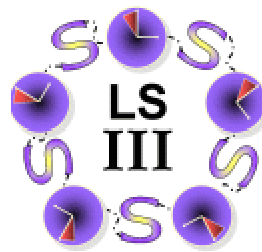
Endbericht PG 518

Verteiltes Energiemanagement/- Analyse elektrischer Ressourcen unter realen Betriebsbedingungen - Real-Dezent

Autoren:
Chen Dagang, Ramzi Dghim, Viktoria Glasmachers, Marius Helf, Hüseyin Kagba, Shinejil Khurel-Odon, Li Ma, Khayyam Mahmood, Benjamin Pickhardt, Stefan Pöter, Sebastian Ruthe, Han Yu



Projektgruppe
am Fachbereich Informatik
der Universität Dortmund

WS 2007/2008, SS 2008

Betreuer:
Prof. Dr. Horst F. Wedde
Prof. Dr.-Ing. C. Rehtanz
Dipl.-Inform. Sebastian Lehnhoff
Dipl.-Ing. Olav Krause

Inhaltsverzeichnis

Inhaltsverzeichnis 1

Abbildungsverzeichnis 5

Tabellenverzeichnis 7

1 Einleitung 9
- 1.1 Einleitung . . . 9

2 Hardware 13
- 2.1 Einleitung . . . 13
- 2.2 Übersicht . . . 13
  - 2.2.1 Generator Steuern mittels SPS . . . 14
  - 2.2.2 SPS-PC-Kommunikation über Modbus . . . 15
  - 2.2.3 Konzept des Netzmodells . . . 15
- 2.3 Laborumgebung . . . 16
  - 2.3.1 Schalttafel . . . 18
  - 2.3.2 Synchrongeneratoren . . . 19
  - 2.3.3 Asynchronmaschine . . . 23
- 2.4 MODBUS TCP/IP Steuerung über UPnP . . . 24
  - 2.4.1 MODBUS . . . 24
  - 2.4.2 TCP/IP . . . 25
  - 2.4.3 MODBUS TCP/IP über UPnP . . . 28
- 2.5 Das SPS Subscription Protocol (SSP) . . . 31
  - 2.5.1 Einleitung . . . 31
  - 2.5.2 Details . . . 32
  - 2.5.3 Maschinen-spezifische Spezifikationen . . . 33
  - 2.5.4 Einschränkungen . . . 34
- 2.6 Realisierung des SSP auf einer SPS . . . 34
  - 2.6.1 Einleitung . . . 34
  - 2.6.2 Die Bibliothek Ethernet.lib . . . 35
  - 2.6.3 Kommunikation zwischen einer SPS und einem TCP Client . . . 37
  - 2.6.4 Der Notify-Mechanismus des SSP-Protokolls . . . 38
  - 2.6.5 Die “Input loop” . . . 40
  - 2.6.6 Die Taskkonfiguration . . . 43
- 2.7 Benutzerschnittstellen der Laborumgebung . . . 43

2.7.1 SPS Control GUI . . . 45
2.7.2 (A)Synchron Machine Control GUI . . . 46
2.7.3 Machine Value GUI . . . 48
2.7.4 Abriss über das Verhalten elektrischer Maschinen . . . 52
2.8 Generatorsteuerung . . . 53
2.8.1 Machine Control Protokoll . . . 53
2.8.2 MCP Fehlercodes . . . 57
2.8.3 SPS - Implementierung der Steuerung mit MCP . . . 58
2.8.4 Implementierung in Java . . . 63
2.9 Netzregelung . . . 65
2.9.1 Konzept . . . 65
2.9.2 Netzregelung . . . 65
2.9.3 Spannungsregelung . . . 65
2.9.4 Momentenregelung . . . 66
2.10 Digitales Auslesen des SINEAX Multimeters . . . 67
2.10.1 Pinbelegung . . . 67
2.11 Photovoltaik-Anlage . . . 73
2.11.1 Einleitung . . . 73
2.11.2 Wechselrichter . . . 74
2.11.3 Gleichstromaggregate . . . 74
2.11.4 Matlab Simulation . . . 75
2.11.5 Auswertung . . . 80
2.12 Photovoltaik-Simulation mit Java . . . 80
2.12.1 Allgemeine Beschreibung der Photovoltaik-Simulation . . . 80
2.12.2 Die Werte der Leistung aus der Dateien lesen . . . 80
2.12.3 Photovoltaik simulieren mit Gleichstrom-Aggregat-Controller (GSAController) 81
2.13 SPS-Implementierung Photovoltaik . . . 83
2.13.1 Nachbildung einer Photovoltaikanlage . . . 83
2.13.2 Steuerung . . . 84
2.14 Consumer . . . 84
2.14.1 Einleitung . . . 84
2.14.2 Elektronische Lasten . . . 85
2.14.3 Funktionsweise der Consumer . . . 86
2.15 Consumer-Simulation mit elektronischen Lasten . . . 87
2.15.1 Allgemeine Beschreibung der Consumer-Simulation . . . 87
2.15.2 Beschreibung der Java-Klasse: „ConsumerController“ . . . 88
2.15.3 Werte schreiben auf die SPS-Register mit „valuesToSPS“ . . . 89
2.16 SPS-Implementierung Consumer . . . 91

