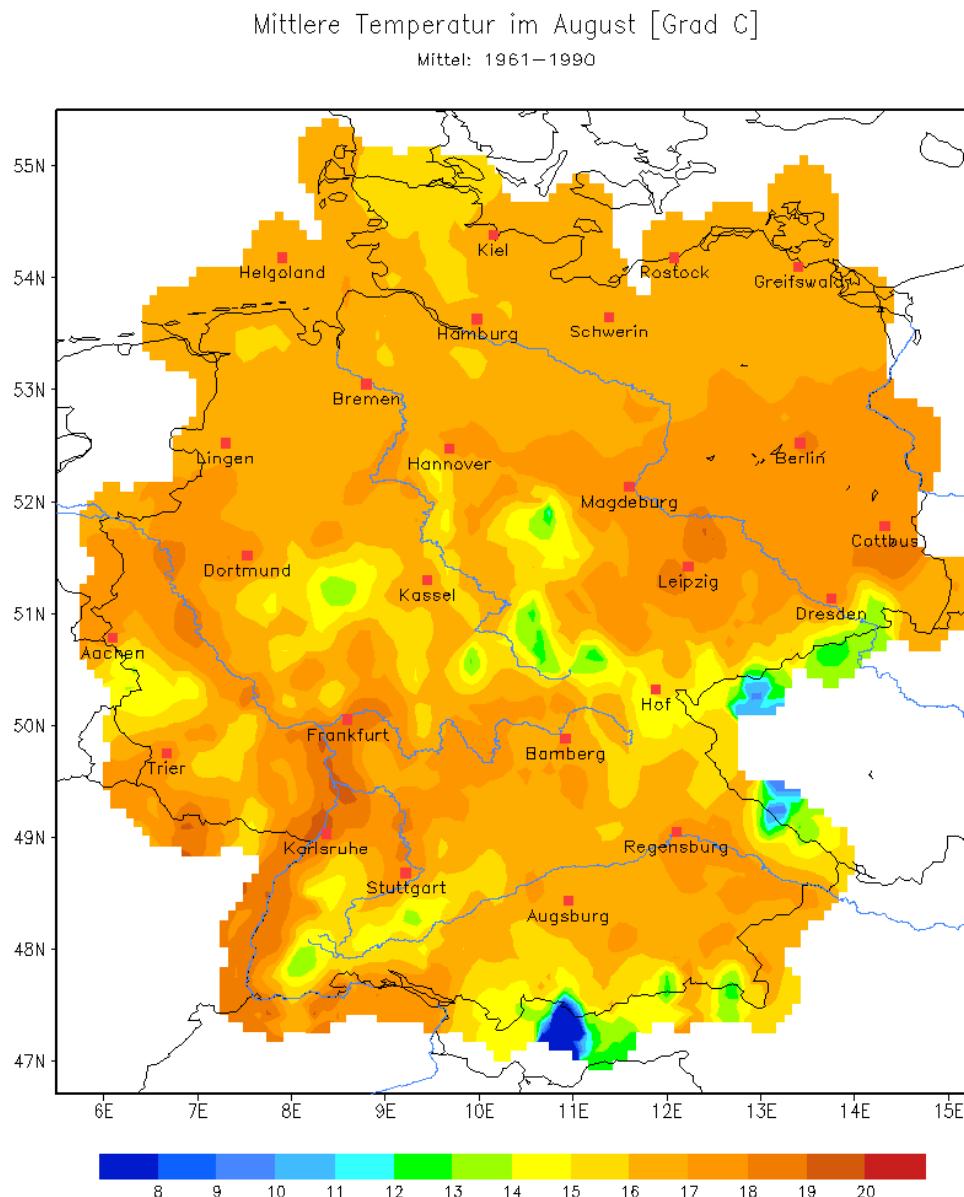
**Temperatur** Vermutlich bedingt durch die große Verbreitung und die bereits lange Existenz von Temperaturlühlern gibt es eine sehr hohe Zahl verschiedener Möglichkeiten Temperatur zu messen, wobei sich die verschiedenen Umsetzungen zum Teil auch sehr stark unterscheiden. Dabei gibt es sowohl Sensoren mit passiven als auch mit aktiven Aufnehmern.

Ein Beispiel für eine Umsetzung ist ein Temperatursensor mit Aufnehmer in Form eines Heißleiters. Dieser verringert seinen Widerstand, wenn seine Temperatur steigt. Es kann nun eine stets konstante Spannung an den Heißleiter angelegt werden, um auf die Temperatur des Bauteils zu schließen. Da zum Abgreifen der Messwerte eine Spannung von außen angelegt werden muss, liegt ein passiver Aufnehmer vor.

**Helligkeit** Ein Lichtsensor oder auch Fotodetektor dient dazu, um die Stärke des einfallenden Lichts zu bestimmen, also die Helligkeit. Diese können unter Nutzung des fotoelektrischen Effekts ganz ähnlich zum vorgestellten Temperatursensor mit Heißleiter genutzt werden. Die sogenannte Fotoleitung bezeichnet hierbei die Zunahme der Leitfähigkeit durch steigende Bestrahlung. Um den Sensor mit einem aktiven Aufnehmer zu nutzen, kann ein Sensor auch mit dem photovoltaischen Effekt gebaut werden. Dieser wird ebenfalls bei der Gewinnung von elektrischer Energie durch das Sonnenlicht benutzt. Je stärker die einfallende Sonnenenergie auf der Photovoltaikfläche ist, um so mehr Energie wird erzeugt und um so größer ist die Helligkeit.

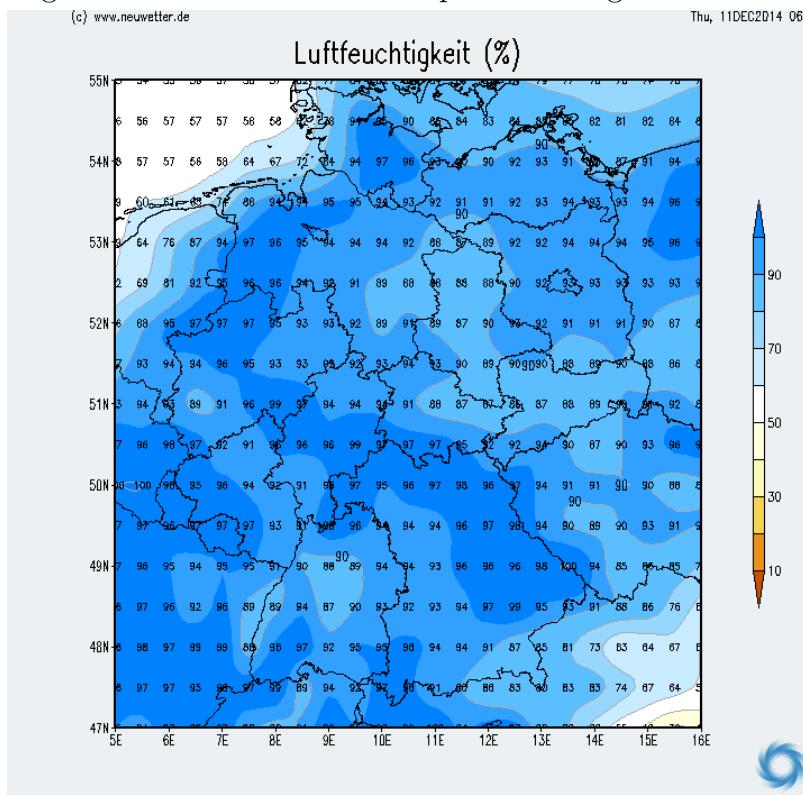
**Luftdruck** Auch von Drucksensoren gibt es sehr viele verschiedene Arten, welche auf unterschiedliche Weise funktionieren. Dabei gibt es Sensoren, welche lediglich Messwertänderungen feststellen können und andere, die wiederum auch statische Werte ermitteln können. Ein Beispiel für die zweite Variante ist ein piezoresistiver Drucksensor. Bei diesem befindet sich im Aufnehmer eine Membran, deren Position sich durch unterschiedlich hohen Druck verändert. Um diese Bewegung nun messbar zu machen, werden auf der Membran Widerstände aufgebracht.

Abbildung 2.2: Deutschlandkarte der mittleren Temperatur zwischen 1961 und 1990



**Luftfeuchtigkeit** Sensoren zur Bestimmung der Luftfeuchtigkeit werden auch Hygrometer genannt. Eine Variante dieser ist ein Absorptionshygrometer. Dabei befindet sich eine Schicht zwischen 2 Elektroden, welche die Feuchtigkeit der Umgebung gut aufnehmen kann. Diese Schicht wird als hygrokopische Schicht bezeichnet. Wenn nun eine Spannung angelegt wird, dann ist der Widerstand dieser Schicht abhängig von der Feuchtigkeit und somit kann wiederum indirekt auf diese geschlossen werden.

Abbildung 2.3: Deutschlandkarte: Beispielverteilung der Luftfeuchtigkeit



## 2.3 Sensorknoten

Sensorknoten gibt es natürlich in verschiedenen Formen und Größen, doch gewisse Grundelemente sind in jedem Sensorknoten gleich oder wenigstens ähnlich. In Abbildung 2.4 ist ein grober Aufbau beschrieben. In der heutigen Zeit befinden sich dabei alle Bauteile auf einem einzigen Chip, man spricht daher von einem System-on-a-Chip.

Natürlich ist das wohl charakteristischste Bauteil, denn es macht einen Netzwerkknoten zum Sensorknoten, ein Sensor. Es ist ebenso möglich, dass ein Sensorknoten viele verschiedene Sensoren besitzt, um an der Position mehrere verschiedene Parameter

aufnehmen zu können.

Um aber mit den Messwerten etwas anfangen zu können, sind auch andere Bauteile essenziell.

**Prozessor** Ein Prozessor, typischerweise auf einem Mikrocontroller, muss sich dabei um die Steuerung der anderen Bauteile kümmern. Er kann verschiedenen Powermodi gezielt einsetzen, um in gewissen Zeitintervallen Strom zu sparen. Außerdem initiiert er Sensormessungen und kann die Messwerte hinterher verschieden behandeln, beispielsweise durch Abspeichern in den Memory oder durch direktes Versenden der Daten über das Funkmodul.

**Funkmodul** Ein Funkmodul ermöglicht dem Knoten die Kommunikation mit Anderen in seiner erreichbaren Umgebung. Nur dadurch ist es möglich in großflächigen Netzwerken die Daten auswerten zu können, da es in manchen Gebieten sogar unmöglich sein kann die Knoten nach dem verstreuen, noch direkt zu erreichen.

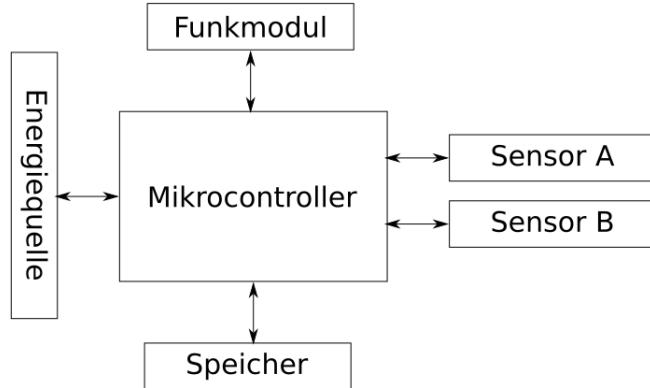
**Speicher** Ein Speicher sollte ebenfalls auf einem Knoten vorhanden sein. Zum einen müssen natürlich Programme, die der Prozessor ausführt, gespeichert werden. Ebenso ist es denkbar, dass die Funkkommunikation gelegentlich für längere Zeit abgeschaltet wird, um so Energie zu sparen. Wenn innerhalb dieses Zeitraums relevante Messwerte bestimmt wurden, so sollen diese natürlich nicht einfach verfallen, sondern bis zur nächsten Funkverbindung gesichert werden.

**Energiequelle** Eine Energiequelle, in verschiedenen möglichen Varianten, ist für einen Sensorknoten ebenfalls entscheidend, damit dieser funktionieren kann. Dazu gehört typischerweise eine Batterie. Weiterhin ist es denkbar, durch energy harvesting zusätzlich Energie zu gewinnen. Da dies aber nicht unbedingt rund um die Uhr möglich ist, muss auch in diesem Fall die gewonnene Energie zunächst gespeichert werden.

## 2.4 Sensornetzwerke

Da ein einzelner Sensorknoten nur sehr begrenzt einsetzbar ist, werden die immer kleiner und mobiler werdenden Knoten oft zu großen Netzwerken verbunden. Dadurch ist es möglich, große Flächen mit den Messwerkzeugen abzudecken. Diese Tatsache erlaubt es, detaillierte Daten von einem Gebieten zu erhalten.

Abbildung 2.4: Abstraktes Sensorsnoten Modell



Bei einem sehr großen Netzwerk ist ein gutes Routing sehr wichtig, besonders da die Knoten sehr abhängig von der effizienten Nutzung der Energiequelle sind. Für das Übertragungsprotokoll kommt dabei meist der Standard IEEE 802.15.4 zum Einsatz, da dieser für drahtlose Kommunikation mit niedriger Übertragungsraten und für Geräte mit geringer Leistungsaufnahme ausgelegt ist. Es deckt lediglich die unteren beiden Schichten des OSI-Modells ab.

Die Verbindung zu Anwendungsschicht stellt dann beispielsweise das ZigBee-Framework dar.

**Wach-und-Schlaf-Zyklen** Da Batteriebetriebene Technik stets nur begrenzte Lebenszeit aufweist, ist es notwendig den Energieverbrauch so weit wie möglich zu senken. In Sensornetzwerken wird daher oftmals ein fester Wach-und-Schlaf-Zyklus für die Sensorknoten definiert. Wenn alle Knoten innerhalb des Netzwerkes gleichzeitig in einen Ruhemodus wechseln, in dem keine Kommunikation stattfindet, kann viel Energie gespart werden, ohne dass Nachrichten verloren gehen.

Wiederum können auch Zeitfenster definiert werden, in denen alle Knoten gegenseitig Nachrichten austauschen, um die Messwerte an eine Datensenke zu schicken, in der später von außen eine Auswertung dieser Werte stattfinden kann.

Wenn dieser Wechsel zwischen aktiven und inaktiven Phasen zeitlich synchron im ganzen Netzwerk erfolgt, kann erheblich Energie gespart werden, ohne dass dabei Informationen verloren gehen müssen.

## 3 Simulationsumgebung

Um eine Simulation in einem geeigneten zeitlichen Rahmen erstellen zu können, bietet sich die Verwendung von einer Simulationsumgebung an. Diese enthält hauptsächlich ein Framework mit vielen Bibliotheken, welche für den speziellen Anwendungsfall viel Arbeit ersparen können.

Es gibt eine große Auswahl an verschiedenen Simulationsumgebungen, wobei jede ihre Vor- und Nachteile mit sich bringt. Wichtig für die Arbeit ist ein Simulator der Netzwerke bereitstellt und auch drahtlose Kommunikation ermöglicht. Es sollten eigene Knoten erstellt und angepasst werden können, damit die Hardwaresensorik und das Verhalten im Umgang mit Nachrichten untereinander genau definiert werden können. Auch Funktionen für die Repräsentation von Batterieeigenschaften und die Erfassung und Analyse von Statistiken sollten vorhanden sein.

Im Falle von speziellen Anforderungen der Frameworks sollte eine IDE vorhanden sein, die diese unterstützt. Auch eine sehr wichtige Anforderung an den Simulator ist eine grafische Umgebung für die Simulation selbst, sodass Informationen nicht nur aus Dateien ausgelesen werden können, sondern für den Nutzer auch auf den ersten Blick sichtbar sind.

### 3.1 wichtige Simulationsumgebungen

Es gibt viele Umgebungen zum Simulieren von Netzwerken. Zunächst werden hier 4 Wichtige vorgestellt: die **IKR Simulation Library (IKR SimLib)** von der Universität Stuttgart, der **Open Source Wireless Network Simulator** kurz **openWNS** von der Universität Aachen, **ns-3** vom ns-3 project und **Simanet** von der TU Chemnitz.

**Simulation Library (IKR SimLib)[19]** Eine freie Simulationsbibliothek unter der Lizenz GNU LGPL für Kommunikationsnetzwerke in C++ und Java von der Universität Stuttgart. Diese steht für Linux und Unix(-artige) Systeme zur Verfügung, während die Verwendung unter Windows nicht offiziell getestet wurde.

Da es lediglich eine Bibliothek für Java darstellt, existieren keine extra IDE und auch keinerlei GUI oder ähnliche Hilfswerkzeuge. Allerdings kann ein Projekt mit jeder normalen Java- oder C++ IDE benutzt werden, schließlich muss nur die Bi-

bibliothek importiert werden.

Da schon seit den 1980ern an IKR SimLib entwickelt wird, kann die Bibliothek weitreichende Funktionalitäten zur Verfügung stellen, wie zum Beispiel Unterstützung für mobile, IP- oder P2P-Netzwerke.

**openWNS[16]** OpenWNS ist ein Simulator für kabellose Kommunikation, entwickelt von der Universität Aachen. Das Projekt steht kostenlos zur Nutzung bereit und ist mit der GPLv2 lizenziert.

