



TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Professur für Technische Informatik

Bachelorarbeit

Modellierung und Integration von Sensorknoten in einer
Simulationsumgebung

Thomas Rückert

Chemnitz, den 9. Dezember 2014

Prüfer: Prof. Dr. Wolfram Hardt

Betreuer: Dipl.-Inf. Mirko Lippmann

Thomas Rückert,

Modellierung und Integration von Sensorknoten in einer Simulationsumgebung

Bachelorarbeit, Fakultät für Informatik

Technische Universität Chemnitz, Dezember 2014

Abstract

In dieser Arbeit soll die Simulation von Sensorknoten in einem Netzwerk untersucht werden. Dabei sollte zum einen die Kommunikation zwischen den verschiedenen Knoten, von denen es viele und verschiedene geben soll betrachtet. Diese führen eine kabellose Kommunikation miteinander. Die einzelnen Knoten besitzen Sensoren, die verschiedene Umweltparameter auslesen. Die Bereitstellung dieser Umweltparameter in der Simulationsumgebung ist auch ein Teil der Implementierung. Außerdem sind die Knoten batteriebetrieben und auch deren Energieverwaltung wurde betrachtet. Nach Test und Modellierung ist die Auswertung und Visualisierung von Simulationsdaten von entscheidender Rolle. Als Simulationsumgebung und -sprache wurde Omnet++ mit dem Framework MiXiM, welches Grundfunktionen für mobile Knoten mit kabelloser Kommunikation bereit stellt.

Inhaltsverzeichnis

Abbildungsverzeichnis	iii
Tabellenverzeichnis	vii
1 Einleitung	1
2 Vorbetrachtungen	3
2.1 Evaluation von Systemen	3
2.2 Sensoren	4
2.3 Sensornetzwerke	7
2.4 Wake-up Receiver	7
3 Simulationsumgebung	9
3.1 wichtige Simulationsumgebungen	9
3.2 Vergleich	11
3.3 Omnet++	12
3.3.1 Einleitung	12
3.3.2 NED language	13
3.3.3 Einige Techniken, Funktionen und wichtige Module	14
3.4 MiXiM-Framework als Omnet++-Erweiterung	21
3.4.1 Einleitung	21
3.4.2 Einige wichtige Module	21
4 Implementierung	25
4.1 Einleitung	25
4.2 Aufbau und Struktur	25
4.2.1 Klassenübersicht	25
4.2.2 Übersicht NED-Module	37
4.3 Funktionsweise	47
5 Zusammenfassung	49
Literatur- und Webverzeichnis	51

Abbildungsverzeichnis

2.1	Deutschlandkarte der mittleren Temperatur zwischen 1961 und 1990 .	5
2.2	Deutschlandkarte: Beispielverteilung der Luftfeuchtigkeit	6
3.1	GUI bei der Ausführung einer Simulation	12
3.2	Oberfläche der Entwicklungsumgebung mit Beispiel für die NED-Integration	16
3.3	Beispiel cMessage als Event	20
4.1	Klassenübersicht	26
4.2	CustomLinearMobility: Vererbung	26
4.3	CustomWorldUtility: Vererbung	26
4.4	CustomWorldUtility: Member	27
4.5	CustomWorldUtility: Methoden	28
4.6	MyWirelessNode: Vererbung	32
4.7	NodeType: Vererbung	34
4.8	SensorType: Member	35
4.9	SimpleClasses: Member	35
4.10	ExtendedMessage: Vererbung	36
4.11	Verlauf der Nachrichten und Methodenaufrufe	48

Liste der noch zu erledigenden Punkte

überarbeiten	3
ergänzen	4
ergänzen	7
ergänzen	7
simanet beschreiben	10
omnet kurzbeschreibung(mixim)	10
Ergänzen	23
zusammenfassung schreiben	49

Tabellenverzeichnis

3.1	Übersicht Simulatoren	11
3.2	NED Schlüsselbegriffe	15
3.3	Übersicht über einige Funktionen von cMessage	18

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 kleiner und 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.

In der folgenden Arbeit soll ein solches Netz mit Hilfe von Omnet++[11] simuliert werden.

2 Vorbetrachtungen

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++[11] mithilfe des MiXiM-Frameworks[7] 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 ohne Implementierung und daher ohne zu messen eine kostengünstige Variante zum Bestimmen von Designparametern. 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 wenn man merkt, dass die Implementierung wohl doch nicht die Optimale für ein gewünschtes Ziel ist.

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.

überarbeiten

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 ver-

muten lässt können im analogen Fall zu jeder Zeit Zustandsübergänge stattfinden, im diskreten dagegen nur zu bestimmten Zeitpunkten.

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 an Timing, Größe und Kosten stellt. Prototypen 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. 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 und nicht darauf, wie die einzelnen Bauteile technisch aufgebaut sein könnten oder sollten.

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: Deutschlandkarte der mittleren Temperatur zwischen 1961 und 1990