Es wurde speziell für Linux entwickelt, es läuft allerdings auch unter Windows. Die Entwicklung erfolgt in Python. Für grafische Unterstützung sorgt das integrierte Tool Wrowser[13], welches in erster Linie dazu dient die Resultate der Simulation zu sammeln und die Messwerte zu visualisieren, aber auch beim Erstellen einer Simulation Hilfestellungen bietet, wie beispielsweise beim Einrichten der Simulationsdatenbank. Es gibt keine extra für openWNS erstellte oder angepasste IDE, allerdings wird in der Dokumentation beschrieben, wie der Texteditor Emacs[3] den speziellen Anforderungen vom openWNS-Stil angepasst werden kann. Allerdings ist es notwendig für die Entwicklung stets einen Texteditor, den Wrowser und die Kommandozeile zu benutzen, was im Vergleich zu einer alles umfassenden IDE deutlich weniger komfortabel ist.

**NS-3[18]** ns-3 ist ein freier Netzwerksimulator, der unter der GPLv2 lizenziert ist. Das Erstellen der Simulationen funktioniert mithilfe der Sprache Python. Dafür steht keine extra IDE zur Verfügung, was allerdings auch nicht notwendig ist, da es genügend andere Python-Umgebungen gibt.

Es werden sowohl kabelgebundene als auch kabellose Verbindungen unterstützt. Allerdings besteht nicht für alle wichtigen Protokolle eine Unterstützung, wie beispielsweise WSN.

Ein weiterer großer Nachteil ist, dass es keine Oberfläche während der Ausführung gibt. Die Simulation wird per Textausgabe auf der Kommandozeile ausgeführt. Die gesammelten Daten können anschließend mit Plot-Tools visualisiert werden. Für das Generieren von Netzwerken steht jedoch ein grafisches Hilfsmittel zur Verfügung. Mit dem Topology Generator[14] können per GUI Netzwerke angelegt werden und anschließend in C++ oder Python-Code umgewandelt werden.

**Simanet[20]** Simanet ist ein Projekt, welches an der TU Chemnitz entstanden ist. Es eignet sich sehr gut für das Simulieren von Funkkommunikation und daher zum Testen von Routingverfahren.

Im Vergleich zu anderen Simulatoren wie Omnet++ oder ns-3 ist Simanet sehr leichtgewichtig und eignet sich daher hervorragend für sehr große Simulationen.

Es gibt zahlreiche vorhandene Module, die wichtige Funkstandards wie WLAN oder ZigBee, Energiemodelle, Lokalisierung und auch Visualisierung und Auswertung abdecken.

**Omnet++[12]** Omnet++ ist ein Framework für die Simulation von Netzwerken, welches unter der ACADEMIC PUBLIC LICENSE steht. Die Nutzung ist daher kostenfrei und der Quellcode ist offen und darf verändert werden.

Es bietet eine angepasste IDE, welche die Besonderheiten der NED-Sprache und auch den C++-Code interpretieren kann.

Die Simulation selbst bietet ein grafisches Interface und auch für die Auswertung von statistischen Daten stehen grafische Hilfsmittel innerhalb der IDE bereit.

Der größte Vorteil von Omnet++ ist die Möglichkeit, Code auf dem Application Layer der Sensorknoten ausführen zu können, was bei vielen andern Simulatoren nicht möglich ist.

Funkkommunikation ist standardmäßig in Omnet++ zwar nicht möglich, aber durch das MiXiM-Framework wird diese implementiert.

**weitere** Es gibt natürlich außer den bisher genannten Simulationsumgebungen auch noch weitere, auf die hier aber nicht näher eingegangen werden soll. So gibt es beispielsweise GloMoSim, welches die parallele Programmiersprache Parsec benutzt. Es ist geeignet um sowohl kabellose, als auch -gebundene Netzwerke zu simulieren, jedoch wird es zum aktuellen Stand nicht mehr weiterentwickelt.

Eine weitere Alternative wäre NetSim, welches von Tetcos, zusammen mit dem Indian Institute of Science entwickelt wurde. Die bestehenden Bibliotheken sind in C geschrieben und implementieren viele Protokolle wie beispielsweise WLAN, TCP oder LTE.

## 3.2 Vergleich

Simanet unterstützt leider keine komplexe Codeausführung innerhalb von Netzwerk-knoten. Es ist daher für diese Arbeit nicht geeignet, da das Verhalten der Sensorik nicht abgebildet werden kann. Daher wird Simanet im folgenden Vergleich nicht weiter berücksichtigt.

In Tabelle 3.1 ist ein direkter Vergleich der größten Unterschiede zwischen den Simulatoren Omnet++, IKR SimLib, OpenWNS und NS-3 aufgezeigt. Es ist sehr positiv zu bewerten, dass alle Simulationsumgebungen über freie Lizenzen verfügen und die für die Arbeit relevante drahtlose Kommunikation ermöglichen.

Tabelle 3.1: Übersicht Simulatoren

	Omnnet++	IKR SimLib	OpenWNS	NS-3
freie Lizenz	✓	✓(LGPL)	✓(GPLv2)	✓(GPLv2)
alle gängigen Betriebssysteme	✓	kein Windows	✓	kein Windows
GUI bei Simulation	✓	✗	(✓)	✗
IDE	✓	(✗)	(✗)	(✓)
Drahtlose Verb.	(✓) mit MiXiM	✓	✓	✓
Sprache(n)	C++ mit NED	C++ oder Java	Python	Python

Der größte Nachteil der Simulatoren IKR SimLib und ns-Simulator liegt darin, dass beide keine grafische Umgebung für die Simulation selbst bieten. Das wirkt sich wiederum natürlich positiv auf die Performanz aus, allerdings lässt sich über eine grafische Übersicht deutlich leichter und schneller ein Eindruck über die Zustände der Simulation gewinnen. Ein weiterer Nachteil der beiden ist die eingeschränkte Verfügbarkeit auf verschiedenen Betriebssystemen.

OpenWNS bietet nur teilweise Unterstützung in grafischer Hinsicht. Es ist möglich mithilfe des sogenannten Wrowser die Simulation zu starten und die statistischen Daten zu erfassen. Jedoch bietet es keine so umfassende Oberfläche wie Omnet++. Letztendlich ist die Entscheidung zugunsten von Omnet++ gefallen. Die Simulationsumgebung wird im folgenden Abschnitt ausführlicher beschrieben.

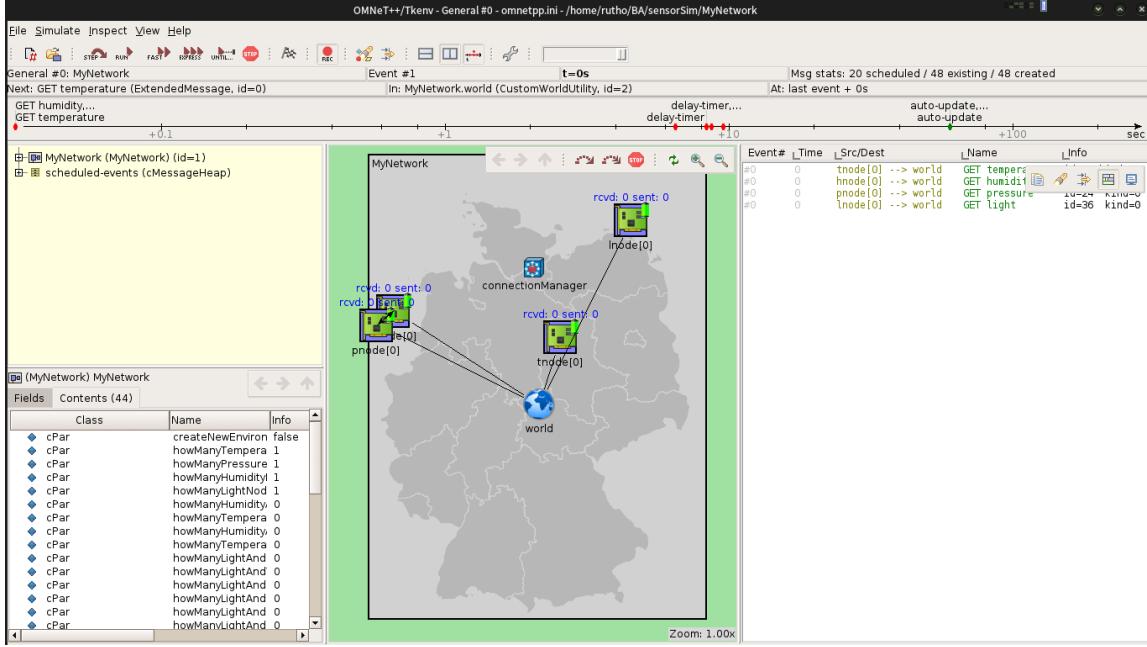
### 3.3 Omnet++

#### 3.3.1 Einleitung

Omnnet++[12] ist eine C++-Bibliothek und ein C++-Framework, welches primär zum Simulieren von Netzwerken dient. Außerdem bietet es eine Netzwerkbeschreibungssprache namens NED (NEtwork Description) und eine auf Eclipse[2] basierende Entwicklungsumgebung. Für die Simulation besteht außerdem ein grafisches Interface, mit dem die Kommunikation der Knoten im Netzwerk gut verfolgt werden kann. Standardmäßig werden keine mobilen oder kabellosen Protokolle in Omnet++ unterstützt. Jedoch kann mithilfe des MiXiM-Frameworks die Funktionalität um eben diese erweitert werden.

Es bietet unterstützt alle gängigen Betriebssysteme wie Linux, andere unixbasierte Systeme, Mac OS und Windows und besitzt außerdem eine kostenlose Lizenz.

Abbildung 3.1: GUI bei der Ausführung einer Simulation



### 3.3.2 NED language

Die Netzwerkbeschreibungssprache NED[9] bietet eine Möglichkeit, auch komplexe Netzwerke relativ einfach zu beschreiben und darzustellen. Man kann schnell ein einfaches Modul mit Gates (siehe Listing 3.3) für die Kommunikation beschreiben oder ihm Submodule für verschiedene andere Aufgaben zuweisen und dieses in ein Netzwerk integrieren und dort mehrere und auch verschiedene Instanzen von Modulen verknüpfen (siehe Listing 3.4).

Dabei helfen die verschiedenen möglichen Module. Es können die 3 Typen simple, module und network definiert werden. Wie der Name schon sagt, ist network dazu da ein Netzwerk zu beschreiben und sollte alle nötigen Module als Submodule beinhalten.

Mit dem Schlüsselwort module lassen sich komplexe Objekte beschreiben. Neben den Standardvariablen wie beispielsweise Parameter, Gates oder Connections lassen sich auch Submodule definieren. Dadurch ist es möglich verschiedene in sich abgeschlossene Modulteile in einem großen Modul zu vereinen.

Geeignet als ein solches Modulteil ist wiederum das simple-Modul. Dieses kann keine weiteren Submodule besitzen, sondern lediglich einfache Funktionalität definieren.

Wenn nicht anders über den Parameter @class angegeben sucht **Omnet++** nach einer Klasse, die den gleichen Namen wie das erstellte Modul besitzt. In dieser können Funktionen deklariert und implementiert werden, die das Verhalten des Moduls

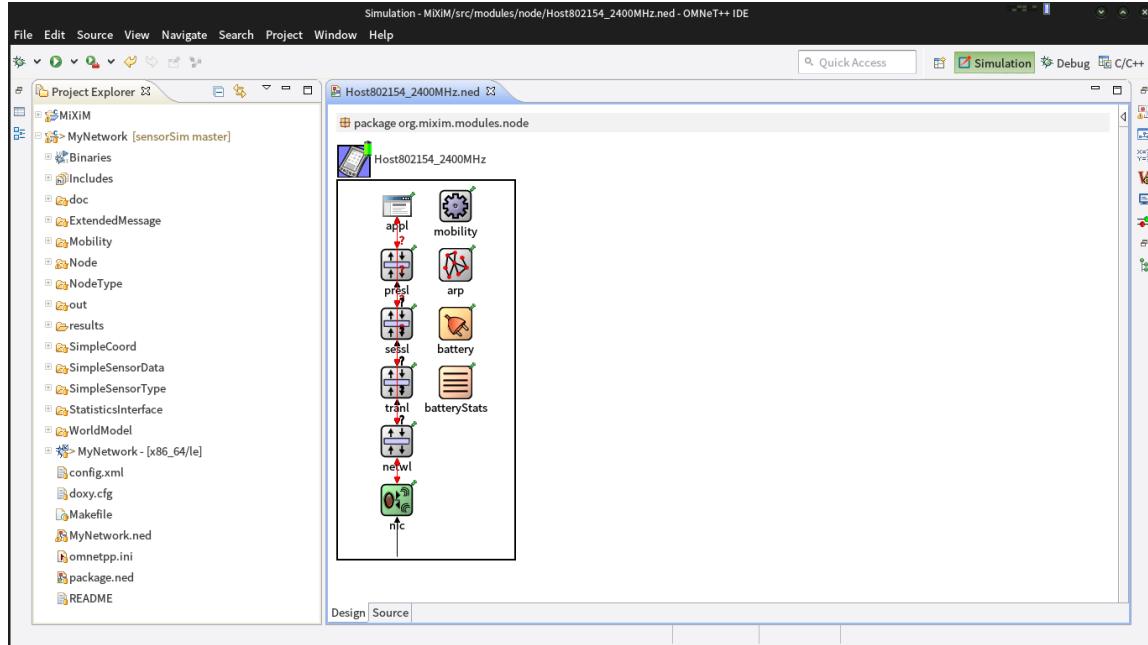
beeinflusst. Welche Funktionen von **Omnet++** interpretiert werden, wird im Kapitel 3.3.3.1 näher erklärt. Eine Übersicht mit Kurzbeschreibung zu den wichtigsten Schlüsselwörtern in NED in der Tabelle 3.2 zu finden.

Tabelle 3.2: NED Schlüsselbegriffe

Kategorie	Begriffe	Funktion
Modulart Sections von Modulen	network	ein Netzwerk, 1 pro Simulation
	simple	ein einfaches, eigenständiges Modul
	module	ein compound Modul, kann Submodule haben
	channel	beschreibt eine Verbindung zwischen Gates
	channelinterface	ein Interface für Channel
	moduleinterface	ein Interface für Module
Typen	types	um eigene Typen im Modul zu definieren
	parameters	Parameter des Moduls definieren
	submodules	andere Module integrieren
	gates	Schnittstellen für Kommunikation definieren
	connections	Gates miteinander verbinden
Verbindungen	int, string, double, bool	
	xmldoc	speichert den Pfad eines XML-Files
	xml	speichert XML
	allowunconnected	erlaubt Kommunikation zwischen Gates ohne definierte Verbindung
weitere	input	definiert ein Gate als eingehend
	output	definiert ein Gate als ausgehend
	inout	legt je ein in- und output Gate an
	@display	Eigenschaften zur Darstellung bei Sim.
	@class	spezielle C++-Klasse zum Modul definieren
	package	definiert den Namensraum
	import	andere Pakete und Module einbinden
	volatile, const, extends, import, like	
	this, false, true, default, if, and, or, else, for	
	index, sizeof, typename	

**Beispiel** Ein kurzes Beispiel soll das Zusammenspiel einiger wichtiger Schlüsselworte aufzeigen:

Abbildung 3.2: Oberfläche der Entwicklungsumgebung mit Beispiel für die NED-Integration



Zunächst wird ein einfaches Modul (Listing 3.1) erstellt, welches eine LED repräsentiert. Dieses erhält dafür einen Parameter vom Typ boolean, welcher anzeigt, ob die LED leuchtet und einen eingehenden Port, welcher die Lampe steuern kann.

Listing 3.1: einfaches Modul: LED

```
simple LED
{
    parameters:
        bool ledLeuchtet;
    gates:
        input control;
}
```

Als zweites Modul, ebenso wie die LED vom Typ simple, wird ein einfacher Knopf (Listing 3.2) definiert. Dieser hat ein input-Gate über welches einen Knopfdruck simulieren kann und ein ausgehendes Gate, welches im Falle eines Knopfdrucks ein Signal ausgeben kann.

Listing 3.2: einfaches Modul: Knopf

```
simple Knopf
{
    gates:
```

```

        input buttonStateChange;
        output signal;
}

```

Als drittes wird ein Modul vom Typ module definiert (Listing 3.3). Es handelt sich dabei um ein Compound-Modul. Dieses kann genutzt werden, um verschiedene simple-Module auf sich zu vereinen und die Interaktion zwischen diesen zu ermöglichen. So können komplexe Bauteile modular definiert werden.

Das Modul repräsentiert einen Bauteil, welches einen Knopf und eine LED besitzt. Zusätzlich besitzt es ein eingehendes und ein ausgehendes Gate. Das output-Gate vom Knopf ist mit dem eingehenden Gate von der LED verbunden, den durch einen Knopfdruck soll die LED ein- oder ausgeschaltet werden können.

Listing 3.3: Compound Modul

```

module Knoten
{
    parameters:
        string name = "KnotenMitLedUndKnopf";
    submodules:
        Blinker: LED {
        }
        Button: Knopf {
        }
    gates:
        input in;
        output out;
    connections:
        Button.signal --> Blinker.control;
}

```

Da es sich in Omnet++ um eine Netzwerksimulation handelt, wird einschließlich noch ein kleines Netzwerk aus den vorher definierten Modulen aufgebaut werden (Listing 3.4). Dabei werden zwei Module vom Typ Knoten erstellt und die jeweiligen Gates der Beiden miteinander verbunden.

Listing 3.4: einfaches Netzwerk

```

network Netzwerk
{
    submodules:
        node1: Knoten;
        node2: Knoten;
    connections:
        node1.in <-- node2.out;
}

```

---

```

    node1.out --> node2.in;
}

```

### 3.3.3 Einige Techniken, Funktionen und wichtige Module

Im folgenden Abschnitt wird ein Ausschnitt darüber gegeben, was Omnet++ an Funktionalitäten bereitstellt. Grundlegend wird für eine einfache Simulation ein Netzwerk benötigt, welches in der Netzwerkbeschreibungssprache NED beschrieben wird. Dieses Netzwerk kann dann Nodes definieren, welche selbst Module sind, welche wiederum auch in NED beschrieben werden. Wenn diese Module ausgehende und/oder eingehende Gates besitzen, können diese im Netzwerk wiederum miteinander verbunden werden.

Dieser einfache Grundaufbau genügt im Prinzip schon, damit eine valide Simulation ablaufen kann. Um dazu noch etwas Funktionalität in die Module zu bringen ist es möglich, für jedes Modul eine Klasse in C++ zu definieren. Man kann zum einen eigene Klassen definieren oder die in der Simulationsbibliothek vorhandenen Klassen nutzen.

Zusätzlich zur Standardbibliothek gibt es noch 3 relevante Frameworks, die weitere Funktionen zur Verfügung stellen: MiXiM, INET und Castalia.

#### 3.3.3.1 Nodes and Messages

Für die Steuerung innerhalb einer Simulation sind Nachrichten das wichtigste Werkzeug in **Omnet++**. So kann man eine Nachricht zu einer festgelegten Simulationszeit verschicken, um diese als Events einzusetzen. Dabei können Knoten auch Nachrichten an sich selbst versenden.

Um einem selbst definierten Modul die Möglichkeit zu geben Nachrichten zu verstehen und zu benutzen werden einige Funktionen bereitgestellt, die man selbst definieren muss.

Diese sind für die Funktionalität eines **cSimpleModule** entscheidend und sollten nach dem Erstellen eines neuen Moduls implementiert werden:

- void initialize()
- void handleMessage(cMessage \*msg)
- void activity()
- void finish()

**initialize()** Die Funktion **initialize()** wird nach dem Erstellen eines Moduls aufgerufen. Es kann so ähnlich wie ein Konstruktor verwendet werden. Entscheidend ist, dass die Methode erst aufgerufen wird, nachdem auch der **NED**-Teil des Moduls eingelesen wurde. Das bedeutet, dass erst an dieser Stelle auf Parameter des Moduls zugegriffen werden kann und das ist im Konstruktor noch nicht möglich. Auch Nachrichten kann das Modul erst ab diesem Zeitpunkt verschicken.

**handleMessage(cMessage \*msg)** Diese Methode kann eingehende Nachrichten auswerten. Sollten bei einem Modul Nachrichten ankommen, ohne dass diese Funktion definiert wurde, wird ein Fehler auftreten. Wie Nachrichten genauer aufgebaut sind, ist im Abschnitt **cMessage** beschrieben. Nachrichten können zeitgesteuert Events auslösen. Die Methode **handleMessage()** ist somit das Herzstück der meisten Module, da hier das komplette Verhalten geregelt wird.

Es können im Regelfall natürlich viele verschiedene Arten von Nachrichten in einem Modul eintreffen, die auch unterschiedlich behandelt werden müssen. Zur Fallunterscheidung stehen wiederum viele Funktionen im Nachrichtenmodul zur Verfügung, die beispielsweise Informationen über den Sender liefern, ob die Nachricht eine Selfmessage war, also vom Modul an sich selbst gesendet wurde oder einfache Informationen wie der Name der Nachricht.

**activity()** Diese Methode ist eine eher unwichtige Funktion. Wenn **handleMessage()** korrekt verwendet wird, sollte man auf die Benutzung von **activity()** am besten komplett verzichten. Es verhält sich oberflächlich betrachtet wie **handleMessage()**, allerdings wird diese Methode nicht einfach aufgerufen, sollte eine Nachricht ankommen, sondern läuft in einer Endlosschleife und wartet permanent aktiv auf Nachrichten. Daher ist sie wesentlich rechenintensiver als das Gegenstück **handleMessage()**.

**finish()** Diese Methode wird aufgerufen, nachdem die Simulation beendet wurde und noch bevor das Modul gelöscht wurde. Sie sollte nicht zum Löschen anderer Module verwendet werden, da nach dem Aufruf von **finish()** die Simulation erneut gestartet werden kann, ohne dass das Netzwerk komplett neu initialisiert wird. Wenn allerdings wichtige Module an dieser Stelle gelöscht wurden, ist ein Neustart nicht mehr möglich.

Die Methode ist stattdessen dafür da die eben abgelaufene Simulation auszuwerten. Es können an dieser Stelle alle relevanten Informationen gespeichert werden, damit diese hinterher statistisch ausgewertet werden können.

### 3.3.3.2 cMessage

Für die Nachrichten selbst existiert ein fertig implementiertes Modul namens **cMessage**. Dieses erfüllt schon die wichtigsten Anforderungen, die man an ein Nachrichtenmodul stellt. So können Nachrichten nicht nur **strings** übertragen, sondern alle Klassen, die von **cNamedObject** erben. Man kann also auch komplexe, selbst definierte Objekte mithilfe von cMessage übertragen.

Es empfiehlt sich dennoch eine eigene Kindklasse von cMessage zu definieren, da man in diesem eigene Parameter definieren kann, die verschiedene Werte beschreiben. So kann man zum Beispiel genauere Informationen für Quelle und Senke in der Nachricht speichern oder verschiedene Werte für Statistiken. Sollte man eine veränderte Kindklasse von cMessage definieren, so wird zusätzlich eine Klasse dazu generiert, die viele Funktionen bereitstellt, die zum Beispiel das Kopieren einer Nachricht ermöglichen - auch inklusive der extra hinzugefügten Parameter.

Tabelle 3.3: Übersicht über einige Funktionen von cMessage

typ	Funktion	Kurzbeschreibung
virtual cArray &	getParList ()	Parameterliste einer Nachricht
bool	isSelfMessage () const	ist Nachricht an sich selbst
cModule *	getSenderModule () const	
cGate *	getSenderGate () const	Informationen über Sender
int	getSenderModuleId () const	äquivalent für Arrival vorhanden
int	getSenderGateId () const	
simtime_t_cref	getCreationTime () const	Zeitpunkt: Nachricht erstellt
simtime_t_cref	getSendingTime () const	Zeitpunkt: Nachricht gesendet
simtime_t_cref	getArrivalTime () const	Zeitpunkt: Nachricht angekommen
bool	arrivedOn (int gateId) const	Id des Inputgate

### 3.3.3.3 XML Support

NED-Parameter eines Moduls können vom Typ `xml` sein. Die dazu gehörende Klasse **cXMLElement** bietet ihrerseits umfangreiche Unterstützung dafür an. Diese orientiert sich dabei an einem DOM-Parser, ist allerdings aus Performanzgründen nur ähnlich aufgebaut. Dabei stellt die Klasse die für X-Path typischen Funktionen für XML-Zugriffe bereit wie zum Beispiel **getParentNode()** oder **getChildren()**.

Listing 3.5: Beispiel einlesen von XML

```
cXMLElement *rootE = par("xmlFile").xmlValue();
cXMLElementList nListRows = rootE->getChildren();
int amountRows = nListRows.size();
int* data = new int[amountRows];
for (int i = 0; i < amountRows; i++){
    //nListRowArray ist eine Zeile aus dem XML file
    //bzw. alle Elemente 1. Ebene unter der Wurzel
    cXMLElement* nListRowArray = nListRows[i];
    //kann ab hier beliebig tief fortgesetzt werden
}
```

### 3.3.3.4 Statistiken

Um die durchgeführten Simulationen ordentlich auswerten zu können, stellt Omnet++ verschiedene Werkzeuge bereit. Neben dem Eventlog ist es möglich, während der Simulation Skalar- sowie Vektorwerte zu definieren. Die Skalarwerte besitzen dabei einen fixen Wert, wogegen Vektoren die Veränderung von bestimmten Variablen im Laufe der Simulationszeit speichern können. Die dazugehörigen Klassen heißen cLongHistogram und cOutVector.

Diese beiden Werte können nach dem erfolgreichen Abschließen einer Simulation auch grafisch ausgewertet werden. Dazu kann nach dem Exportieren der Werte in der Simulationsumgebung ausgewählt werden, welche Daten zusammen in einem Diagramm dargestellt werden sollen. Zum Erstellen der Graphen wird Gnuplot[6] genutzt. Anschließend können noch verschiedene Einstellungen getroffen werden, damit die Daten möglichst gut visualisiert werden können. Beispielsweise kann der Typ des Kurvenverlaufs gewählt werden. In den Bildern 3.4 und 3.3 ist ein Beispiel dafür. Es liegt ein Automat vor, welcher zwischen 3 Zuständen wechselt. Dabei zeigt 3.4 die wesentlich bessere Variante des Graphen.

### 3.3.3.5 weitere Beispiele

**cClassDescriptor** Eine Klasse, die dabei behilflich ist, die Felder eines Objektes zu finden. Dazu muss eine Klasse lediglich von cClassDescriptor erben. Die Verwendung funktioniert wie im Listing 3.6 beschrieben.

Abbildung 3.3: Zustandswechsel Automat falsch

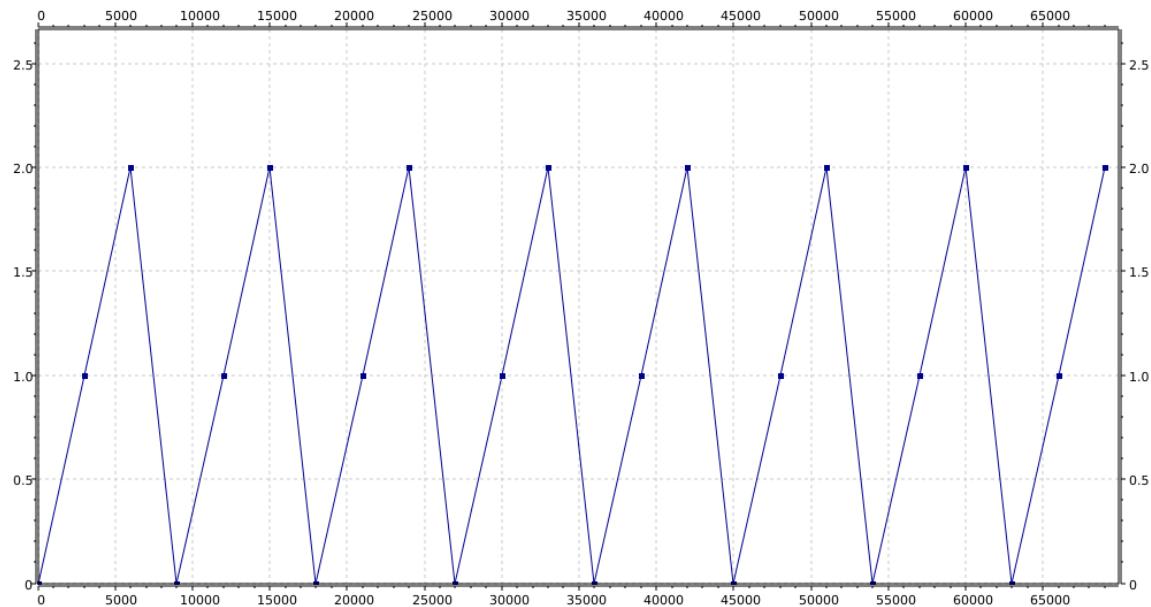
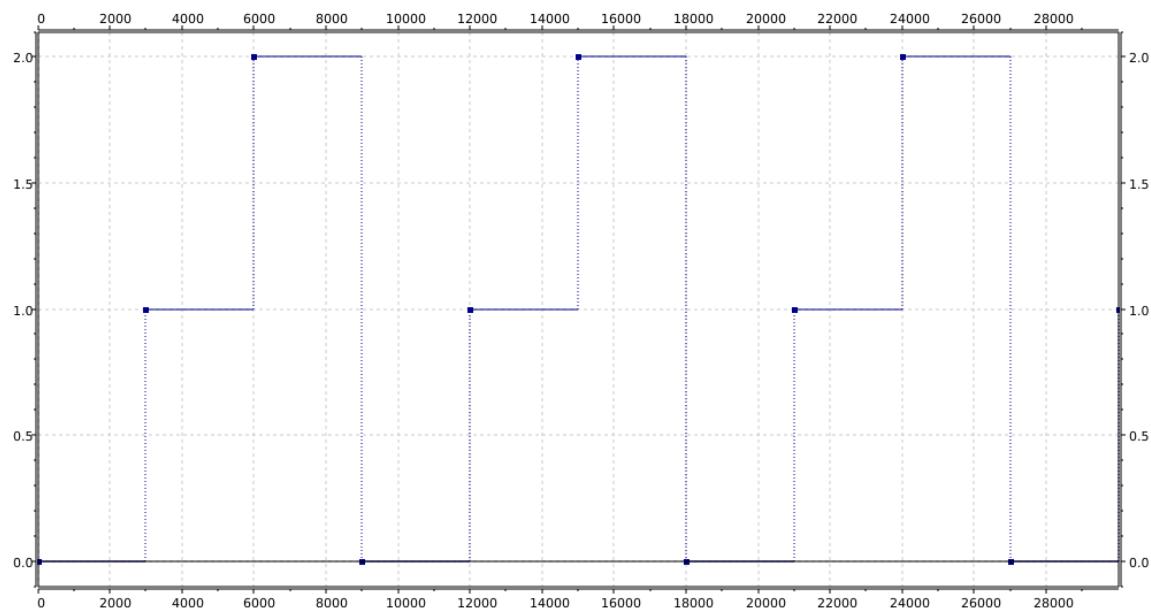


Abbildung 3.4: Zustandswechsel Automat richtig



Listing 3.6: Verwendung von cClassDescriptor

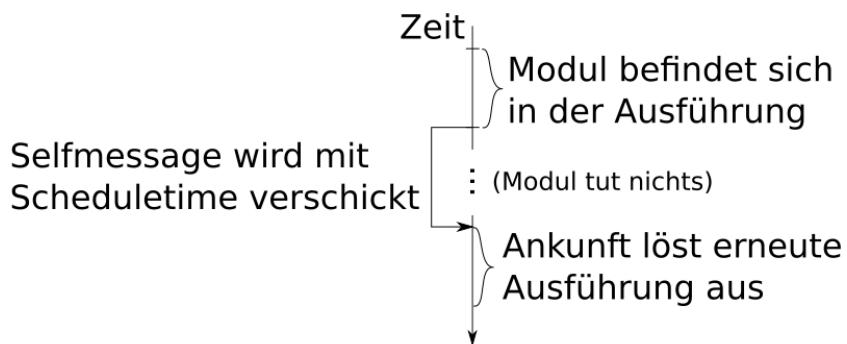
```
cClassDescriptor* thisDescr = cClassDescriptor::
    → getDescriptorFor(this);
int count = thisDescr->getFieldCount(this);

for ( int i = 0; i < count; i++) {
    std::stringstream s;
    s << i << " " << thisDescr->getFieldName(this, i) << " " ;
    s << thisDescr->getFieldAsString(this, i, 0);
    s << i << " " << thisDescr->getFieldName(this, i);
    cMessage *msg = new cMessage(s.str().c_str());
    send(msg, "gate$o");
}
```

**Simulationszeit und Events** Die Zeit einer Simulation verläuft linear. Anhand dieser kann die Ausführung von Events für einen bestimmten Zeitpunkt geplant werden. Dies ist möglich durch das Versenden von cMessage in Kombination mit einer Scheduletime und wenn nötig als Selfmessage.

Wenn ein Modul einen Zeitraum der Ausführung abgeschlossen hat, soll es unter Umständen eingefroren werden und erst zu einem späteren Zeitpunkt reaktiviert werden. Damit es in dieser Zeit nicht aktiv warten muss und somit unnötig Prozessorlast oder im Fall von mobilen Geräten auch noch zusätzlich Energie verbraucht, kann ein Modul jegliche Aktionen unterlassen, bis eine Nachricht, wie ein Event, die weitere Ausführung wieder anstößt (siehe Abbildung 3.5).

Abbildung 3.5: Beispiel cMessage als Event



Die Simulationszeit hat den Typ simtime\_t. Im Listing 3.7 sind Beispiele beschrieben, auf welche Weise die aktuelle Zeit abgefragt werden kann und wie ein Event zu einem bestimmten Zeitpunkt ausgelöst werden kann. Dabei kann einfach ein beliebiger Wert

als Verzögerung gewählt werden. Dieser wird zur momentanen Zeit addiert und dann wird der Event in Auftrag gegeben.