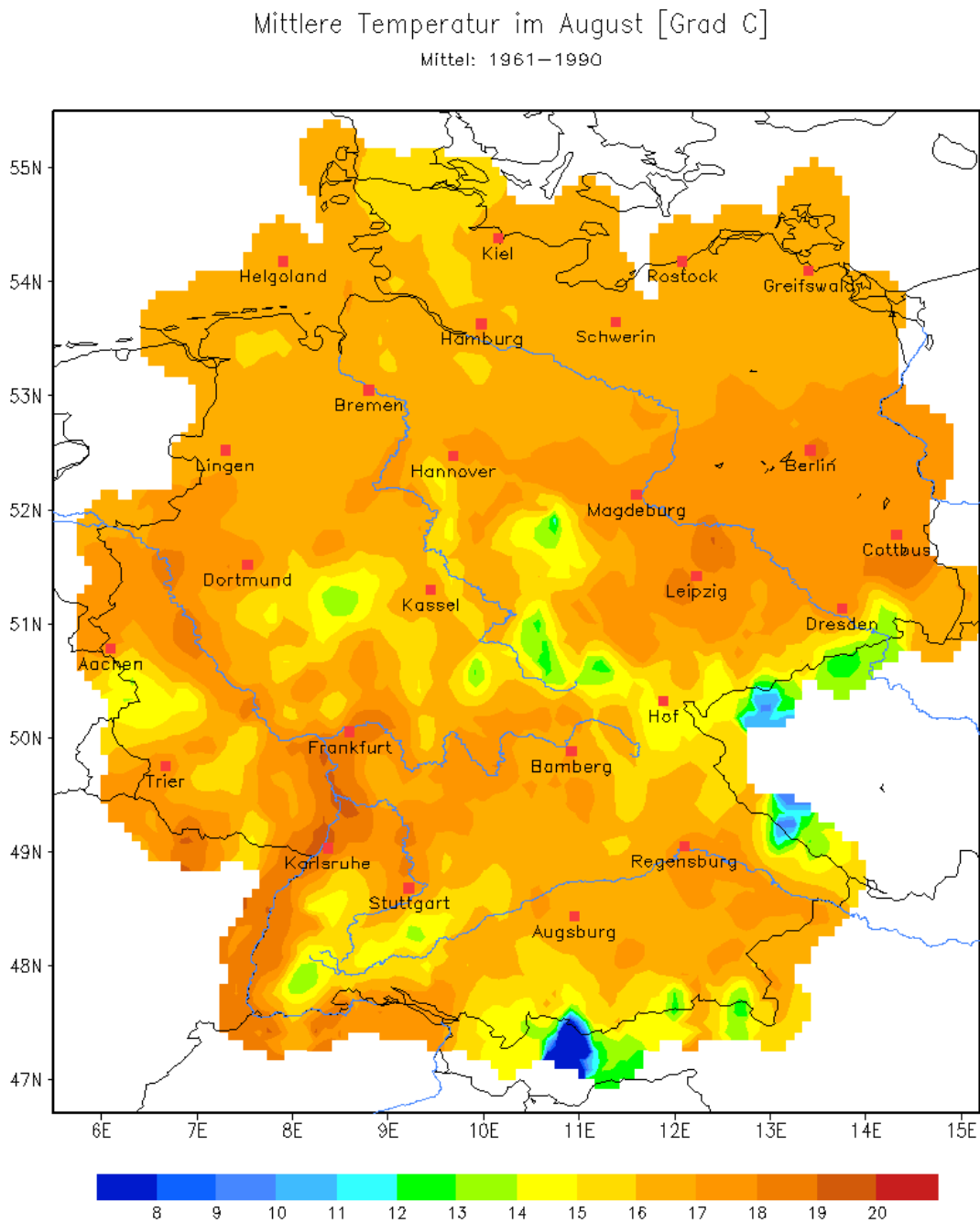
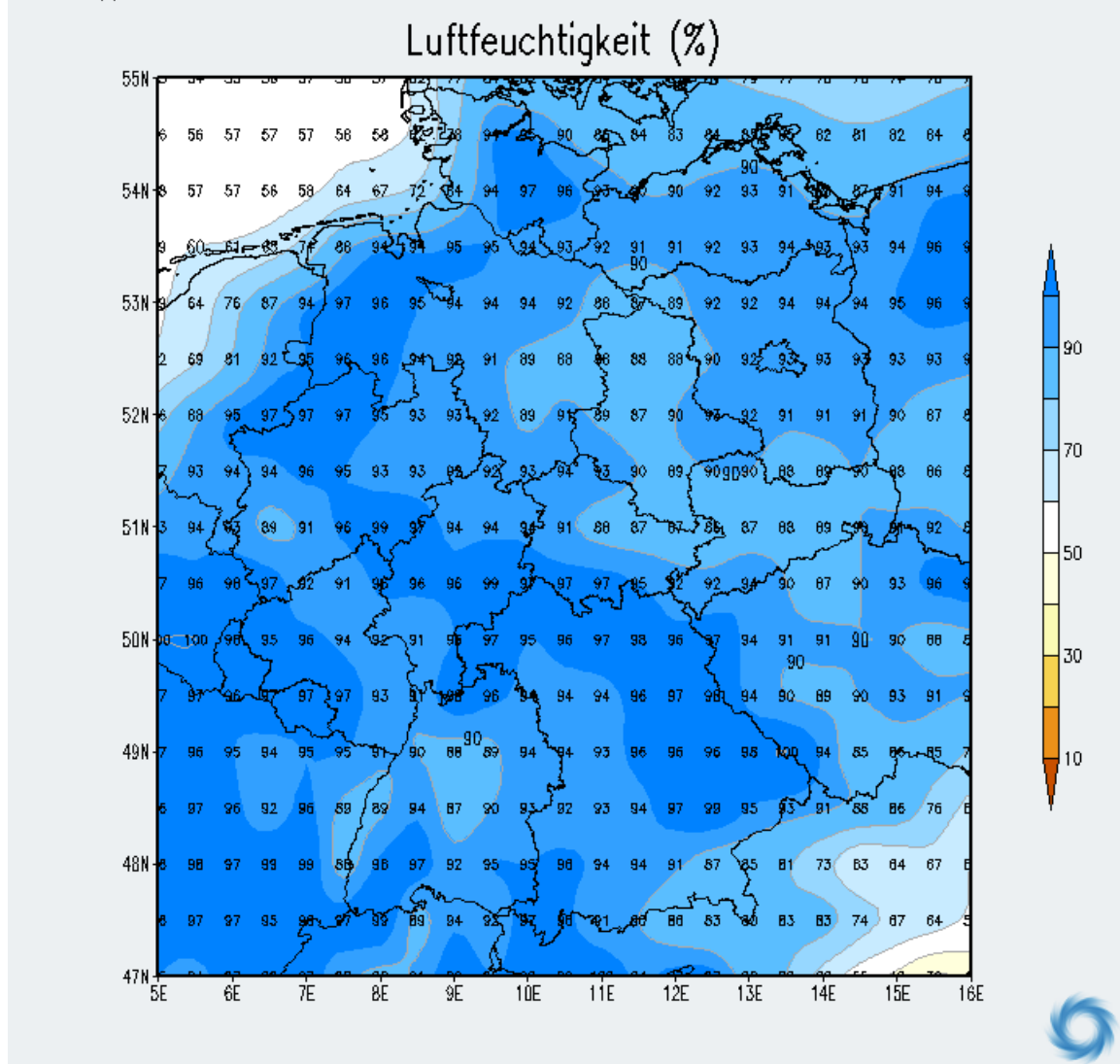


Abbildung 2.2: Deutschlandkarte: Beispielverteilung der Luftfeuchtigkeit

(c) www.neuwetter.de

Thu, 11DEC2014 06Z



2.3 Sensornetzwerke

ergänzen

2.4 Wake-up Receiver

ergänzen

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 kann. 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)[18] 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 existiert 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 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[15] OpenWNS ist ein Simulator für kabellose Kommunikation, entwickelt von der Universität Aachen. Das Projekt steht kostenlos zur Nutzung bereit und ist mit der LGPLv2 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[12], 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[16] ns-3 ist ein freier Netzwerksimulator, der unter der GPLv2 lizenziert ist. Das Erstellen der Simulationen funktioniert mit Hilfe 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[13] können per GUI Netzwerke angelegt werden und anschließend in C++ oder Python-Code umgewandelt werden.

simanet
beschreiben

Simanet[?]

omnet kurz-
beschrei-
bung(mixim)

Omnet++[11]

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 Netzwerkknoten. 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.

Tabelle 3.1: Übersicht Simulatoren

	Omnet++	IKR SimLib	OpenWNS	NS-3
freie Lizenz	✓	✓(LGPL)	✓(LGPLv2)	✓(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

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.

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 mit Hilfe 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 zu Gunsten von Omnet++ gefallen. Die Simulationsumgebung wird im folgenden Abschnitt ausführlicher beschrieben.

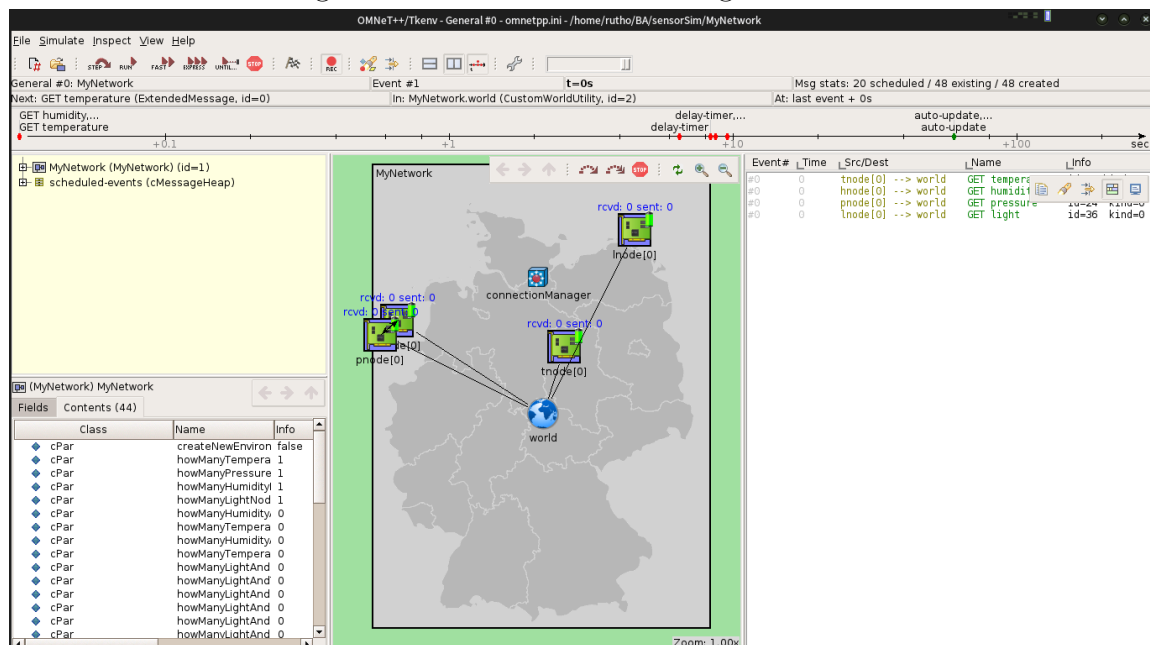
3.3 Omnet++

3.3.1 Einleitung

Omnet++[11] 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 mit Hilfe 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[8] 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 Moduleteile in einem großen Modul zu vereinen.

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

Listing 3.1: einfaches Modul: LED

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

Listing 3.2: einfaches Modul: Knopf

```
simple Knopf
{
    gates:
        input buttonStateChange;
        output signal;
}
```

Listing 3.3: Compound Modul

```
module Knoten
{
    parameters:
        string name="KnotenMitLedUndKnopf ";
    submodules:
        Blinker: LED {
        }
        Button: Knopf {
        }
    gates:
        input in;
        output out;
}
```

Listing 3.4: einfaches Netzwerk

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

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.

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 verbun-

Tabelle 3.2: NED Schlüsselbegriffe

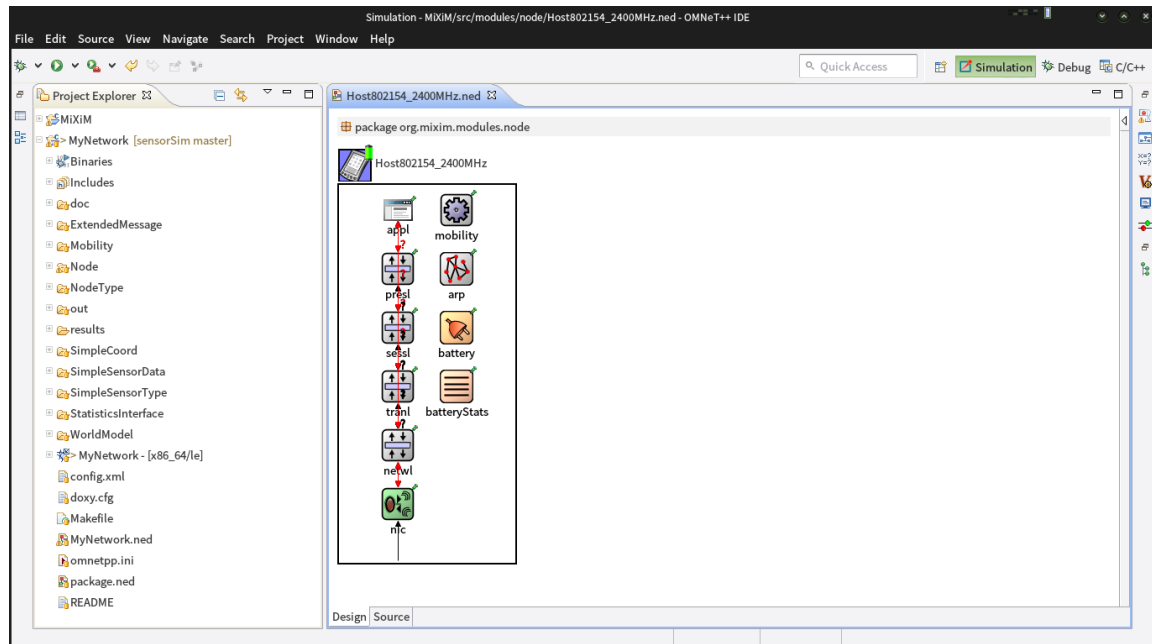
Kategorie	Begriffe	Funktion
Modulart	network	ein Netzwerk, 1 pro Simulation
	simple	ein einfaches, eigenständiges Modul
	module	ein compound Modul, kann Submodule haben
	channel	beschreibt eine Verbindung zwischen Gates
	channelinterface	ein Interface für Channel
	moduleinterface	ein Interface für Module
Sections von Modulen	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
Typen	int, string, double, bool	
	xmldoc	speichert den Pfad eines XML-Files
	xml	speichert XML
Verbindungen	allowunconnected	erlaubt Kommunikation zwischen Gates ohne definierte Verbindung
	input	definiert ein Gate als eingehend
	output	definiert ein Gate als ausgehend
	inout	legt je ein in- und output Gate an
weitere	@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	

den 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.

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



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 bereit gestellt, 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 Modules aufgerufen. Es kann 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 Speicherintensiver 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 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.