Listing 3.7: Simulationszeit und Event

```
//get the simtime
simtime_t time = simTime();
//define a delay
simtime_t delay = 10.0;
//schedule an event at a given time
scheduleAt(simTime()+delay, event);
```

**cArray** Ist eine Klasse, die als Sammlung für Objekte vom Typ cObject dient. Sie kommt zum Beispiel innerhalb der cMessage zum Einsatz. Die Parameterliste der Nachrichten besteht aus einem solchen cArray. Die wichtigsten Funktionen sind hier add() und remove(), die in Listing 3.8 kurz an einem Beispiel gezeigt werden.

Listing 3.8: cArray add und remove

```
//adding object
cMessage *newmsg = new cMessage("any name");
SimpleCoord *coord = new SimpleCoord("pos", position);
newmsg->getParList().add(coord);
send(newmsg, "toWorld$o");

//getting (and removing) objects
SimpleCoord *array = (SimpleCoord*) msg->getParList().remove(
"pos");
//double x = par->x;
//double y = par->y;
string name = msg->getName();
delete msg;
delete array;
```

Weiterhin gibt es noch Funktionen wie size(), find() oder clear(), die selbsterklärend sind. Neben remove() gibt es auch noch die Möglichkeit per get() auf Elemente zuzugreifen, ohne sie dabei sofort zu entfernen.

## 3.4 MiXiM-Framework als Omnet++-Erweiterung

### 3.4.1 Einleitung

MiXiM[8] ist ein Framework welches die Funktionalität von Omnet++ in erster Linie um mobile und kabellose Knoten erweitert. Es implementiert einige Protokolle und stellt verschiedene Knoten bereit.

Außerdem fügt es zusätzlich auch nützliche Hilfsfunktionen zu Omnet++ hinzu, wie beispielsweise die FindModule-Klasse.

### 3.4.2 Einige wichtige Module

#### 3.4.2.1 FindModule

Diese Klasse kann eine Instanz eines Objekts anhand eines Modulnamens finden. Es verhält sich daher wie eine Art Servicemanager. Man kann mithilfe dieser Submodule, globale Module, Hostmodule und Netzwerke finden. Dazu übergibt man einfach an ein Template wie in Beispiel 3.9 den gewünschten Modulnamen und kann dann eine der Methoden, wie zum Beispiel findSubModule() aufrufen.

Listing 3.9: Beispiel FindModule

```
//template der FindModule Klasse
template<typename T = cModule * const >
//Beispielverwendung
FindModule<BasePhyLayer*>::findSubModule(this)
```

#### 3.4.2.2 Coord

Coord ist eine einfache Klasse zur Repräsentation von Koordinaten im 3-dimensionalen Raum. Es ist eine elementare Klasse für das MiXiM-Framework, da dieses Mobilität implementiert. Coord beinhaltet zusätzlich zu den 3-D-Koordinaten auch beispielsweise untere und obere Grenzen, Operatoren zum Rechnen und vergleichen und weitere Methoden wie zum Beispiel eine Distanzberechnung zwischen Koordinaten.

#### 3.4.2.3 Mobility

Wie im vorherigen Abschnitt schon erwähnt ist die Mobilität eine der wichtigsten Funktionen, die das MiXiM-Framework bereitstellt. Es werden 4 verschiedene Arten

von Bewegungen zur Verfügung gestellt, wobei eine davon, die LineSegmentMobilityBase, selbst wiederum viele verschiedene Varianten zur Verfügung stellt:

- CircleMobility
- LinearMobility
- RectangleMobility
- LineSegmentMobilityBase

Während die ersten 3 Arten sich von selbst erklären, ist dass bei der LineSegmentMobilityBase nicht der Fall. Es ist eine weitere Basisklasse für Bewegungsarten, welche aus einer Sequenz verschiedener linearer Bewegungen bestehen. Ein Beispiel ist die sogenannte TurtleMobility. Bei dieser kann ein Skript als XML-File hinterlegt werden, welches die Sequenz beschreibt.



## 4 Implementierung

### 4.1 Einleitung

Ziel dieser Arbeit war es, eine Simulationsumgebung für Sensorknoten zu schaffen. Es sollte viele verschiedene Arten von Sensorknoten geben, die jeweils einen oder mehrere verschiedene Sensoren besitzen. Mit diesen Knoten sollte ein Netzwerk aufgebaut werden, um die Umgebungsparameter eines Gebietes zu erfassen. Die Daten der Simulation sollten visualisiert und ausgewertet werden können, besonders hinsichtlich Energieverbrauch und den dazugehörigen Batteriezuständen der jeweiligen Elemente im Verlauf der Zeit.

### 4.2 Aufbau und Struktur

#### 4.2.1 Klassenübersicht

Die Klassenübersichten wurden teilweise mit Hilfe von doxygen[1] erstellt. Die Tabelle 5.1 zeigt einen Überblick über die definierten Module.

##### 4.2.1.1 CustomWorldUtility

Die Klasse CustomWorldUtility ist eine sehr wichtige für die Simulation. Sie repräsentiert die Umgebung, also den Bereich, indem sich die Knoten befinden. Sie erbt von der Klasse BaseWorldUtility aus dem MiXiM-Framework. BaseWorldUtility stellt die nötigen Funktionalitäten für den sogenannten Playground bereit.

Zusätzlich dazu stellt die Klasse selbst die notwendigen Parameter für die Umwelt bereit. Nach dem Starten der Simulation steht darin jeweils ein k-dimensionales Array pro Sensortyp bereit: temperatureArray, pressureArray, humidityArray und lightArray. Dabei ist k die definierte Anzahl der Dimensionen, je nachdem ob eine 2-dimensionale Fläche oder ein 3-dimensionaler Raum simuliert werden soll. Die Arrays enthalten die Parameter der Umgebung; temperatureArray beinhaltet zum Beispiel, wie der Name schon sagt, Informationen über die Temperatur.

Es kann zu Beginn der Simulation entschieden werden, ob neue Werte berechnet

Tabelle 4.1: Klassenübersicht

Klasse	Beschreibung
BatteryAccess	gibt einer Klasse, welche von dieser erbt Zugriff auf die Batterie
Memory	eine einfache Implementierung eines key-value-Speichers mit CRUD-Operationen
Processor	Repräsentiert ein paar Grundfunktionen eines Prozessor übernimmt die Steuerung der Sensoreinheit, hat Zugriff auf den Speicher und kann außerdem zwischen verschiedenen power-Modi wechseln
AbstractSensingUnit	einfache Implementierung einer SensingUnit diese kann Werte der Umgebung messen und verbraucht dabei Energie
SensorNode	ist hauptsächlich für die Initialisierung zuständig
AbstractSignalConditioner	modelliert den Energieverbrauch vom SignalConditioner
AbstractSignalConverter	modelliert den Energieverbrauch vom SignalConverter
AbstractTransducer	modelliert den Energieverbrauch vom Transducer
CustomWorldUtility	stellt die Umgebung dar: generiert Umweltparameter und speichert diese liefert auf Anfrage von Sensoren Messwerte
ExtendedMessage	Nachricht mit einigen Parameters für Statistiken
SimpleSensorData	eine Klasse die von cNamedObject erbt kann eingesetzt werden um an die Parameterliste von Nachrichten angehängt zu werden mit dieser können Integerwerte versendet werden
StatisticsInterface	Interface welches Statistiken zur Verfügung stellt

werden sollen oder die bereits vorhandenen Werte für die Umgebung übernommen werden sollen. Die Arrays besitzen die gleiche Größe wie der Playground. Diese Größe ist auch zusätzlich in den Parametern sizeX, sizeY und sizeZ gespeichert.

Da jedoch im 3-dimensionalen Fall die Datenmenge sehr schnell steigt, ist auch möglich über den Parameter dataGranularity im zugehörigen ned-Modul zu definieren, wie detailliert die Daten erstellt werden sollen. Sollte man zum Beispiel den Wert 10 setzen, so wird nur alle 10 Meter ein Wert generiert.

Wenn neue Werte generiert werden sollen, so wird für jeden Messtyp eine xml-Datei

in der entsprechenden Größe angelegt und mit Messdaten gefüllt. Diese werden anschließend ausgelesen und in Form der oben genannten Arrays gespeichert. Wenn keine neuen Daten erstellt werden sollen, so wird ein bereits existierendes xml-File genutzt.

Zum Erstellen neuer Daten kann die Funktion **generateEnvironmentData()** genutzt werden. Es ist dadurch auch möglich während der Simulation neue Werte zu generieren, indem man diese Funktion aufruft. Die Funktion legt pro Umweltparameter eine xml-Datei im Ordner *WorldModel/data* an. Jede der xml-Dateien wird beim Start mithilfe der Funktion **readXML(int)** eingelesen, verarbeitet, das heißt in ein Array umgewandelt und anschließend in der Klasse gespeichert.

Sollte nun ein Sensor einen Messwert auslesen wollen, so kann dieser auf die Funktion **getValueByPosition((std::string type, Coord \*position))** zugreifen und anhand seines Sensortyps und seiner Position einen entsprechenden Wert geliefert bekommen.

**Einblick in einige Funktionen** Wie im vorigen Abschnitt beschrieben, übernimmt die Funktion **generateEnvironmentData()** das Erstellen von Umweltparametern. Dafür ruft sie, je nach Sensordatentyp, eine der folgenden Funktionen auf:

- **generateTemperature()** - °C
- **generateHumidity()** - %
- **generatePressure()** - hPa
- **generateLight()** - lx

Diese Funktionen generieren je ein 2- oder 3-dimensionales Array in der Größe des Playgrounds mit Werten, die die jeweils oben angegebenen Einheiten besitzen. In Listing 5.1 ist die **generateTemperature()**-Funktion als ein Beispiel aufgeführt. Im Falle dieser Funktion werden Zufällige integer-Werte generiert, welche sich im Bereich 10 bis 30 bewegen. Dieser erste Entwurf ermöglicht es erst einmal, dass einfach und halbwegs realitätsnahe Werte für den jeweiligen Umweltparameter vorliegen. Später kann an dieser Stelle dann auch ein Zugriff auf eine externe Datenbank erfolgen, in der Messwerte gespeichert sind, die in einem realen Gebiet gemessen wurden.

Listing 4.1: **generateTemperature()**

```
int* CustomWorldUtility::generateTemperature(int size)
{
    int* data = new int[size];
    for (int i = 0; i < size; i++) {
        // 10 - 30
        data[i] = (int)((rand() % 100)/5) + 10;
    }
    return data;
}
```

Listing 4.2: Kurzbeschreibung readXML()

```
cXMLElementList nListRows = rootE->getChildren();
int count = nListRows.size();
for (int i = 0; i < count; i++) {
    val[i] = atoi(nListRows[i]->getNodeValue());
}
```

Um eben diese Daten wieder lesen zu können, dient die Funktion **readXML()**. Sie verwendet die cXMLElement-Funktionen aus dem Omnet++-Framework. Diese sind sehr hilfreich beim Verarbeiten von XML-Dateien. Ein Beispiel für die Verwendung bei einem XML-File mit einer Verschachtelungstiefe von 1 ist im Listing 5.2 zu sehen. Da in dieser Simulation die Umweltparameter in Form von XML gespeichert sind, ist dies eine sehr nützliche Funktionalität.

Die bis jetzt beschriebenen Funktionen dienen alle zur Initialisierung der Simulationsdaten. Es ist natürlich auch möglich diese in einem späteren Verlauf der Simulation erneut aufzurufen und damit neue Werte zu generieren.

Für den späteren Verlauf der Simulation ist nun noch die Funktion **getValueByPosition(std::string, Coord\*)** (5.3) von großer Bedeutung. Diese Funktion ermöglicht den Sensoreinheiten, auf die gespeicherten Daten zuzugreifen.

Listing 4.3: getValueByPosition()

```
int CustomWorldUtility::getValueByPosition(std::string type,
                                         Coord *position)
{
    int*** data = readXML(getEnumFromType(type));
    int dataAtPosition =
        data
            [((int)(position->x/par("dataGranularity")).
             → longValue())]
            [((int)(position->y/par("dataGranularity")).
             → longValue())]
            [((int)(position->z/par("dataGranularity")).
             → longValue())];
    return dataAtPosition;
}
```

#### 4.2.1.2 SensorNode

Die Klasse AbstractSensorNode selbst enthält eher weniger Funktionalität. Sie dient hauptsächlich zum Initialisieren des gesamten Knotens. Das funktioniert in Omnet++ natürlich zum größten Teil automatisch, allerdings liegt für die Sensoren eine

parameterabhängige, generische Initialisierung vor. So können innerhalb des NED-Moduls die folgenden boolschen Parameters gesetzt werden:

- hasTemperatureSensor
- hasHumiditySensor
- hasPressureSensor
- hasLightSensor

Je nachdem ob der jeweilige Parameter gesetzt wird oder nicht, wird das entsprechende Sensormodul auf dem Sensorknoten erstellt. Die Gates und Connections eines Moduls lassen sich in der NED-Definition leider nicht so schön generisch erstellen. Daher wird die Initialisierung des Prozessors zusammen mit seinen Gates, Connections und den entsprechenden Channels in der C++-Klasse übernommen. Nach dem Erstellen des Prozessors selbst werden für jeden existierenden Sensor ein eingehendes und ein ausgehendes Gate erstellt. Zusätzlich wird noch ein beidseitiges Gate für die Verbindung zum Memory-Modul angelegt.

Die Prozessorgates erhalten dann die Verbindungen zu den jeweils entsprechenden Gates, zum Sensor verläuft ein Gate zur Sensing Unit, das vom Sensor eingehende kommt vom Transducer. Beide Verbindungen haben unterschiedliche Funktionen, einmal lediglich zum Senden eines Signals um innerhalb der Sensing Unit einen Messvorgang gestartet und ein anderes Mal um die gemessenen Werte an den Prozessor zu übermitteln. Es wird daher in 2 verschiedene Channels zum Senden unterschieden, welche entsprechend eine höhere oder niedrigere Datenrate aufweisen können, je nachdem wie diese innerhalb des ini-Files definiert sind.

Wie genau der Prozessor Schritt für Schritt angelegt wird, ist im Listing 5.4 zu sehen. Für das Beispiel wurde jedoch nur ein Paar von Gates verbunden, da die Verbindung für alle anderen Gates analog aussieht.

Listing 4.4: teilweise Initialisierung des Prozessors

```
//get processor module type
cModuleType *moduleType = cModuleType::get("sensortechology.
    ↪ src.SensorNode.Processor.Processor");
//create module
cModule *processor = moduleType->create("Processor", this);
processor->finalizeParameters();
//create gate
cGate *fromSensor = processor.addGate(("from" + SensorType + "
    ↪ Sensor").c_str(), cGate::INPUT);
//get sensor
cModule *Sensor = this->getSubmodule((SensorType + "Sensor").
    ↪ c_str());
```

```
//get sensor gate
cGate *sensorOut = Sensor->gate("fromTransducerToNodeProcessor
    ↵ ");
//connect the gates
sensorOut->connectTo(fromSensor);
//finish the process
processor->buildInside();
```

#### 4.2.1.3 Sensormodule

Der Sensor in seiner Gesamtheit besitzt keine eigene C++-Klasse. Jedoch besitzt er 4 Module, die gemeinsam die Funktionalität des Sensors ausmachen, welche jeweils eine Klasse haben. Diese werden im folgenden Abschnitt beschrieben.

**SensingUnit** Die SensingUnit ist der wichtigste Teil des Sensors. Innerhalb der Definition von **handleMessage(cMessage\*)** wartet das Modul auf ein Signal vom Prozessor. Sobald dieses kommt, wird ein Messvorgang gestartet. Zunächst überprüft das Modul dabei die eigene Position. In Abhängigkeit von dieser wird anschließend ein dem Typ entsprechender Messwert aus dem Modul **CustomWorldUtility** ausgelesen, welcher hinterher an den SignalConditioner weitergegeben wird. Dazu wird eine Instanz der Klasse SimpleSensorData genutzt, welche an eine cMessage angehängt wird und dann über das entsprechende Gate gesendet wird.

**SignalConditioner** Der SignalConditioner wartet auf eingehende Daten von der SensingUnit. In der Praxis würde das Signal hier aufbereitet werden, jedoch ist die genaue Modellierung an dieser Stelle nicht relevant. Daher kann lediglich ein Energieverbrauch für die durchgeföhrten Operationen definiert werden, welcher dann bei jedem Aufruf verbraucht wird. Anschließend werden eingegangene Nachrichten weitergeleitet an den SignalConverter.

**SignalConverter** Für den SignalConverter gilt dasselbe wie für den SignalConditioner. Die eingegangenen Nachrichten werden lediglich an den Transducer weitergeleitet, ohne das genaue Verhalten zu simulieren und auch dabei steht wieder nur die Modellierung des Energiehaushalts im Vordergrund.

**Transducer** Das letzte Bauteil der Messkette ist der Transducer. Dieser erhält seine Daten vom SignalConverter. Auch hier steht der Energieverbrauch im Vordergrund.

Wie in der Realität auch, hat der Transducer am Ende den verwertbaren Messwert in einer festgelegten Einheit vorliegen. Dieser wird anschließend an den Prozessor gesendet, welcher sich dann um die Verwendung dessen kümmert.

#### 4.2.1.4 Prozessor

Eine wichtige Funktion des Prozessors wurde eben schon indirekt beschrieben. Die Klasse **Processor** kümmert sich um die Steuerung des Sensors. Aber er erfüllt auch andere Funktionen, wie die Energieverwaltung des Sensorknotens und die Verwertung von statistischen Daten.

**Sensorsteuerung** Der jeweilige Prozessor kann über das entsprechende, ausgehende Gate zu seinen Sensoren einen Messvorgang auslösen, indem die Funktion startSensingUnit() (Listing 5.5) aufgerufen wird. Anschließend bekommt er die Daten vom Transducer übermittelt. Diese Information kann nun unterschiedlich behandelt werden. Typischerweise kann dieser Wert nun zunächst im Speicher hinterlegt werden. Dafür wird die eingegangene Message direkt an den Memory weitergeleitet. Wie das genau aussieht, ist im folgenden Abschnitt des Memories beschrieben.

Es ist auch möglich, periodisch Messungen durchzuführen. Dazu muss lediglich im NED-Modul ein Wert für sensingIntervall gesetzt werden. Dieser legt fest, welche zeitlichen Abstände zwischen 2 Messungen vorliegen sollen. Es wird dann bei der Initialisierung ein Event gestartet, welcher jeweils nach dem festgelegten Zeitintervall ausgelöst wird. In diesem Fall wird dann ein Messvorgang gestartet. Danach wird der nächste folgende Event initialisiert und zum gegebenen Zeitpunkt beginnt der Vorgang von Neuem.

Listing 4.5: startSensingUnit()

```
void Processor::startSensingUnit()
{
    cModule* SensorNode = getParentModule();
    if (SensorNode->par("hasTemperatureSensor")) {
        cMessage* msg = new cMessage("startMeasuring");
        send(msg, "toTemperatureSensor");
    }
    if (SensorNode->par("hasHumiditySensor")) {
        cMessage* msg = new cMessage("startMeasuring");
        send(msg, "toHumiditySensor");
    }
    if (SensorNode->par("hasPressureSensor")) {
        cMessage* msg = new cMessage("startMeasuring");
        send(msg, "toPressureSensor");
    }
}
```

```

    }
    if (SensorNode->par("hasLightSensor")) {
        cMessage* msg = new cMessage("startMeasuring");
        send(msg, "toLightSensor");
    }
}

```

**ProcessorMode** Weiterhin hat der Prozessor die Möglichkeit zwischen verschiedenen Powermodi zu wechseln. Die existierenden Modi sind in einem enum (Listing 5.6) definiert und können per Funktion switchProcessorMode gewechselt werden (Listing 5.7).

Listing 4.6: enum MODES

```

enum MODES {
    NORMAL = 0,
    POWER_SAVING = 1,
    HIGH_PERFORMANCE = 2,
    OFF
}

```

Listing 4.7: switchProcessorMode()

```

void Processor::switchProcessorMode(MODES mode)
{
    //switch batteries power accounts
    if (mode == POWER_SAVING) {
        drawCurrent(currentOverTimePowerSaving, 1);
    } else if (mode == NORMAL) {
        drawCurrent(currentOverTimeNormal, 0);
    } else if (mode == HIGH_PERFORMANCE) {
        drawCurrent(currentOverTimeHighPerformance, 2);
    }
}

```

So können beispielsweise Wach-und-Schlaf-Zyklen der Sensorknoten gesteuert werden. Da die Energiequelle von Sensorknoten oft sehr klein ist, ist es sinnvoll die Knoten in Zeiten, in denen keine Messwerte aufgenommen werden sollen in einen Stand-by Zustand zu versetzen. Dadurch kann viel Energie gespart werden.

Diese Funktion wird durch die verschiedenen Powermodi bereitgestellt. Es ist möglich einen Zeitintervall zu definieren, in dem die verschiedenen Knoten gemeinsam zwischen den verschiedenen Modi wechseln. Da ein Knoten in Zeiten des Stand-by keine Kommunikation durchführen kann, ist am sinnvollsten alle Knoten im Netzwerk synchron schlafen zu lassen. Der entsprechende Zeitintervall kann durch die Variable

shiftProcessorModeIntervall definiert werden.

Dadurch werden nun wieder zeitlich periodische Wechsel zwischen den Modi durchgeführt, welche durch Events gesteuert werden, indem die Funktion switchProcessorMode() (5.7) aufgerufen wird. Diese Funktion ist überladen und kann auf 3 verschiedene Arten aufgerufen werden: mit dem Modus in die gewechselt werden soll, mit dem int-Wert der diesen Modus repräsentiert oder ohne Parameter, in dem Fall wird der nächste Modus anhand der Integerwerte bestimmt, wie bei einem Modulozähler. Für jeden einzelnen Zustand kann dann wiederum im ini-File definiert werden, wie der Energieverbrauch pro Zeiteinheit während welches Modus genau aussehen muss.

**Statistiken** Die dritte Funktion des Prozessors, welche ebenfalls über Events gesteuert wird, ist das Erfassen statistischer Daten. Auch hierfür kann, in Form von der Variable collectStatisticsIntervall, ein zeitlicher Intervall definiert werden, nach diesem jeweils ein Event eine Erhebung von statistischen Messwerten des Batteriezustands durchführt. Die dazu gehörige Funktion ist in Listing 5.8 zu sehen.

Listing 4.8: doCollectStatistics()

```
void Processor::doCollectStatistics()
{
    voltage = battery->getVoltage();
    voltageStats.collect(voltage);
    voltageVector.record(voltage);

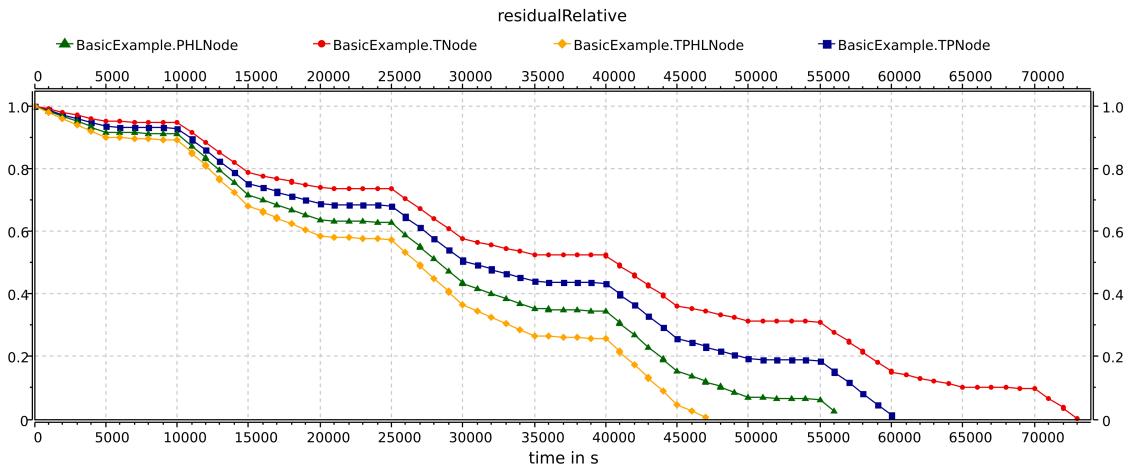
    residualRelative = battery->estimateResidualRelative();
    residualRelativeStats.collect(residualRelative);
    residualRelativeVector.record(residualRelative);

    residualAbs = battery->estimateResidualAbs();
    residualAbsStats.collect(residualAbs);
    residualAbsVector.record(residualAbs);
}
```

Mit Abbildung 5.1 ist ein Beispiel gegeben, bei dem alle 3 Events in Verwendung waren. Dabei ist natürlich von der Sensorsteuerung nicht viel zu sehen. Alle 5000 Sekunden wurde hierbei der Modus gewechselt, beginnend bei dem normalen Modus, danach einem energiesparenden Modus und zuletzt einem Modus mit sehr hohem Energieverbrauch. Danach wieder beginnend von vorn. Dabei wurde alle 1000 Sekunden der Ladezustand abgefragt. Diese Operationen wurden auf 4 verschiedenen Sensorknoten parallel durchgeführt. Man kann sehr gut sehen, dass der gelbe Verlauf besonders schnell nach unten geht. Das liegt daran, dass dort ein Knoten mit 4 Sensoren genutzt wurde. Der Rote dagegen konnte deutlich länger am Leben bleiben,

weil dieser Knoten lediglich einen Temperatursensor besaß. Der Energieverbrauch für die jeweils analogen Bauteile war bei allen 3 Sensorknoten gleich.

Abbildung 4.1: Beispielstatistik Relativer Batterieladezustand



#### 4.2.1.5 Memory

Der Speicher besitzt Arrays für einen key-value-Speicher und implementiert zur Benutzung dieses Speichers die CRUD-Operationen, wie in Listing 4.9 zu sehen ist.

Listing 4.9: CRUD-Operationen vom Memory

```
#define error -9999
const static int storageSize = 4;

void Memory::createEntry(std::string type, int value)
{
    int emptyId = -1;
    for (int i = 0; i < storageSize; i++) {
        if (storageType[i] == "") {
            emptyId = i;
            break;
        }
    }
    if (emptyId == -1) {
        return;
    }
    storageType[emptyId] = type;
    storageValue[emptyId] = value;
}
```

```
int Memory::readEntry(std::string type)
{
    int id = getIdByType(type);
    if (id == -1) {
        return error;
    }
    return storageValue[id];
}

void Memory::updateEntry(std::string type, int value)
{
    int id = getIdByType(type);
    if (id == -1) {
        return createEntry(type, value);
    }
    storageValue[id] = value;
}

void Memory::deleteEntry(std::string type)
{
    int id = getIdByType(type);
    if (id == -1) {
        return;
    }
    storageValue[id] = error;
    storageType[id] = "";
}
```

Wenn nun von außen eine Message mit einem Datensatz an den Speicher geschickt wird, so kann dieser je nach Bedarf Datensätze neu anlegen, löschen oder updaten, damit diese später, wenn die Daten benötigt werden, wieder abgerufen werden können.

#### 4.2.1.6 BatteryAccess

Die Klasse BatteryAccess ermöglicht den Modulen des Sensors auf die Batterie zugreifen zu können. Dafür ist zunächst eigentlich die Klasse MiximBatteryAccess zuständig. BatteryAccess erbt von dieser und erweitert sie um einige wichtige Funktionen. So können zum Beispiel direkt Werte für den Energieverbrauch des entsprechenden Moduls hinterlegt werden. Immer wenn nun eine energieintensive Operation durchgeführt wird, verbraucht die Batterie Energie in Höhe dieses definierten Wertes, indem

die Funktion draw() aufgerufen wird. Diese ruft anschließend die Funktion drawEnergie(float) mit dem gespeicherten Wert **float energiePerOperation** auf.

Die Klasse kümmert sich weiterhin um die Initialisierung und um den Fall, dass die Batterie den leeren Zustand erreicht (handleHostState()).

Listing 4.10: BatteryAccess

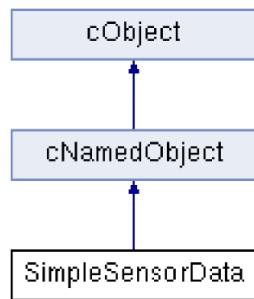
```
class BatteryAccess : public MiximBatteryAccess {
protected:
    float currentOverTime;
    float energiePerOperation;
    //int deviceID; - defined inside MiximBatteryAccess
public:
    BatteryAccess();
    virtual ~BatteryAccess();
    void initialize(int stage);
    void draw();
    void finish();
    virtual void handleHostState(const HostState &state);
};
```

#### 4.2.1.7 SimpleSensorData

Die Klasse SimpleSensorData dient dazu, die Übertragung von Sensordaten zu ermöglichen. In Omnet++ ist es möglich, eine Parameterliste an Nachrichten anzuhängen. Die möglichen Parameter sind dabei eingeschränkt. Um nach dem Einfügen auch wieder auf die Parameter zugreifen zu können, wird cNamedObject als Parameter benötigt. Das ermöglicht beim späteren Auslesen des Parameters den Zugriff per string.

Es ist also nicht möglich, die C++-build-in-types direkt anzuhängen. Darum wird die Klasse SimpleSensorData genutzt, um die Integerwerte von Messungen per Nachricht zu übertragen.

Abbildung 4.2: SimpleClasses: Member



Listing 4.11: Beispiel für Senden und Empfangen mit Parameterliste

```
//im Sender:
cMessage *msg = new cMessage(msgname);
SimpleSensorData *data = new SimpleSensorData("Temperature",
    ↳ temp);
msg->getParList().add(coord);

//
// Nachricht uebertragen
//

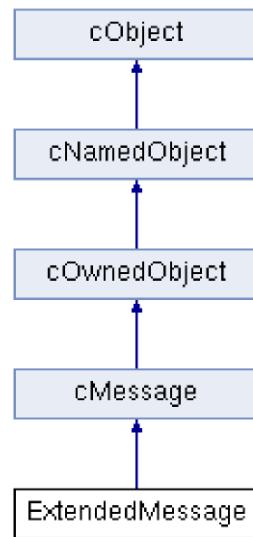
//im Empfaenger:
SimpleSensorData *data = (SimpleSensorData*) msg->getParList()
    ↳ .remove("Temperature");
```

#### 4.2.1.8 ExtendedMessage

ExtendedMessage erbt direkt von der Klasse cMessage, der Standard-Nachrichtenklasse in Omnet++. Im Grunde stellt cMessage alle benötigten Funktionen bereit. ExtendedMessage ist nur aus dem Grund vorhanden, um zusätzliche Statistiken über Nachrichten erstellen zu können, zum Beispiel wie oft eine einzelne Nachricht weitergesendet wurde.

Im Listing 5.12 ist zu sehen welche Statistikparameter erhoben werden.

Abbildung 4.3: ExtendedMessage: Vererbung



Listing 4.12: ExtendedMessage

```

message ExtendedMessage extends cMessage
{
    int source;
    int destination;
    int hopCount = 0;
}
  
```

#### 4.2.1.9 StatisticsInterface

Dieses Interface enthält grundlegende Attribute für Statistiken. Klassen, die dieses implementieren, speichern somit zum Beispiel, wie viele Nachrichten sie empfangen oder gesendet haben.

#### 4.2.2 Übersicht NED-Module

Im folgenden Abschnitt werden alle Simulationsobjekte erläutert, also all jene, die durch die Sprache NED beschrieben wurden:

- Simple Module
  - AbstractSensingUnit
  - AbstractSignalConditioner
  - AbstractSignalConverter
  - AbstractTransducer
  - CustomWorldUtility
  - ExampleProcessor
  - Memory
  - Processor
  - SensingUnitHumidity
  - SensingUnitLight
  - SensingUnitPressure
  - SensingUnitTemperature
  - SignalConditionerHumidity
  - SignalConditionerLight
  - SignalConditionerPressure
  - SignalConditionerTemperature
  - SignalConverterHumidity
  - SignalConverterLight
  - SignalConverterPressure
  - SignalConverterTemperature
  - TransducerHumidity
  - TransducerLight
  - TransducerPressure
  - TransducerTemperature
- Compound Module
  - AbstractSensor
  - SensorNode
  - HumiditySensor
  - LightSensor
  - PressureSensor
  - TemperatureSensor
- Messages und Channel
  - ExtendedMessage
  - DatarateChannel
- Network
  - BasicWSN

### 4.2.2.1 Simple Module

Simple Module sind Komponenten in einer Omnet++ Simulation, die die größte Auswirkung auf die Wirkungsweise des Netzwerkes haben. Das liegt daran, dass bei ihnen neben einer Beschreibung in NED auch eine Beschreibung in C++ vorliegt. Daher kann das Verhalten jener Module während der Simulation ausführlich definiert werden. Simple Module werden oft in Gruppen zusammengefasst, also in Compound

Modulen. Dadurch kann man komplexe Modellbeschreibungen erzeugen. Im folgenden Teil werden einige dieser Simple Module beschrieben.

#### 4.2.2.2 CustomWorldUtility

Wie im Codebeispiel (5.13) zu sehen ist, ist das Modul CustomWorldUtility eine Erweiterung des Moduls BaseWorldUtility. Zusätzlich zu den darin definierten Eigenschaften hat es einige weitere Parameter. Zum einen den integer-Wert dataGranularity, welcher festlegt, wie genau die Messwerte generiert werden sollen. Er legt fest, wie groß die Fläche, beziehungsweise der Würfel sein soll, für die jeweils ein Messwert angelegt wird. Dabei definiert dataGranularity die Kantenlänge. Die restlichen Variablen speichern die Speicherorte für die Messwerte, welche als xml-Datei erstellt werden.