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");
}

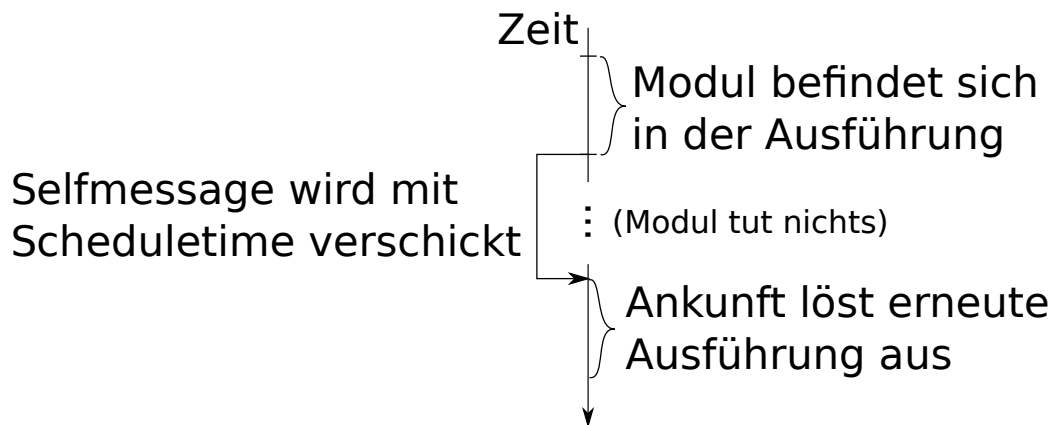
```

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 Schedulingtime 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 3.3).

Abbildung 3.3: 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[7] 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. Mann kann mit Hilfe 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.

3.4.2.4 Wireless

BaseArp src/modules/node/Host80211.ned

Host80211 IEEE 802.11 Norm für Funknetzwerke

Ergänzen

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.

4.2 Aufbau und Struktur

4.2.1 Klassenübersicht

Die Klassenübersichten wurden teilweise mit Hilfe von doxygen[1] erstellt.

4.2.1.1 CustomLinearMobility

Diese Klasse erbt direkt von der in MiXiM definierten LinearMobility. Sie ist um einige Parameter erweitert, die es ermöglichen, dass die Knoten beschleunigen können und eine maximale Geschwindigkeit **maxSpeed** erreichen, sollte eine definiert sein. Der Parameter **maxSpeed** kann ebenso auf 0 gesetzt werden, um die Knoten von mobil in stationär umzuwandeln.

4.2.1.2 CustomWorldUtility

Die Klasse CustomWorldUtility ist eine der wichtigsten für die Simulation. Sie repräsentiert den **Playground**, 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 Playground bereit.

Abbildung 4.1: Klassenübersicht



Abbildung 4.2: CustomLinearMobility: Vererbung

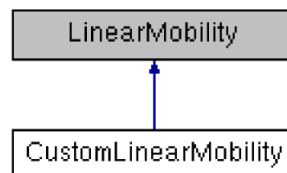
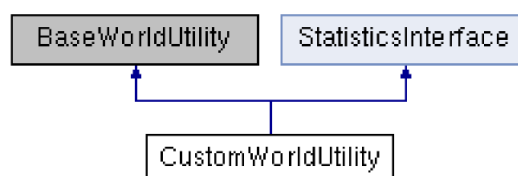


Abbildung 4.3: CustomWorldUtility: Vererbung



Zusätzlich dazu stellt die Klasse selbst die notwendigen Parameter für die Umwelt bereit. Nach dem Starten der Simulation steht darin jeweils ein zweidimensionales Array pro Sensortyp bereit: `temperatureArray`, `pressureArray`, `humidityArray` und `lightArray`. Diese 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 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` und `sizeY` gespeichert.

Abbildung 4.4: CustomWorldUtility: Member

int	<code>numNodes</code>
int**	<code>temperatureArray</code>
int**	<code>pressureArray</code>
int**	<code>humidityArray</code>
int**	<code>lightArray</code>
int	<code>sizeX</code>
int	<code>sizeY</code>

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 mit Hilfe der Funktion **readXML(int)** eingelesen, verarbeitet, also in ein Array umgewandelt und anschließend in der Klasse gespeichert.

Sollte nun ein Sensor nach dem Wert an seiner aktuellen Position fragen, so kann diese Klasse mit Hilfe der Funktionen **generateMessage(const char*)** und **sendSensorResponse(std::string, cGate *)** eine Antwort generieren. Die Position kann dabei mit der Klasse **SimpleCoord** und der Wert an dieser Position mit **SimpleSensorData** repräsentiert und per Nachricht verschickt werden.

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 ein 2-dimensionales Array in der Größe des Playgrounds mit Werten, die die jeweils oben angegebenen Einheiten besitzen. In Listing 4.2.1.2 ist die `generateTemperature()`-Funktion als ein Beispiel aufgeführt. Im Falle dieser

Abbildung 4.5: CustomWorldUtility: Methoden

int**	readXML (int)
void	setTemperature ()
void	setPressure ()
void	setHumidity ()
void	setLight ()
void	initialize (int stage)
void	handleMessage (cMessage *msg)
void	generateEnvironmentData ()
int *	generateTemperature (int size) generates values for the temperature on the playground values will be between 10 and 30 unit: °C More...
int *	generatePressure (int size) generates values for the pressure on the playground values will be between 995 and 1005 unit: hPa More...
int *	generateHumidity (int size) generates values for the humidity on the playground values will be between 70 and 75 unit: % More...
int *	generateLight (int size) generates values for the light intensity on the playground values will be between 10'000 and 100'000 unit: lx More...
ExtendedMessage *	generateMessage (const char *msgname)
void	updateDisplay ()
void	finish ()
void	sendSensorResponse (std::string, cGate *)
void	destroySensorData ()

Funktion werden Zufällige integer-Werte generiert, welche sich im Bereich 10 bis 30 bewegen.

```
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;
}
```

Um eben diese Daten wieder lesen zu können dient die Funktion readXML(). Sie verwendet die cXMLElement-Funktionen aus dem Omnet++-Framework. Diese ist sehr hilfreich beim Verarbeiten von XML-Dateien. Da in dieser Simulation die Umweltparameter in Form von XML gespeichert sind, ist dies eine sehr nützliche Funktionalität.

```
int** CustomWorldUtility::readXML(int fileName)
{
    // get the xml from the parameter, return type cXMLElement
    [...]

    // get a vector (of type cXMLElement) with all childs of
    ↪ the root-tag
    cXMLElementList nListRows = rootE->getChildren();
    int amountRows = nListRows.size();

    int** data = new int*[amountRows];
    for (int i = 0; i < amountRows; i++){
        //this is a row as cXMLElement
        cXMLElement* nListRowArray = nListRows[i];
        //this is a list of an entire row as cXMLElements
        cXMLElementList nLeafElementsList = nListRowArray->
            ↪ getChildren();
        int amountColumns = nLeafElementsList.size();
        data[i] = new int[amountColumns];
        for (int j = 0; j < amountColumns; j++) {
            data[i][j] = atoi(nLeafElementsList[i]->
                ↪ getNodeValue());
        }
    }

    return data;
}
```

Die bis jetzt beschriebenen Funktionen werden alle bereits bei der Initialisierung der Simulation ausgeführt. Während der Laufzeit sind dagegen Andere von Bedeutung. Die zentrale Funktion für die Kommunikation ist hier, wie auch bei anderen Klassen, die `handleMessage(cMessage)`-Funktion.

Diese wartet auf eine Nachricht vom Typ `'GET [Sensortyp]'`. Sollte eine solche Nachricht ankommen, so ist diese vom Typ `ExtendedMessage`, kommt von einem Sensor und enthält einen Parameter, welcher die Position des anfragenden Sensors beinhaltet. Mit diesen Informationen wird die Funktion `sendSensorResponse` aufgerufen.

```
void CustomWorldUtility::sendSensorResponse(string sensorType,
    ↪ cGate* srcGate, SimpleCoord* position)
{
    SimpleSensorData *data;
    if (sensorType == "temperature") {
        data = new SimpleSensorData("data", this->
            ↪ temperatureArray[(int)position->x][(int)position
            ↪ ->y]);
    } else if (sensorType == "pressure") {
        data = new SimpleSensorData("data", this->
            ↪ pressureArray[(int)position->x][(int)position->y
            ↪ ]);
    } else if (sensorType == "humidity") {
        data = new SimpleSensorData("data", this->
            ↪ humidityArray[(int)position->x][(int)position->y
            ↪ ]);
    } else if (sensorType == "light") {
        data = new SimpleSensorData("data", this->lightArray[(
            ↪ int)position->x][(int)position->y]);
    } else {
        throw new exception;
    }