Listing 4.13: CustomWorldUtility

```
package mynetwork.WorldModel;
import org.mixim.base.modules.BaseWorldUtility;

simple CustomWorldUtility extends BaseWorldUtility
{
    bool createData;
    int dataGranularity;
    string basePath = "src/WorldModel/data/";
    xml xmlDoc xmlTemperature = xmlDoc("src/WorldModel/data/
        ↪ temperature.xml");
    xml xmlDoc xmlPressure = xmlDoc("src/WorldModel/data/pressure
        ↪ .xml");
    xml xmlDoc xmlHumidity = xmlDoc("src/WorldModel/data/humidity
        ↪ .xml");
    xml xmlDoc xmlLight = xmlDoc("src/WorldModel/data/light.xml")
        ↪ ;
    @class ("CustomWorldUtility");
}
```

#### 4.2.2.3 Sensormodule

Es gibt 4 verschiedene Module aus denen sich ein Sensor, unabhängig von seinem Typ, zusammensetzt:

- SensingUnit

- SignalConditioner
- SignalConverter
- Transducer

Für jeden dieser Typen gibt es ein abstraktes Modell, von welchem alle Sensortypen mit einer eigenen Implementierung erben. Weiterhin steht ein Interface zur Verfügung, welche diese implementieren müssen. Unterschiedliche Parameter besitzen die einzelnen Implementierungen nicht, jedoch bieten die verschiedenen Benennungen die Möglichkeit, den Energieverbrauch für jeden unterschiedlichen Sensortyp genau steuern zu können.

Alle dieser Module haben jeweils Parameter für den Energieverbrauch, welcher in der dahinter liegenden Klasse ausgewertet wird und ein input- und ein output-Gate. Die Verbindungen, zu denen die Gates führen, unterscheiden sich jedoch.

Während für die in der Messkette mittleren Module jeweils ein Gate zum vorherigen und eines zum nachfolgenden Modul führt, gilt das nicht für die SensingUnit und den Transducer. Die SensingUnit hat ein eingehendes Gate vom Prozessor, welcher ein Signal senden kann, um eine Messung in Gang zu setzen und der Transducer besitzt ein ausgehendes Gate zum Prozessor, um die fertig eingelesenen Daten an diesen zu übermitteln.

### 4.2.2.4 Prozessor

Das NED-Modul des Prozessors bietet nicht besonders viele Einstellungsmöglichkeiten. Es existiert ein Parameter numGates, welcher definiert, wie viele Gates der Prozessor besitzt, wobei der Wert dafür innerhalb der Initialisierung der C++-Klasse gesetzt wird. Zusätzlich kann die Anzahl der Prozessor Modi geändert werden, wobei dann wiederum zusätzliche Anpassungen innerhalb der C++-Klasse notwendig sind, damit diese auch genutzt werden können. Weiterhin können für die bestehenden 3 Modi die Parameter für den Energieverbrauch jeweils definiert werden.

### 4.2.2.5 Memory

Das NED-Modul des Speichers bietet nicht viele Parameter. Lediglich der Energieverbrauch des Bauteils kann an dieser Stelle definiert werden.

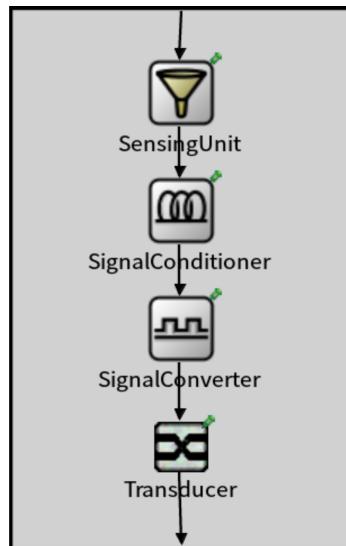
#### 4.2.2.6 Compound Module

Compound Module dienen dazu, andere Module zusammenzufassen, sollen jedoch keine eigene aktive Funktionalität definieren. Ihr Verhalten soll sich allein durch die Submodule ergeben. Es ist daher nicht sinnvoll eine C++-Klasse für diese Module zu definieren.

Für den Sensorknoten wurde dennoch eine eigene Klasse definiert, welche allerdings keine Funktion während der Simulation übernimmt, sondern allein für die generische Initialisierung zuständig ist.

#### 4.2.2.7 Sensoren

Abbildung 4.4: allgemeines Modell eines Sensors



Es liegt ein abstraktes Modul und ein Modulinterface für die verschiedenen Sensoren vor und für jeden Sensorotyp existiert ein Sensormodul, welches vom abstrakten Modul erbt und das Sensorinterface implementiert.

Die abstrakte Klasse besitzt wenige Parameter; zum einen type, in dem in der jeweiligen Sensorimplementierung der Typ des Sensors definiert werden muss und die beiden Parameter dataBandwidth und controlBandwidth, welche die Bandbreite für die 2 verschiedenen Arten von Channels definieren.

Das wichtigste ist natürlich die Definition der Submodule (5.14), denn diese machen die Funktionalität eines Compound Moduls aus. Die Abbildung 5.4 zeigt diesen Aufbau in bildlicher Form.

Listing 4.14: AbstractSensor

```
submodules:
    SensingUnit: <"SensingUnit"+type> like
        ↪ SensingUnitInterface {
            @display("p=100,50");
    }
    SignalConditioner: <"SignalConditioner"+type> like
        ↪ SignalConditionerInterface {
            @display("p=100,120");
    }
    SignalConverter: <"SignalConverter"+type> like
        ↪ SignalConverterInterface {
            @display("p=100,190");
    }
    Transducer: <"Transducer"+type> like
        ↪ TransducerInterface {
            @display("p=100,260");
    }
```

Anhand der Submoduldefinition ist auch ersichtlich, weshalb jedes Modul des Sensors jeweils ein Interface implementieren muss. Für das parametrische Anlegen von Submodulen muss wenigstens ein Interface definiert werden, dem das Modul angehören muss, damit nicht jedes beliebige Modul an dieser Stelle eingesetzt werden kann.

#### 4.2.2.8 SensorNode

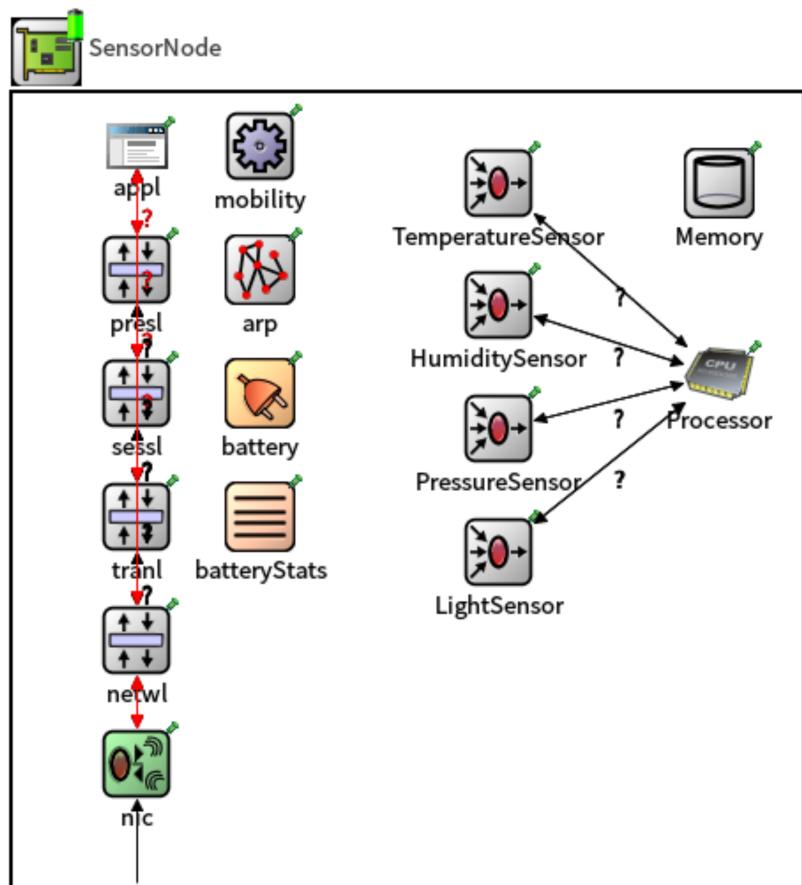
Der SensorNode ist das komplexeste Compound Modul in der Simulation, denn es beinhaltet die bisher beschriebenen Module Sensor, Prozessor, Memory und die in MiXiM enthaltenen Module für die Funkkommunikation und eine Batterie.

Die MiXiM-Module werden dem Sensorknoten durch das erben von dem Modul **WirelessNodeBatteryPlusTran** aus dem MiXiM-Framework zugewiesen, da dieses die benötigte Batterie und die Funkkommunikation bereitstellt. Alle anderen Module werden wie im Beispiel 5.15 definiert. Dieses ist dabei auf die Definition der Submodule reduziert.

Listing 4.15: SensorNode mit den neu definierten Modulen

```
module SensorNode extends WirelessNodeBatteryPlusTran like
    ↪ SensorNodeInterface
{
    parameters:
        // [...]
```

Abbildung 4.5: Aufbau des Compoundmoduls SensorNode



```
submodules:
    TemperatureSensor: TemperatureSensor if
        ↪ hasTemperatureSensor {
            @display("p=275,51");
            dataBandwidth = dataBandwidth;
            controlBandwidth = controlBandwidth;
    }
    HumiditySensor: HumiditySensor if hasHumiditySensor {
        @display("p=275,120");
        dataBandwidth = dataBandwidth;
        controlBandwidth = controlBandwidth;
    }
    PressureSensor: PressureSensor if hasPressureSensor {
        @display("p=275,190");
        dataBandwidth = dataBandwidth;
        controlBandwidth = controlBandwidth;
    }
    LightSensor: LightSensor if hasLightSensor {
        @display("p=275,260");
        dataBandwidth = dataBandwidth;
        controlBandwidth = controlBandwidth;
    }
    Memory: Memory {
        @display("p=400,51");
    }
    Processor: Processor {
        @display("p=400,159");
    }

connections:
//[...]
```

Durch das Setzen von `hasTemperatureSensor` und den analogen booleschen Variablen für die anderen Sensortypen kann für jeden Sensorknoten festgelegt werden, über welche Sensoren er verfügt. Das kann beispielsweise innerhalb der Definition des Netzwerkes geschehen.

- `TemperatureSensor`
- `HumiditySensor`

- LightSensor
- PressureSensor

### 4.2.2.9 Messages und Channels

Nachrichten sind das essenzielle Werkzeug, um in einem Netzwerk Kommunikation zu ermöglichen. Es gibt ein vordefiniertes Modul cMessage, welches dafür genutzt werden kann. Dieses beinhaltet notwendige Informationen wie zum Beispiel Sendermodul und -gate, Empfängermodul und -gate, Sende-, Empfangs- und Erstellzeit und mehr.

### 4.2.2.10 ExtendedMessage

Die ExtendedMessage erweitert diese Funktionalität um einige Informationen für die Statistik.

### 4.2.2.11 DatarateChannel

DatarateChannel ist ein von Omnet++ definiertes Modul. Es kann mit verschiedenen Parametern versehen werden, um den jeweiligen Umständen angepasst zu werden. Der Wichtigste ist dabei der double-Wert datarate, welcher in Bit pro Sekunde angegeben wird und festlegt, mit welcher Rate Daten durch den Channel gesendet werden können. Weiterhin kann auch ein delay und eine bit- beziehungsweise packet-error-rate festgelegt werden.

In der Simulation wurde dieses Modul genutzt, um die verschiedenen Kanäle für den Datenverkehr und die Steuerungskanäle zu trennen.

### 4.2.2.12 BasicWSN

Das Netzwerk BasicWSN dient als abstraktes Modul für alle Netzwerke, welche die Sensorikimplementierungen verwenden sollen. Es beinhaltet viele grundlegende Initialisierungen. So wird hier zum Beispiel die Umgebung definiert, welche durch die Klasse CustomWorldUtility beschrieben wird. Für diese werden Werte wie die Playground-Größe gesetzt, sowie einige Standardwerte definiert. So zum Beispiel der Speicherort für die Umgebungswerte.

Neu definierte Netzwerke mit Sensorknoten sollten stets von diesem Netzwerk erben. Alle Beispiele dieses Projekts tun dies ebenfalls.

### 4.2.3 Simulationsparameter und omnetpp.ini

Die omnetpp.ini ist eine Datei, in der Eigenschaften der Simulation geändert werden können, ohne dass man den Sourcecode bearbeiten und danach neu kompilieren muss. Es bietet sich daher an, alle Variablen und Konstanten, welche sich öfters ändern hier zu definieren. Die entscheidenden Parameter, welche die Eigenschaften der Sensoriksimulation beeinflussen, werden im Folgenden aufgeführt und erläutert.

- Energieverwaltung - kann global oder auch für jedes Modul einzeln definiert werden
  - currentConsumption - integer (in mA): Verbrauch über Zeit
  - energyConsumption - integer (in mWs): Verbrauch für eine Operation
- Energieverwaltung - Prozessorspezifisch, Energieverbrauch für die verschiedenen Prozessormodi
  - currentConsumptionNormal - integer (in mA)
  - energyConsumptionNormal - integer (in mWs)
  - currentConsumptionPowerSaving - integer (in mA)
  - energyConsumptionPowerSaving - integer (in mWs)
  - currentConsumptionHighPerformance - integer (in mA)
  - energyConsumptionHighPerformance - integer (in mWs)
- Energieverwaltung - die folgenden Parameter gelten für die Peripheriemodule (also alle außer dem Prozessor)
  - normalRatio - double: Verhältnis von currentConsumption und energyConsumption im normalen Modus
  - powerSavingRatio - double: Verhältnis von currentConsumption und energyConsumption im power saving Modus
  - highPerformanceRatio - double: Verhältnis von currentConsumption und energyConsumption im high performance Modus
  - battery.nominal - double (in mAh): Akkukapazität
- Metadaten/Eventintervalle - alle Events können komplett deaktiviert werden, indem der Wert auf 0 gesetzt wird

## 4 IMPLEMENTIERUNG

---

- dataGranularity - integer: je größer der Wert gewählt wird, um so grober ist die Berechnung der Sensorwerte, beim Werte n wird für einen Würfel der Seitenlänge = n jeweils 1 Messwert generiert
  - sensingInterval - integer (in s): mit welchem Zeitabstand Messungen werden durchgeführt
  - shiftProcessorModeNormalIntervall - integer (in s): wie lange wird der Prozessormodus normal behalten, bevor wieder gewechselt wird
  - shiftProcessorModePowerSavingIntervall - integer (in s): wie lange wird der Prozessormodus power saving behalten, bevor wieder gewechselt wird
  - shiftProcessorModeHighPerformanceIntervall - integer (in s): wie lange wird der Prozessormodus high performance behalten, bevor wieder gewechselt wird
  - collectStatisticsInterval - integer (in s): mit welchem Zeitabstand werden statistische Daten gemessen
  - readAndClearStorageInterval - integer (in s): mit welchem Zeitabstand wird der Memory ausgelesen und geleert
  - dataRecreationInterval - integer (in s): mit welchem Zeitabstand werden neue Umgebungswerte generiert
- Netzwerkparameter
    - createData - boolean: beim Start der Simulation neue Messwerte generieren
    - numHosts - integer: wie viele Sensorknoten befinden sich in der Simulation
    - noisyWorld - boolean: wenn auf true gesetzt, macht die Umwelt einige Ausgaben mit Informationen über Zustand und durchgeführte Aktionen
    - noisy - boolean: wenn auf true gesetzt, macht der Sensorknoten einige Ausgaben mit Informationen über Zustand und durchgeführte Aktionen
    - playgroundSizeX - integer (in m): Größe der X-dimension
    - playgroundSizeY - integer (in m): Größe der Y-dimension
    - playgroundSizeZ - integer (in m): Größe der Z-dimension
  - Sensorknoten

- dataBandwidth - double (in bps): Bandbreite mit der Daten innerhalb des Sensorknotens versendet werden
- controlBandwidth - double (in bps): Bandbreite mit der die Steuerungsleitungen des Prozessors senden
- Memory.storageSize - integer: definiert die Anzahl von Datensätzen, die das Memorymodul parallel speichern kann

## 4.3 Funktionsweise mit Beispielanwendungen

All die in diesem Kapitel beschriebenen Module wirken für die Simulation zusammen, um ein Netzwerk aus Sensorknoten zu schaffen, in dem die Knoten miteinander und mit ihrer Umgebung gemeinsam agieren können. Dazu werden nach dem Start der Simulation Umweltparameter bereitgestellt und eine festgelegte Anzahl von verschiedenen Sensorknoten erzeugt. Diese können sich anschließend bewegen oder ihre Position beibehalten. Sie können mit ihren Sensoren Werte der Umgebung erfassen und diese über Funk an andere Sensorknoten übertragen, sollte diese in der näheren Umgebung zur Verfügung stehen.

### allgemeine Beispiele

Die allgemeinen Beispiele zeigen auf, wie einfache Anwendungen unter Nutzung der Sensorknoten aussehen könnten. Dabei werden einfache Netzwerke mit unterschiedlichen Knoten gezeigt, und wie diese erzeugt werden können.

**BasicExample** Das Netzwerk BasicExample beinhaltet einige verschiedene Typen von Sensorknoten. Zur Verwendung von der Sensorik sollte das Netzwerk vom Modul BasicWSN erben, welches schon viele Parameter initialisiert, die für das korrekte Funktionieren notwendig sind. Alle weiteren Parameter, welche in der omnet.ini definiert sind, dienen dazu, die Simulation entsprechend des gewünschten Verhaltens anzupassen, wie zum Beispiel Batteriekapazität, Energieverbrauch einzelner Module und so weiter.