    string messageName = "POST ";
    messageName += sensorType;
    ExtendedMessage *newmsg = generateMessage(messageName.
        ↪ c_str());
    newmsg->getParList().add(data);
    string gateName = srcGate->getBaseName();
    gateName += "$o";
    int index = srcGate->getIndex();
    send(newmsg, gateName.c_str(), index); //"worldDataGate$o"
    numSent++;
}
```

Diese erzeugt eine Antwort auf den GET-Request. Mit Hilfe des Sensortypes und der Position auf dem Playground wird das korrekte Parameterarray und die x- und y-Koordinate bestimmt und der darin stehende Integer-Wert als SimpleSensorData gespeichert. Diese neu erzeugte Instanz wird anschließend an eine Nachricht der Form 'POST [Sensortyp]' angehängt und an den Sender zurück geschickt.

4.2.1.3 MyWirelessNode und Sensor

MyWirelessNode implementiert das StatisticsInterface und erbt von der Klasse Sensor, welche vom cSimpleModule erbt. Die Klasse Sensor soll lediglich mehr Struktur in die Klasse MyWirelessNode bringen und die Funktionen und Attribute in jene unterteilen, welche für die allgemeine Kommunikation und weiteres verantwortlich sind und jene, die nur für die Sensordaten zuständig sind.

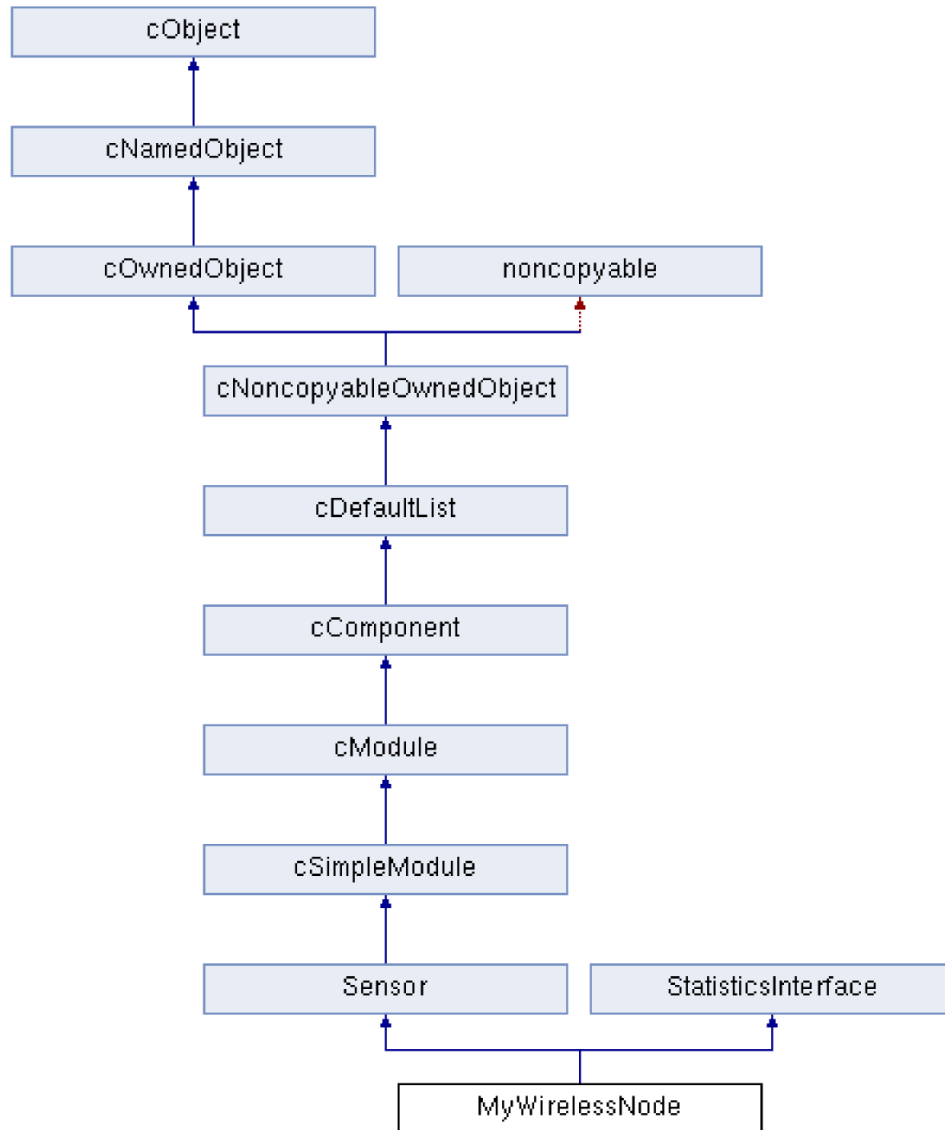
Es ist neben der Klasse für die Umwelt CustomWorldUtility die zweite essentielle Klasse der Simulation. Das Verhalten aller Knoten im Netzwerk wird durch MyWirelessNode definiert. Neben den Standardfunktionen eines Omnet++-Moduls, **initialize()**, **handleMessage()** und **finish()**, können Knoten zum Beispiel ihre Position abfragen.

Dazu dient die Funktion **updatePosition()** 4.2.1.3, welche das Attribut **position** aktualisiert.

```
/**
 * update the position data by a given Coord object
 */
void MyWirelessNode::updatePosition()
{
    Coord* back;
    BasePhyLayer* phy = FindModule<BasePhyLayer*>::
        ↳ findSubModule(this);
    ChannelMobilityPtrType pMobType = phy->getMobilityModule()
        ↳ ;
    if(pMobType != NULL){
        back = new Coord(pMobType->getCurrentPosition());
        delete position;
        position = back;
    }
}
```

Dafür wird die Klasse FindModule aus dem MiXiM-Framework verwendet. Diese enthält eine Funktion findSubModule(cModule), welche zum Finden von service-

Abbildung 4.6: MyWirelessNode: Vererbung



ähnlichen Modulen dient. Die FindModule-Klasse verhält sich für diese Art von Klassen, wie eine Art Servicemanager.

Auf diese Weise hat man Zugriff auf das Mobility-Modul eines bestimmten Sensor-knotens und kann aus diesem die aktuelle Position auf dem Playground abrufen.

Des weiteren beinhaltet das Modul MyWirelessNode Funktionen zum Anfordern von Umweltdaten. Dies erfolgt über einen GET request an das World-Modul, welches anschließend eine Antwort darauf sendet. Dafür stehen die Funktionen requestData() und sendDataRequest(std::string)4.2.1.3 zur Verfügung, wobei sendDataRequest(std::string) innerhalb von requestData() aufgerufen wird.

```
void MyWirelessNode::requestData()
{
    updatePosition();

    //generate messages for data requests
    std::string request = "GET ";
    if (this->type->humidity) {
        this->sendDataRequest(request + this->typenames[0]);
    }
    if (this->type->pressure) {
        this->sendDataRequest(request + this->typenames[1]);
    }
    if (this->type->temperature) {
        this->sendDataRequest(request + this->typenames[2]);
    }
    if (this->type->light) {
        this->sendDataRequest(request + this->typenames[3]);
    }
}

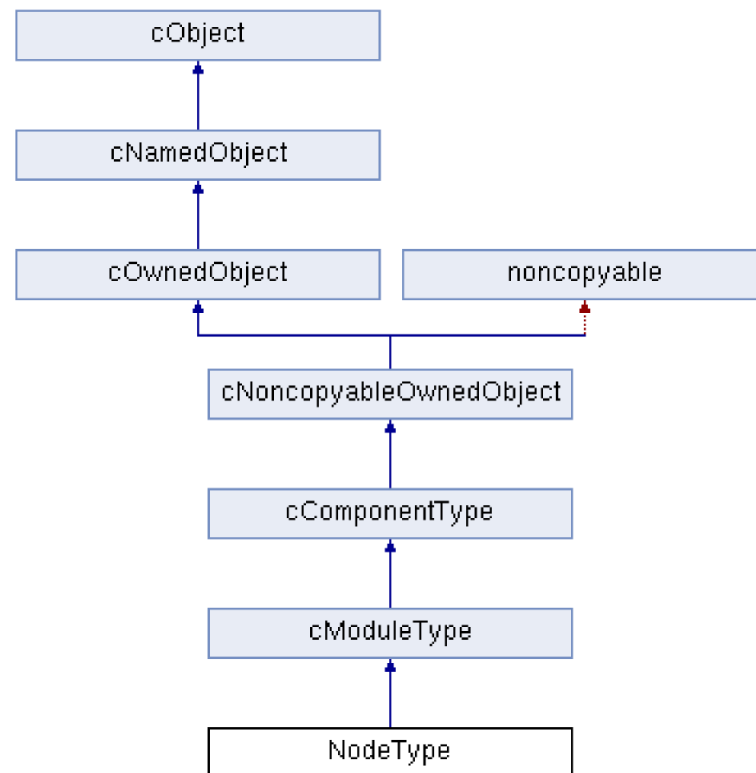
void MyWirelessNode::sendDataRequest(std::string request)
{
    ExtendedMessage *newmsg = generateMessage(request.c_str())
    ↪ ;
    SimpleCoord *coord = new SimpleCoord("pos", this->position
    ↪ );
    newmsg->getParList().add(coord);
    std::stringstream s;
    s << "X: " << coord->x << " Y: " << coord->y;
    ev.bubble(this, s.str().c_str());
    send(newmsg, "toWorld$o");
    numSent++;
}
```

Um aktuelle Daten anfordern zu können wird zunächst immer die aktuell gespeicherte Position des Knoten aktualisiert. Danach wird zu jedem Sensortyp des Knotens eine Nachricht vom Typ 'GET [Sensortyp]' erstellt, mit den aktuellen Koordinaten bestückt und anschließen an die CustomWorldUtility gesendet.

4.2.1.4 NodeType

NodeType erbt von der Klasse cModuleType und erfüllt den gleichen Zweck. Es ist dazu da die MyWirelessNode-Klassen für jeden Knoten anzulegen, sodass diese global bekannt sind, verwendet werden können und am Ende der Simulation wieder gelöscht werden.

Abbildung 4.7: NodeType: Vererbung



4.2.1.5 SensorType

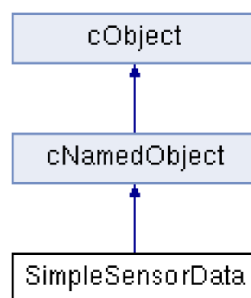
Die Klasse SensorType ist eine Helferklasse für MyWirelessNode. Es besitzt lediglich 4 Attribute, welche definieren ob der jeweilige Knoten einen bestimmten Typ von Sensor besitzt oder nicht.

Abbildung 4.8: SensorType: Member

bool	temperature
bool	pressure
bool	light
bool	humidity

4.2.1.6 Simple*

Abbildung 4.9: SimpleClasses: Member



Es gibt die 3 Klassen SimpleCoord, SimpleSensorData und SimpleSensorType. Sie erben jeweils direkt von cNamedObject und beinhalten nur wenige Attribute. So enthält SimpleCoord beispielsweise 3 Attribute x, y und z, um Koordinaten zu repräsentieren.

cNamedObject und ebenso Kinderklassen davon lassen sich per leicht cMessage übertragen. Alle haben daher den Zweck eine einfache Kommunikation zu ermöglichen. So wird beispielsweise SimpleCoord benutzt, um beim Anfordern von Sensordaten in MyWirelessNode die Koordinaten an die CustomWorldUtility mit zu übertragen, damit in der Response der jeweilige Datensatz der richtigen Position zurückgegeben wird.

Aus diesem Grund ist die einzig benutzte Funktion in diesen Modulen der Konstruktor. Beim Erstellen einer der Klassen werden die nötigen Variablen gesetzt und an die Instanz ein Name vergeben. So lassen sich diese Objekte nach dem Hinzufügen zur Parameterliste einer Klasse auch leicht wieder abrufen (siehe Beispiel 4.2.1.6).

```

//im Sender:
cMessage *msg = new cMessage(msgname);
SimpleCoord *coord = new SimpleCoord("position", this->
    ↪ position);
msg->getParList().add(coord);

//
// Nachricht uebertragen

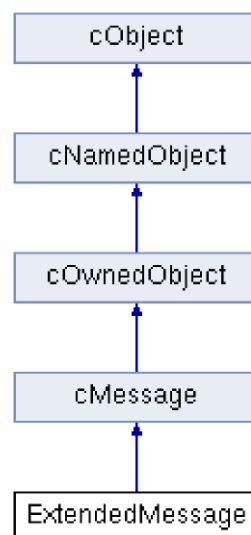
```

```
//  
  
//im Empfaenger:  
SimpleCoord *position = (SimpleCoord*) msg->getParList().  
    ↪ remove("position");
```

4.2.1.7 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.

Abbildung 4.10: ExtendedMessage: Vererbung



Im Listing 4.5 ist zu sehen welche Statistikparameter erhoben werden.

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


4.2.1.8 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
 - CustomLinearMobility
 - CustomWorldUtility
 - HumiditySensor
 - LightSensor
 - PressureSensor
 - Sensor
 - TemperatureSensor
- Compound Module
 - HLNode
 - HLPNode
 - HNode
 - HPNode
 - LNode
 - LPNode
 - MyWirelessNode
 - PNode
 - THLNode
 - THLPNode
 - THNode
 - THPNode
 - TLNode
 - TLPNode
 - TNode
 - TPNode
- Networks
 - MyNetwork
- Messages
 - ExtendedMessage

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.

CustomLinearMobility Das Modul CustomLinearMobility ist eine Erweiterung der LinearMobility, welche MiXiM bereitstellt. Neben dem Verhalten welches in C++ beschrieben ist, ist das Modul lediglich um den Parameter maxSpeed erweitert, welcher in der omnetpp.ini definiert werden kann.

CustomWorldUtility Wie im Codebeispiel (4.1) zu sehen ist, ist das Modul CustomWorldUtility eine Erweiterung des Moduls BaseWorldUtility. Zusätzlich zu den darin definierten Eigenschaften hat es einige weitere Parameter. Zunächst eine bool'sche Variable, welche festlegt, ob zu Beginn der Simulation neue Umweltparameter generiert werden sollen.

Während numSensorNodes definiert, wieviele Sensorknoten im Netzwerk vorhanden sind und somit wieviele Gates zu Sensorknoten gehen, so definiert numGates die Anzahl der einzelnen Sensoren und somit auch die Anzahl der dahin gehenden notwendigen Gates. Da Sensorknoten auch mit mehr als einem Sensor bestückt sein können, ist der Wert beider Variablen mitunter durchaus verschieden.

Die restlichen Variablen speichern die Umweltparameter in Form einer xml-Datei.

Listing 4.1: CustomWorldUtility

```

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

simple CustomWorldUtility extends BaseWorldUtility
{
    parameters:
        bool createData;
        int numGates;
        int numSensorNodes;
        xml xmlTemperature = xmldoc("WorldModel/data/
            ↪ temperature.xml");
        xml xmlPressure = xmldoc("WorldModel/data/pressure.xml
            ↪ ");
        xml xmlHumidity = xmldoc("WorldModel/data/humidity.xml
            ↪ ");
        xml xmlLight = xmldoc("WorldModel/data/light.xml");

        @class("CustomWorldUtility");
    gates:
        inout worldDataGate[numGates];
        inout toNode[numSensorNodes];
}

```

Neben den Parametern besitzt dieses Modul auch noch zusätzlich 2 verschiedene Arten von inout-Gates. Die Menge von worldDataGates verbindet die Welt mit jedem Sensor der auf einem Sensorknoten verbaut ist. Die Menge der toNode-Gates dient für die direkte Verbindung zwischen World und Nodes.

Sensoren Es gibt 5 Module welche die Sensoren definieren. Zum einen den allgemeinen Sensor, der das Vatermodul für alle 4 Sensortypen bildet. Er enthält nicht viel, lediglich Gates für Verbindungen zur Welt, um die Umweltparameter abfragen zu können und zum jeweiligen Knoten, für die Kommunikation innerhalb des Bauteils. Weiterhin besitzt er noch einen Parameter, der im Kindmodul den Typ des Sensors beschreibt.

- HumiditySensor
- PressureSensor
- LightSensor
- TemperatureSensor

Die 4 Sensortypen, welche vom allgemeinen Sensor erben, definieren jeweils nur den Typ des Sensors. Sie können auf den Knoten als Submodule verwendet werden. Je-

der Sensorknoten kann 1 bis 4 Sensoren verschiedener Typen beinhalten und somit ergeben sich 15 verschiedene Arten von Sensorknoten.

Listing 4.2: Sensor