**AllNodes** AllNodes ist ein Beispielnetzwerk, in dem alle möglichen Kombinationen von Sensorknoten vorgeführt werden. Aus den 4 verschiedenen Sensortypen können 15 verschiedene Arten von Sensorknoten erzeugt werden, wenn man den Knoten ohne jeglichen Sensor nicht mitzählt, welcher in diesem Netzwerk aber ebenfalls vorhanden

ist. Das Beispiel dient zum Testen aller verschiedener Kombinationsmöglichkeiten, da sich die einzelnen Implementierungen leicht voneinander unterscheiden.

**TrafficGenExample** Das Netzwerk TrafficGenExample unterscheidet nur wenig vom BasicExample. Der entscheidende Unterschied ist, dass hierbei ein anderes Modul aus dem ApplicationLayer von MiXiM genutzt wurde, nämlich TrafficGen. Dieses kann sehr viele Nachrichten generieren und so den Kommunikationsverkehr sehr gut testen.

**SensorExample** Im Beispiel SensorExample wird die Funktionsweise von den Bauelementen der simulierten Sensorik und der Umgebung aufgezeigt. Dafür wurden verschiedene Parameter in der omnet.ini und in den Modulen so gewählt, damit dies gut abgelesen werden kann (Listing 5.16).

Listing 4.16: Parameter für das Beispiel SensorExample

```
**.sensingIntervall = 10000s
**.shiftProcessorModeNormalIntervall = 3000s
**.shiftProcessorModePowerSavingIntervall = 3000s
**.shiftProcessorModeHighPerformanceIntervall = 3000s
**.collectStatisticsIntervall = 1700s
**.readAndClearStorageIntervall = 34000s
**.dataRecreationIntervall = 13000s
**.noisyWorld = true;
**.noisy = true;
```

Die beiden Parameter noisy und noisyWorld dienen dazu, dass während der Simulation an den entscheidenden Stellen der Simulation Ausgaben gemacht werden. Diese geben Informationen darüber, welche Aktionen ausgeführt werden und wie der Zustand von einigen Modulen ist, so zum Beispiel wird bei Änderung der Speicherinhalt vom Memory ausgegeben und stets der aktuelle Prozessormodus angegeben.

Im Beispiel wird alle 10000 Sekunden von den Sensoren ein Messwert abgegriffen. Außerdem werden die vorliegenden Werte alle 13000 Sekunden geändert. Die verschiedenen Werte werden dann nacheinander im Memory abgelegt.

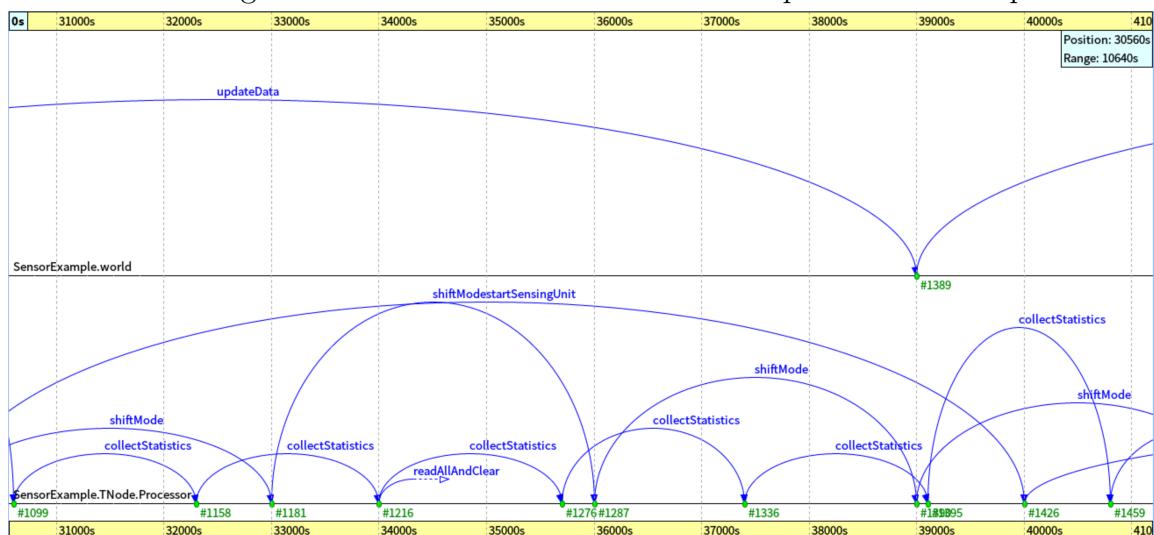
Listing 4.17: Ausgabe vom Prozessor bei t=40000s

```
Processor: received data:
Processor: Temperature: 10 (t=0s)
Processor: Temperature: 10 (t=10000s)
Processor: Temperature: 11 (t=20000s)
Processor: Temperature: 27 (t=30000s)
```

Da der Speicher im Beispiel nur Platz für 4 Datensätze bietet und auch nur ein einziger Sensor genutzt wird ist er nach 4 Messungen voll. Daher wird alle 34000 Sekunden, also nach jeweils 4 Messungen, der Speicher geleert. Die bis dahin gespeicherten Werte werden dann an den Prozessor gesendet. Eine Beispielausgabe des Prozessors aus einem Testlauf, in dem in dem nur ein einziger Knoten mit einem Temperatursensor genutzt wurde, sind in Listing 5.17 zu sehen. Diese Ausgabe wurde zum Zeitpunkt  $t=34000\text{s}$  getätigkt, ausgelöst durch den Event `readAndClearStorage`. Es wurde also gleichzeitig auch der Memory gelöscht. Direkt danach wurde der Speicher wieder mit neuen Datensätzen gefüllt.

In der Realität ist dies ein realistisches Szenario. Während sich die Knoten in einem Ruhemodus befinden, um Strom zu sparen, kann dennoch regelmäßig eine Messung durchgeführt und gespeichert werden. Wenn nun ein Zeitfenster erreicht wird, in dem Kommunikation im Netzwerk stattfindet, so werden gleich mehrere Messwerte auf einmal versendet.

Abbildung 4.6: Die verschiedenen Events im Beispiel SensorExample



Im Bild 5.6 sind die verschiedenen Events zu sehen, die im Sensor- und Worldmodul auftauchen. In der CustomWorldUtility wird nur einer der Events ausgelöst, welcher zum erneuten Erstellen von Sensordaten dient: `updateData` mit dem Intervall `dataRecreationInterval`.

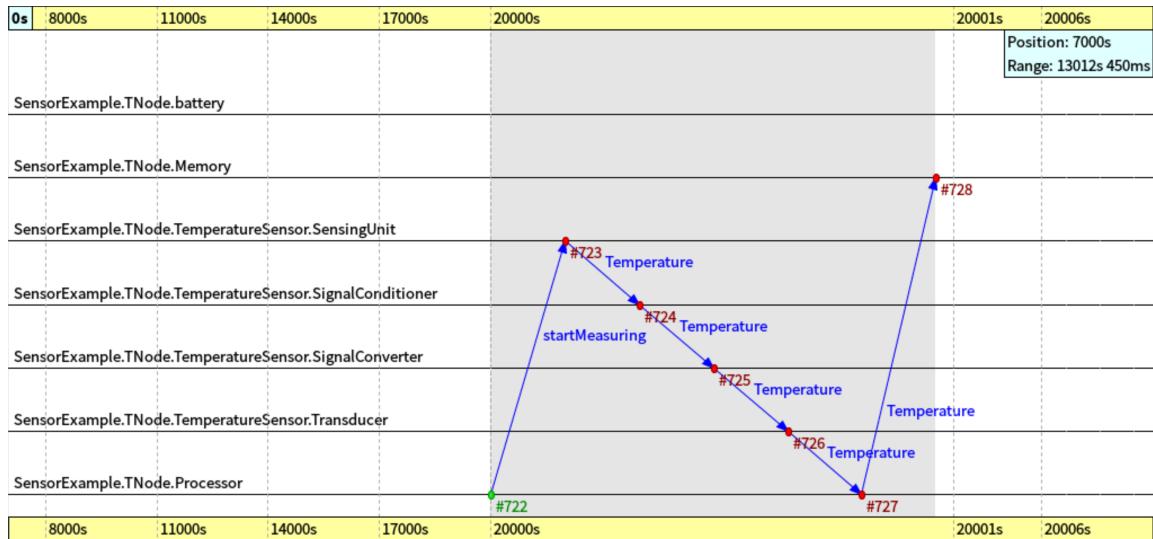
Die anderen Events finden im Prozessor statt. Sie dienen dazu bestimmte Arbeitsabläufe anzustoßen. Der wichtigste Event ist wohl `startSensingUnit`, dessen Zeitintervall durch `sensingInterval` festgelegt wird. Er dient dazu, einen Messvorgang in Gang zu setzen. Am Ende werden die gemessenen Werte stets im Memory gespeichert.

- `startSensingUnit`

## 4 IMPLEMENTIERUNG

---

Abbildung 4.7: Beispiel für den Nachrichtenverlauf beim Einlesen von Sensordaten



- sensingIntervall
- shiftMode
  - shiftProcessorModeNormalIntervall
  - shiftProcessorModePowerSavingIntervall
  - shiftProcessorModeHighPerformanceIntervall
- collectStatistics
  - collectStatisticsIntervall
- readAllAndClear
  - readAndClearStorageIntervall
- updateData
  - dataRecreationIntervall

In Bild 5.7 ist der Nachrichtenverlauf zu sehen, welcher durch den Event startMeasuring ausgelöst wird. Der Event wird innerhalb des Prozessormoduls gestartet. Dort wird anschließend eine Nachricht an die SensingUnits der zugehörigen Sensoren geschickt. Innerhalb der SensingUnit wird dann eine Messung durchgeführt. Diese Messdaten werden dann wiederum durch die verschiedenen Module des Sensor geleitet. Zuerst zum SignalConditioner, zum SignalConverter und zuletzt zum Transducer.

Nachdem diese Stationen durchlaufen wurden, wird der fertige Messwert zurück zum Prozessor gesendet. Dieser speichert dann zunächst den Wert im Memory ab.

### spezielles Verhalten

Im folgenden Abschnitt werden einige speziellere Netzwerke, als die bisherigen beschrieben. Diese führen nicht nur stupide Kommunikationen durch, sondern testen außerdem die verschiedenen Prozessormodi und zeigen, wie diese die Lebenszeit eines Knotens beeinflussen können.

Abbildung 4.8: Powermodi im Beispiel SleepVsNoSleep

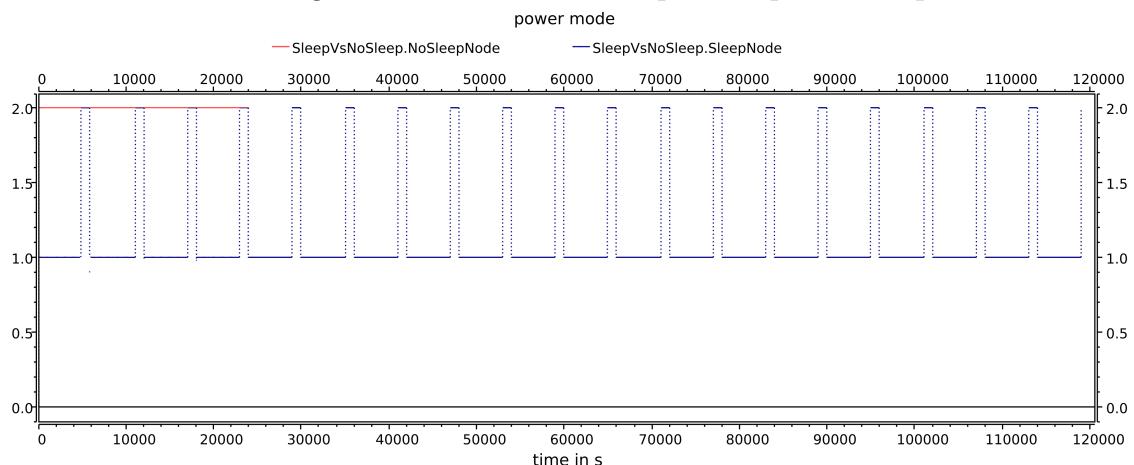
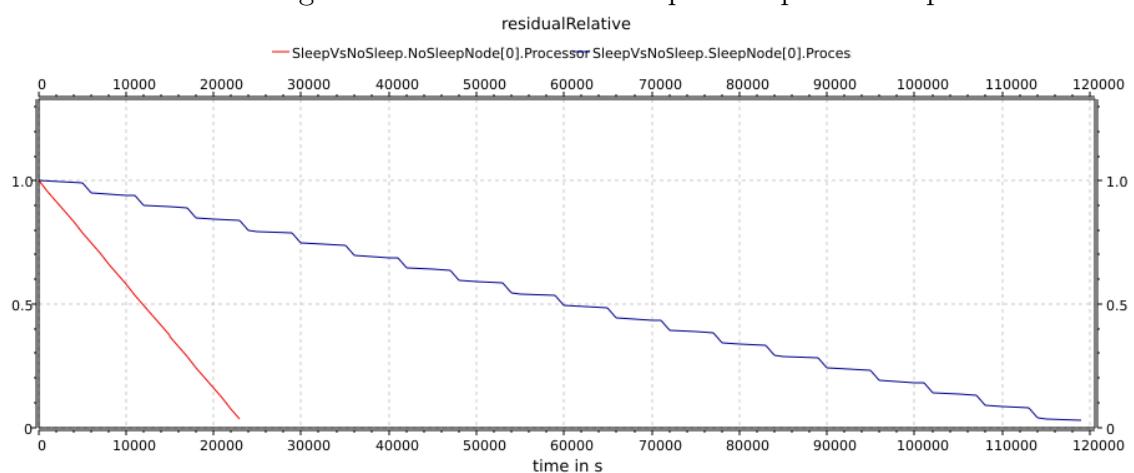


Abbildung 4.9: Ladezustand im Beispiel SleepVsNoSleep



**SleepVsNoSleep** Im Netzwerk SleepVsNoSleep befinden sich 2 beinahe identische Knoten. Die Bauteile auf beiden sind die selben, der einzige Unterschied besteht darin, dass einer der beiden Knoten nur einen Energiemodus benutzt, in dem ganz normal Energie im Stand-by-Modus verbraucht wird. Im Gegensatz dazu kann der andere Knoten zwischen 3 verschiedenen Modi zyklisch wechseln, wobei er in jedem dritten Intervall in einen schlafenden Zustand übergeht, bei dem deutlich Energie gespart werden kann.

Die Abbildung 5.8 zeigt die verschiedenen Powermodi. Dabei ist entscheidend, dass der Energiesparmodus durch den Wert 1 repräsentiert wird. Der rote Knoten wechselt in diesem Beispiel niemals in den Energiesparmodus, sondern bleibt während der gesamten Zeit in einem Modus. Der andere Knoten wechselt jeweils für 5000 Sekunden in den Energiesparmodus. Nur dazwischen wechselt er immer für kurze Zeit in einen aktiven Modus.

Das Resultat ist in Abbildung 5.9 sofort ersichtlich. Der rote Knoten, welcher nie in den schlafenden Modus wechselt, hat eine erheblich kleinere Lebenszeit. Das Verhältnis der Lebenszeit ist natürlich davon abhängig, wie viel Energie durch den Energiesparmodus gespart werden kann.