```
package mynetwork.Node.Sensor;

simple Sensor
{
    parameters:
        //type of the sensor
        string type;
        @display("i=block/wrx");
        @class("Sensor");
    gates:
        inout toNode;
        inout worldDataGate;
}
```

4.2.2.2 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.

Alle Sensorknoten, die in dieser Simulation vorkommen sind solche Compound Module.

MyWirelessNode Zum einen existiert das Modul MyWirelessNode, welches das Vatermodul für alle Sensorknoten bildet. Es erbt selbst von einem Modul, welches die kabellose Kommunikation unter den Knoten und Batteriestatistiken ermöglicht. Außerdem beinhalten die Knoten auch Funktionen der Mobility, genauer gesagt der CustomLinearMobility, welche auch mobile Knoten ermöglicht. Es besitzt weiterhin ein inout Gate zur World.

Sensorknoten Es gibt insgesamt 15 verschiedene Arten von Sensorknoten, welche am Anfang von Kapitel 4.2.2 aufgelistet sind. Die Benennung ergibt sich wie folgt (jeweils aus dem Anfangsbuchstaben des englischen Begriffs):

- T - Temperatur

- H - Luftfeuchtigkeit
- L - Licht
- P - Druck

Hierzu ein Codebeispiel des NED-Modules des Knotens, der alle 4 Sensoren enthält:
Listing 4.3.

Listing 4.3: THLPNode

```
package mynetwork.Node;
import mynetwork.Node.Sensor.TemperatureSensor;
import mynetwork.Node.Sensor.HumiditySensor;
import mynetwork.Node.Sensor.LightSensor;
import mynetwork.Node.Sensor.PressureSensor;

module THLPNode extends MyWirelessNode
{
    @display("bgb=210,491");
    gates:
        inout toSensor[4];
        inout worldGate[4];

    submodules:
        TemperatureSensor: TemperatureSensor {
            @display("p=140,380");
        }
        HumiditySensor: HumiditySensor {
            @display("p=140,310");
        }
        LightSensor: LightSensor {
            @display("p=140,450");
        }
        PressureSensor: PressureSensor {
            @display("p=70,450");
        }
    }

    connections:
        toSensor[0] <--> { delay = 0ms; } <-->
            ↳ TemperatureSensor.toNode;
        worldGate[0] <--> { delay = measureTime; } <-->
            ↳ TemperatureSensor.worldDataGate;

        toSensor[1] <--> { delay = 0ms; } <--> HumiditySensor.
            ↳ toNode;
```

```
worldGate[1] <--> { delay = measureTime; } <-->
    ↳ HumiditySensor.worldDataGate;

toSensor[2] <--> { delay = 0ms; } <--> LightSensor.
    ↳ toNode;
worldGate[2] <--> { delay = measureTime; } <-->
    ↳ LightSensor.worldDataGate;

toSensor[3] <--> { delay = 0ms; } <--> PressureSensor.
    ↳ toNode;
worldGate[3] <--> { delay = measureTime; } <-->
    ↳ PressureSensor.worldDataGate;
}
```

Wie man sehen kann erbt der Sensorknoten von `MyWirelessNode`. Er übernimmt zunächst also alle Eigenschaften des vorher bereits erklärten Modules. Hinzu kommen pro Sensor jeweils 1 Gate zum Sensor und zur world, welches den Sensor mit der Welt verbindet.

Entscheidend für das jeweilige Modul ist jedoch die Definition der Submodule. Im Falle des `THLPNode` werden genau 4 Submodule definiert:

- `TemperatureSensor`
- `HumiditySensor`
- `LightSensor`
- `PressureSensor`

Also für jeden Sensor das jeweilige Modul.

Zuletzt werden noch für die Gates die Verbindungen definiert.

4.2.2.3 Networks

Das Netzwerk ist das wichtigste Modul jeder `Omnet++`-Simulation, denn es fügt alle Elemente zusammen. Ein Netzwerk ist den Compound-Modulen daher nicht unähnlich.

MyNetwork Für das Modul `MyNetwork` heißt das, dass es eine Instanz des `CustomWorldUtility` und variabel viele Sensorknoten zusammenfügt, indem es diese als Submodule definiert und deren Gates miteinander verbindet. Die Definition ist dem `MiXiM`-Modul `BaseNetwork` ähnlich, aber hat deutlich mehr zusätzliche Submodule

und Parameter.

So sind die Definitionen des Playgrounds, der Mobilität und des ConnectionManager analog, die Submodule der World und des Sensorknoten und die dazu benötigten Parameter, die zum Beispiel die Anzahl der jeweiligen Sensorknoten definieren und die Gateverbindungen jedoch kommen nur hier vor.

Listing 4.4: Network

```
network MyNetwork
{
    parameters:
        bool createNewEnvironmentData;

        //...

        int numTNodes = default(0);
        //...
        int numTHLPNodes = default(0);

        // total number of hosts in the network
        int numNodes = numTNodes + numPNodes + numHNodes +
            ↪ numLNodes + numTHNodes + numTLNodes + numTPNodes
            ↪ + numHLNodes + numHPNodes + numLPNodes +
            ↪ numTHLNodes + numTLPNodes + numTHPNodes +
            ↪ numHLPNodes + numTHLPNodes;
        string wuType = default("CustomWorldUtility");

        //BaseNetwork
        double playgroundSizeX @unit(m); // x size of the area
            ↪ the nodes are in (in meters)
        double playgroundSizeY @unit(m); // y size of the area
            ↪ the nodes are in (in meters)
        double playgroundSizeZ @unit(m); // z size of the area
            ↪ the nodes are in (in meters)
        **.mobility.constraintAreaMinX = default(0m);
        **.mobility.constraintAreaMinY = default(0m);
        **.mobility.constraintAreaMinZ = default(0m);
        **.mobility.constraintAreaMaxX = default(
            ↪ playgroundSizeX);
        **.mobility.constraintAreaMaxY = default(
            ↪ playgroundSizeY);
        **.mobility.constraintAreaMaxZ = default(
            ↪ playgroundSizeZ);
```

```
string cmType = default("org.mixim.base.  
    ↳ connectionManager.ConnectionManager"); //  
    ↳ connection manager to use  
  
@display("bgi=maps/germany,bgb,$playgroundSizeX,  
    ↳ $playgroundSizeY,white;bgp=0,0;bgb=  
    ↳ $playgroundSizeX,$playgroundSizeY");  
  
submodules:  
    world: CustomWorldUtility {  
        //for every sensor (NOT only every node) there is  
        ↳ one gate needed  
        numGates = numNodes  
        + numTHNodes + numTLNodes + numTPNodes +  
        ↳ numHLNodes + numHPNodes + numLPNodes  
        + 2*numTHLNodes + 2*numTLPNodes + 2*numTHPNodes +  
        ↳ 2*numHLPNodes  
        + 3*numTHLPNodes;  
        numSensorNodes = numNodes;  
        createData = createNewEnvironmentData;  
    }  
  
    tnode[numTNodes]: TNode {  
        parameters:  
            @display("i=device/card");  
            numHosts = numNodes;  
    }  
    //...  
    thlpnode[numTHLPNodes]: THLPNode {  
        parameters:  
            @display("i=device/card");  
            numHosts = numNodes;  
    }  
  
    connectionManager: <cmType> like IConnectionManager {  
        parameters:  
            @display("i=abstract/multicast;is=s");  
    }  
  
connections allowunconnected:  
  
    world.worldDataGate++  
    <--> { delay = 100ms; } <-->
```



```

    tnode[i].worldGate[0] for i=0..(numTNodes-1);
    //...
    world.worldDataGate++
    <--> { delay = 100ms; } <-->
    thlpnode[i].worldGate[3] for i=0..((numTHLPNodes-1));

    world.toNode++
    <--> { delay = 0ms; } <-->
    tnode[i].toWorld for i=0..(numTNodes-1);
    //...
    world.toNode++
    <--> { delay = 0ms; } <-->
    thlpnode[i].toWorld for i=0..(numTHLPNodes-1);
}

```

Nicht auf den ersten Blick zu erkennen ist die Kommunikation zwischen den Knoten selbst. Die connections sind durch den Term `allowunconnected` so definiert, dass auch dynamisch Verbindungen aufgebaut werden können, die nicht strikt im Compound Module definiert sind. Mit Hilfe der `sendDirect()`-Funktion können so dennoch Nachrichten zwischen Gates versendet werden.

4.2.2.4 Messages

Nachrichten sind das essentielle 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.

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

Listing 4.5: `ExtendedMessage`

```

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

```

4.2.2.5 omnetpp.ini

Die omnetpp.ini ist eine Datei, in der Eigenschaften der Simulation geändert werden können, ohne das 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.

```
#####  
#                               Custom Parameters                               #  
#####  
  
#informations  
##set maxspeed to 0mps for stationary nodes or to any higher  
    ↳ value for mobility nodes  
**.mobility.maxSpeed = 0mps  
***.mobility.maxSpeed = 10mps  
  
#this boolean value defines if new data for the sensor will be  
    ↳ created  
MyNetwork.createNewEnvironmentData = ask  
  
#set the amount of nodes here  
MyNetwork.howManyHumidityAndTemperatureNodes = ask  
[...]  
  
#set the number of nodes for the simulation here  
#MyNetwork.numNodes = 1  
MyNetwork.networkPosX = 0  
MyNetwork.networkPosY = 0  
MyNetwork.networkSensorAlgorithm = 1  
MyNetwork.networkValue = -1  
**.netwl.headerLength = 24bit  
  
***.mobilityType = "CustomLinearMobility" -> NED default  
#Gaussian distribution for the acceleration with a mean of 1  
    ↳ leads to a result if ~half of the nodes will end  
    ↳ stationary  
**.mobility.speed = 1mps  
**.mobility.acceleration = normal(1, 1)  
**.mobility.angle = default  
  
#Playground  
**.playgroundSizeX = 400m
```

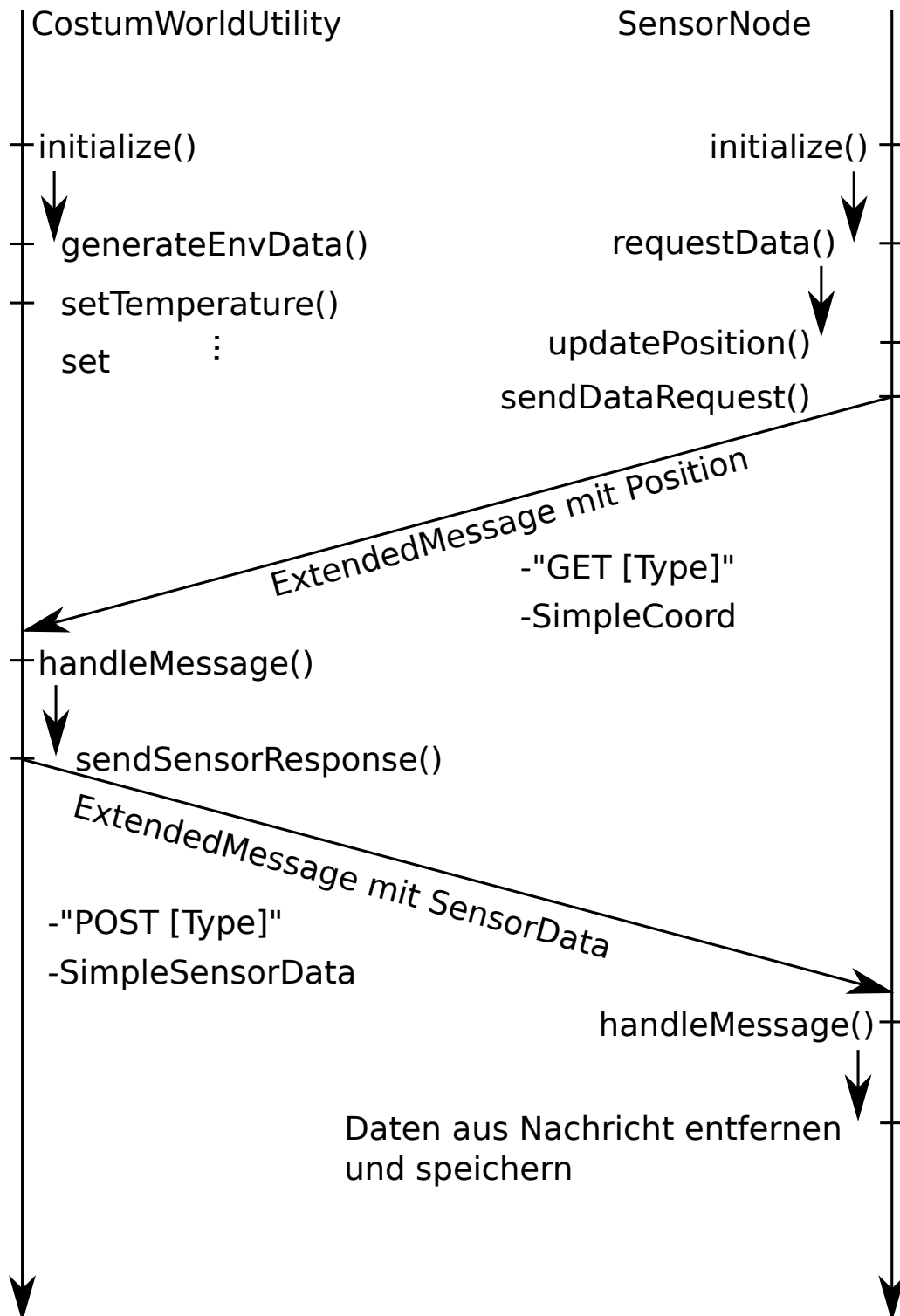
```
**.playgroundSizeY = 550m
```

Für die relevanten Parameter wurde in der ini-Datei eine spezielle Sektion eingeteilt. Hier können beispielsweise Werte für die playgroundSize vergeben oder festgelegt werden ob Mobilität der Knoten erlaubt sein soll.

4.3 Funktionsweise

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 ihr 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.

Abbildung 4.11: Verlauf der Nachrichten und Methodenaufrufe



5 Zusammenfassung

wurde das ziel der BA erreicht? alles umgesetzt was verlangt war?

zusammenfassu
schreiben

Literatur- und Webverzeichnis

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

Sensornetz; zuletzt besucht am 5. August 2014.

- [15] DANIEL BÜLTMANN, MACIEJ MÜHLEISEN, SEBASTIAN MAX (CHAIR OF COMMUNICATION NETWORKS RWTH AACHEN UNIVERSITY): *openWNS*. Online unter <http://openwns.org>; zuletzt besucht am 18. November 2014.
- [16] GEORGE F. RILEY, THOMAS R. HENDERSON: *The ns-3 Network Simulator*. Online unter <http://www.nsnam.org/>; zuletzt besucht am 18. November 2014.
- [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 18. August 2014.
- [18] JÖRG SOMMER, 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 18. November 2014.

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 9. Dezember 2014

Thomas Rückert