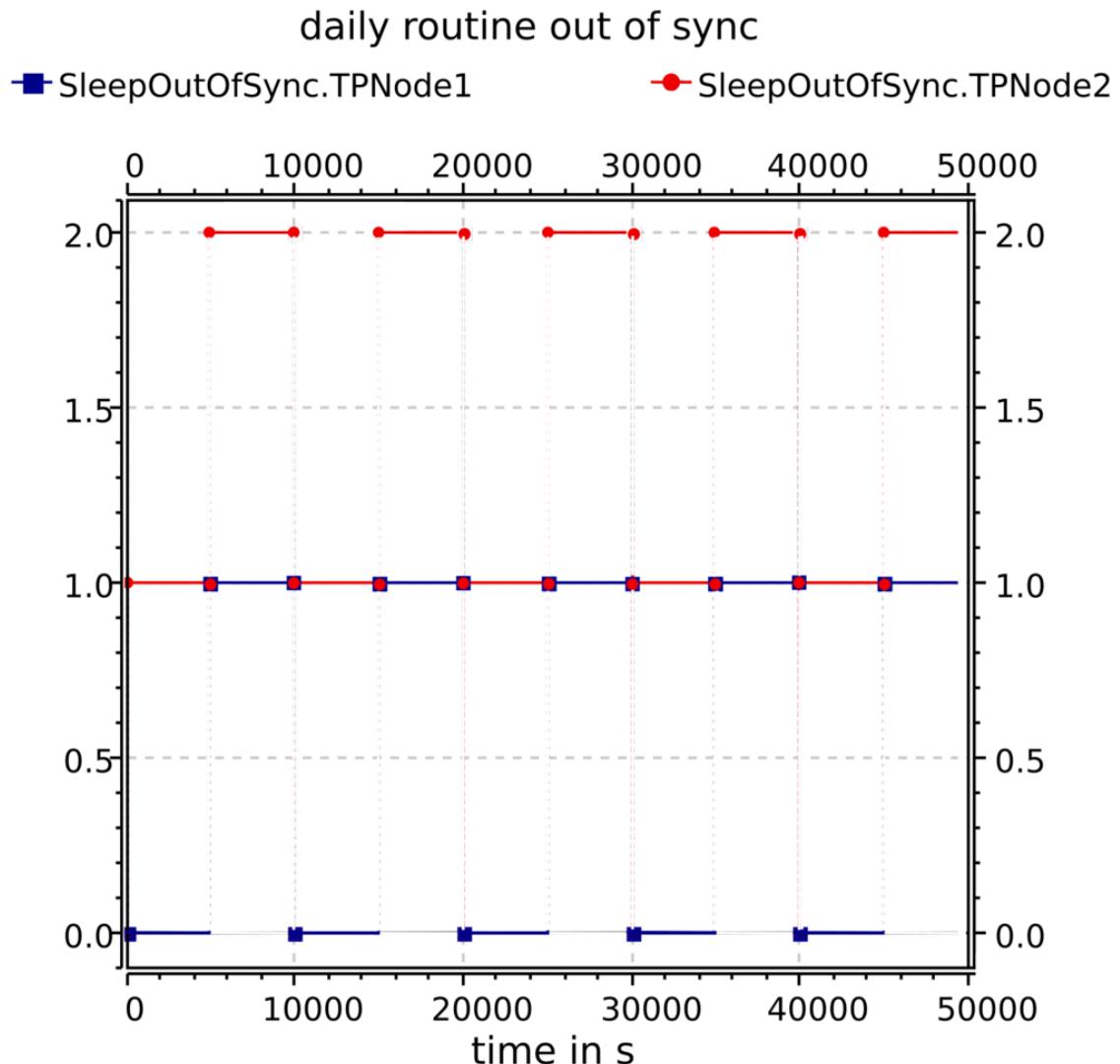
**SleepOutOfSync** Das Beispiel namens SleepOutOfSync zeigt ebenso wie SleepVsNoSleep die Verwendung der Prozessor-Energie-Modi. Allerdings wird bei diesem Beispiel gezeigt, warum der Wach-und-Schlaf-Zyklus von den verschiedenen Knoten, welche sich in einem Sensornetzwerk befinden zeitlich aufeinander abgestimmt werden muss.

Dazu ist lediglich ein kleines Netzwerk von 2 Knoten vorhanden, welche genau zeitversetzt zwischen dem Energiesparmodus, in dem nichts gesendet oder empfangen werden kann und einem normal Modus wechseln. In dieser extremen Variante würde das Netzwerk gar nicht erst vorhanden sein, da beide Knoten niemals von der Existenz des jeweils anderen erfahren würden.

Das zeigt, weshalb verschiedene Knoten eines Netzwerkes in der Praxis einen synchronen Wach-und-Schlaf-Zyklen besitzen müssen, damit keine unnötige Kommunikation stattfindet, an der nicht alle Knoten beteiligt sind.

In Abbildung 5.10 ist dieser Ablauf visualisiert. Dabei kann im Modus mit dem Wert 1 keine Kommunikation stattfinden. Wie man sehen kann, wechseln sich die beiden Knoten genau zwischen aktiven und inaktiven Modi ab. Es kann dadurch zu keiner Kommunikation kommen.

Abbildung 4.10: Powermodi im Beispiel SleepOutOfSync





## 5 Auswertung

**Anforderungen** In der Arbeit wurden verschiedene Ziele verfolgt. So sollte zum einen der aktuelle Stand der Technik dargestellt werden. Im Kapitel 2 wurden verschiedene Methoden für Evaluation von Modellen aufgeführt und der Aufbau von Sensoren und deren Verwendung erläutert.

Außerdem sollten Grundlagen über das genutzte Framework vermittelt werden. Nach der Gegenüberstellung verschiedener Netzwerksimulationen wurde das Framework Omnet++ zusammen mit den erweiternden Funktionen aus dem MiXiM-Framework ausgewählt. Die Grundlagen zur Verwendung dieser wurden in Kapitel 3.3 und 3.4 beschrieben.

Das Hauptziel der Arbeit war die Implementierung einer Sensormodellierung. Dazu sollten zum einen verschiedene Sensorknoten simuliert werden. In Kapitel 4 ist die Implementierung näher beschrieben.

Es wurden 4 verschiedene Arten von Sensoren geschaffen, die auf einem Sensorknoten verwendet werden können. Diese können je nach Sensortyp verschiedene Umgebungsparameter bestimmen, wie zum Beispiel Temperatur. Damit die Sensoren Messdaten zur Verfügung haben, musste zusätzlich die Umgebung der Simulation implementiert werden. Dafür ist ein Modul entstanden, welches für die Koordinaten Daten generieren kann. Es repräsentiert einen Teil der realen Welt mit beispielsweise verschiedenen Temperaturen an verschiedenen Positionen. Die Sensoren der Knoten haben die Möglichkeit für ihre jeweilige Position die entsprechenden Messwerte auszulesen. Im Laufe der Simulation können in vorgegebenen Intervallen neue Werte für die Umgebungsparameter generiert werden.

Durch die Implementierung der 4 verschiedenen Sensortypen ist es möglich daraus eine Reihe verschiedener Sensorknoten zu bilden, nämlich  $2^4$ , also 16 Stück. Diese verschiedenen Sensorknoten haben jedoch Module für die Funkkommunikation und die Energieverwaltung gemeinsam. Dafür wurden fertige Module aus dem MiXiM-Framework genutzt. Das Funkmodul ermöglicht die drahtlose Kommunikation zwischen den verschiedenen Sensorknoten. Nur so können später verschiedene Routingverfahren getestet werden, was eine mögliche Verwendung der Simulation darstellt. Sehr relevant für Tests von mobilen Bauteilen ist das Simulieren des Energiehaushalts. Dafür wurde das Batteriemodul aus dem MiXiM-Framework zu Hilfe genommen. Dieses stellt die Energiequelle selbst bereit. Für alle neu entstandenen Module wurde zusätzlich der Zugang zur Batterie ermöglicht und eine Präsentation des Energieverbrauchs dieser Teile. Diese verbrauchen permanent Strom im Stand-by-Modus. Zusätzlich kann ein fester Betrag definiert werden, welcher beim Ausführen

energieintensiver Operationen von der Restladung des Akkus abgezogen werden soll. Damit die Simulation auch ausgewertet werden kann, sollten die relevanten Daten gespeichert und nach dem Ende eines Simulationslaufes auch visualisiert werden. Omnet++ bietet mit der Einbindung von Gnuplot bereits ein Werkzeug zum Darstellen der gemessenen Daten. Es werden verschiedene Vektor- und Skalarwerte während einer Simulation aufgenommen. Dazu gehört zum Beispiel der aktuelle Ladezustand der Batterie zu den verschiedenen Zeitpunkten der Simulation.

Eine letzte Anforderung war das Testen der Implementierung. Dazu wurden verschiedene Beispielnetzwerke erstellt. Diese benutzen die verschiedenen Module und zeigen, wie man mit verschiedenen Einstellungen der Parameter unterschiedliche Ergebnisse erzielen kann. Die Beispiele sind im Abschnitt 4.3 beschrieben.

**Zusätzliches** Neben den geforderten Teilen der Sensorik wurden für die Simulation der Sensorknoten zusätzlich ein Speicher- und ein Prozessormodul implementiert. Der Speicher dient dazu, verschiedene Messwerte vorübergehend innerhalb des Knotens abzulegen, bevor diese über die Funkschnittstelle versendet werden. Dies ist besonders dann sehr sinnvoll, wenn entweder verschiedene Sensoren auf dem Knoten vorhanden sind und nicht alle Messung exakt zeitlich fertig sind oder wenn das Prozessormodul entsprechend genutzt wird. Dieses enthält nämlich eine Implementierung von verschiedenen Powermodi, wodurch es dem Modul möglich ist, in einen Ruhezustand zu wechseln. Wenn dieser Zustand jegliche Kommunikation unterbinden soll, aber innerhalb der Phase dennoch Messungen durchgeführt werden ist es nicht vermeidbar, die gemessenen Werte zunächst zu speichern. Wenn später wieder eine Kommunikation stattfindet, können dann mehrere Werte innerhalb einer Nachricht übermittelt werden.

**Grenzen** Die Messdaten werden zum aktuellen Zeitpunkt komplett zufällig generiert. Nahe beieinandergelegene Punkte beeinflussen einander genauso wenig, wie weit voneinander entfernte. Auch die verschiedenen Arten von Messwerten haben keinen Einfluss aufeinander. So führt zum Beispiel eine höhere Helligkeit, was auf eine erhöhte Sonneneinstrahlung an einem Punkt schließen lässt, nicht zu einer höheren Temperatur als in einer schattigen Region. Die Daten sind außerdem sehr grob. Es kann zwar innerhalb der Parameter festgelegt werden, wie genau die generierten Messwerte sein sollen, aber bereits bei einem Messwert pro Quadratmeter bei einem 3-dimensionalen Würfel von 100m bis 200m Kantenlänge wird die Simulation beim Start extrem langsam, weil das Auslesen und Schreiben derart großer Daten aus einer xml-Datei alles andere als performant ist.

Diese Tatsache ist jedoch zunächst einmal nicht sehr schlimm, da die Qualität der Messwerte eher eine untergeordnete Rolle spielt. Wichtiger sind die Vorgänge inner-

---

halb des Sensorknotens und der Umgebung, unabhängig von den Werten der Messdaten.

**Ausblick** In Zukunft ist es denkbar, dass die Messdaten nicht mehr über eine xml-Datei gelesen werden. Stattdessen wäre es möglich, eine Datenbank in die Simulation einzubinden. Diese würde wesentlich performantere lesende und schreibende Zugriffe erlauben. Zusätzlich dazu könnten dann reale Werte von Messungen einer echten Umgebung importiert werden, vielleicht sogar über einen Zeitintervall gemessen. Das würde beides, schnelle Messungen und reale Werte ermöglichen.

Die zur Verfügung gestellten Prozessormodi wirken sich bisher nur auf die Sensormodule und deren Energiehaushalt aus. Wenn die Kommunikation zwischen den Knoten in einem großen Netz simuliert wird, könnten Phasen definiert werden, in denen die Knoten in einen Ruhezustand übergehen, in denen keine Kommunikation mehr stattfindet. Dann könnte das gesamte Netz gemeinsame Zyklen von Ruhe- und Wachzustand durchführen, um Energie zu sparen.

Als Erweiterung des Ruhezustands wäre anschließend wiederum denkbar, dass ein Wake-up-Receiver implementiert wird. Dieser könnte es sogar ermöglichen, dass Knoten von selbst gar nicht mehr aus dem Ruhezustand aufwachen, sondern nur, falls das von außen gesteuert wird.

Diese Steuerung könnte wiederum ein anderes Modul durchführen. Es könnte ein Netzwerkknoten erstellt werden, welcher als Datensenke und Steuerung des Netzwerkes fungiert. Dieses Modul könnte beispielsweise auf manuelle Eingabe oder zu definierten Zeitpunkten Nachrichten an die Wake-up-receiver senden und die zuletzt gemessenen Daten einsammeln. Wenn alle Knoten erwacht sind und die Datensenke alle relevanten Messdaten eingesammelt hat könnten wiederum Nachrichten geschickt werden, welche alle Knoten zurück in den Ruhezustand versetzen.



## 6 Zusammenfassung

In der Arbeit wurde die Implementierung von einer Sensorik für die Simulationsumgebung Omnet++ umgesetzt. Dafür wurden Module für 4 verschiedene Arten von Sensoren geschaffen: Temperatur, Luftdruck, Helligkeit und Luftfeuchtigkeit. Deren Sensorik wurde wiederum in 4 verschiedene Module geteilt: SensingUnit, SignalConditioner, SignalConverter und Transducer. Diese 4 Module bilden die eigentliche Sensorik ab und dienen dazu Messwerte aus der Umgebung auszulesen und diese verwertbar zurückzugeben.

Für die Steuerung dieser Sensorik wurde ein Prozessormodul implementiert. Dieses kann Messungen in Gang setzen und hinterher die gemessenen Daten in einem Memorymodul ablegen, bis diese gebraucht werden. Außerdem ist der Prozessor in der Lage zwischen verschiedenen Energiemodi zu wechseln und somit Energie zu sparen. Neben diesen selbst definierten Modulen beinhaltet ein Sensorknoten noch ein Funkmodul und eine Batterie, welche jeweils aus dem MiXiM-Framework für Omnet++ übernommen wurden. Alle implementierten Bauteile haben dabei verschiedene Parameter, die den eigenen Stromverbrauch regeln und Zugang zur Batterie des Knotens, um den Energieverbrauch realistisch abbilden zu können.

Der Prozessor enthält weiterhin viele Parameter für die statistische Auswertung von Simulationen. Diese stellen beispielsweise Informationen über die Prozessormodi, den Ladezustand der Batterie oder auch den Energieverbrauch von einzelnen Modulen bereit. Diese Informationen können wiederum grafisch ausgewertet werden, um einen möglichst hohen Informationsgehalt aus den Daten der Simulationen erhalten zu können.

Alle implementierten Funktionen wurden bestmöglich in Form von Beispielnetzwerken umgesetzt. Diese zeigen die Verwendung der verschiedenen Module mit unterschiedlichen Einstellungen und wie diese in Netzwerken interagieren können.



## Literatur- und Webverzeichnis

- [1] *Doxygen*. Online unter [www.doxygen.org/](http://www.doxygen.org/); zuletzt besucht am 24. April 2015.
- [2] *Eclipse*. Online unter <https://www.eclipse.org/>; zuletzt besucht am 06. Mai 2015.
- [3] *Emacs*. Online unter <http://www.gnu.org/software/emacs/>; zuletzt besucht am 26. April 2015.
- [4] *Git*. Online unter <http://git-scm.com/>; zuletzt besucht am 06. Mai 2015.
- [5] *Github*. Online unter <https://github.com>; zuletzt besucht am 06. Mai 2015.
- [6] *Gnuplot*. Online unter <http://www.gnuplot.info/>; zuletzt besucht am 26. Mai 2015.
- [7] *MiXiM API Reference*. Online unter <http://mixim.sourceforge.net/doc/MiXiM/doc/doxy/>; zuletzt besucht am 08. Mai 2014.
- [8] *MiXiM Official Website*. Online unter <http://mixim.sourceforge.net/>; zuletzt besucht am 08. Mai 2015.
- [9] *NED Language Beschreibung auf ieee.org*. Online unter <http://www.ewh.ieee.org/soc/es/Nov1999/18/ned.htm>; zuletzt besucht am 06. Mai 2015.
- [10] *NS-Simulator auf Wikipedia*. Online unter [http://en.wikipedia.org/wiki/NS\\_%28simulator%28](http://en.wikipedia.org/wiki/NS_%28simulator%28); zuletzt besucht am 25. April 2015.
- [11] *Omnet++ Manual*. Online unter <http://www.omnetpp.org/doc/omnetpp/manual/usman.html>; zuletzt besucht am 06. Mai 2015.
- [12] *Omnet++ Official Website*. Online unter <http://www.omnetpp.org>; zuletzt besucht am 08. Mai 2015.
- [13] *OpenWNS Wrowser*. Online unter <https://launchpad.net/openwns-wrowser>; zuletzt besucht am 25. April 2015.
- [14] *Topology Generator des NS-Simulators*. Online unter [https://www.nsnam.org/wiki/Topology\\_Generator](https://www.nsnam.org/wiki/Topology_Generator); zuletzt besucht am 26. April 2015.

- [15] *Wikipediaeintrag Sensornetz*. Online unter <http://de.wikipedia.org/wiki/Sensornetz>; zuletzt besucht am 08. Mai 2014.
- [16] BÜLTMANN, DANIEL, MACIEJ MÜHLEISEN und SEBASTIAN MAX (CHAIR OF COMMUNICATION NETWORKS RWTH AACHEN UNIVERSITY): *openWNS*. Online unter <http://openwns.org>; zuletzt besucht am 04. Mai 2015.
- [17] HARDT, PROF. DR. WOLFRAM: *Veranstaltungen zu Hard-/Software Co-design 2*. Online unter <https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/4563599363>; zuletzt besucht am 04. Mai 2015.
- [18] RILEY, GEORGE F. und THOMAS R. HENDERSON: *The ns-3 Network Simulator*. Online unter <http://www.nsnam.org/>; zuletzt besucht am 26. April 2015.
- [19] SOMMER, JÖRG, JOACHIM SCHARF, (INSTITUTE OF COMMUNICATION NETWORKS und COMPUTER ENGINEERING DER UNIVERSITÄT STUTTGART): *IKR Simulation Library (IKR SimLib)*. Online unter <http://www.ikr.uni-stuttgart.de/IKRSimLib/>; zuletzt besucht am 04. Mai 2015.
- [20] VODEL, MATTHIAS, MATTHIAS SAUPPE, MIRKO CASPAR und WOLFRAM HARDT: *SimANet - A Large Scalable, Distributed Simulation Framework for Ambient Networks*. JOURNAL OF COMMUNICATIONS, 3(7):11–19, 2008.

## **Selbstständigkeitserklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den 29. Mai 2015

---

Thomas Rückert