3 Hardware-Simulation 95
3.1 Theoretische Grundlagen . . . 96
3.1.1 Lastflussberechnung . . . 96
3.1.2 State Estimation . . . 104
3.1.3 Elektrisches Netz . . . 110
3.2 Netzdarstellung und Implementierung . . . 114
3.2.1 Edge . . . 114
3.2.2 Node . . . 116

3.2.3 SynchroGenerator . . . 117
3.2.4 AsynchroGenerator . . . 118
3.2.5 Inverter . . . 119
3.2.6 Consumer . . . 120
3.2.7 PowerNetDatastructure . . . 121
3.2.8 AbstractBalancer . . . 122
3.3 Bilanzierungsaufgaben . . . 125
3.3.1 Bilanzierungen in der Hardwarsimulation . . . 125
3.3.2 Implementierung der Lastflussestimation in Java . . . 130
3.4 UPnP . . . 132
3.4.1 Komponenten . . . 133
3.4.2 Protokolle . . . 134
3.5 UPnP Ablauf . . . 135
3.6 Eventing in UPnP-Umgebung . . . 135
3.7 UPnP Implementierungen . . . 138
3.7.1 Implementierung Device, Controlpoint . . . 138
3.7.2 UPnP im Einsatz im Rahmen des REAL-DEZENT Projektes . . . 142
3.7.3 Device Generierung . . . 143
3.7.4 Die Annotation Description Factory und GetterSetterDescriptionFactory . . . 144
3.8 UPnP Leistungstest . . . 144
3.8.1 Testumgebung . . . 145
3.8.2 Tests . . . 145
3.8.3 Ergebnisse . . . 146
3.9 Hardwaresimulation . . . 146
3.9.1 Simulation mit der ContainerGUI . . . 146
3.9.2 ContainerGUI . . . 147
3.9.3 Generatorverhalten für die Simulation . . . 151

4 Experimente **163**
4.1 Ziele . . . 163
4.2 Beschreibung . . . 163
4.2.1 Umgebung . . . 163
4.2.2 Steuerung der Maschinen und Messwertaufnahme . . . 164
4.2.3 Simulationsdaten . . . 164
4.3 Ergebnisse . . . 164
4.3.1 Versuch A: geregelt . . . 164
4.3.2 Versuch B: ungeregelt . . . 167
4.3.3 Vergleich . . . 170

5 Fazit und Ausblick **171**

6 Literaturverzeichnis **173**

Abbildungsverzeichnis

1.1 PG Organisation 11

2.1 Konzeption 14
2.2 SPS-PC 15
2.3 Darstellung des Netzmodells 16
2.4 Schalttafel 18
2.5 Synchrongenerator 19
2.6 Generator Schema 20
2.7 Polling basierte MODBUS Kommunikation 29
2.8 Eventbasierte Kommunikation mittels SSP n 30
2.9 Graphische Darstellung des Bausteins Ethernet_Client_Open 35
2.10 Graphische Darstellung des Bausteins Ethernet_Client_Close 36
2.11 Graphische Darstellung des Bausteins Ethernet_Write_Pt 36
2.12 Zustandsdiagramm des SSP_sendMessage Blocks 38
2.13 Grober Ablauf des Notify-Mechanismus’ 39
2.14 Abhandlung einer SSP-Nachricht 42
2.15 Klassendiagramm 45
2.16 SPS Control GUI 46
2.17 Synchron Machine Control GUI 47
2.18 Asynchron Machine Control GUI 48
2.19 Machine Value GUI 49
2.20 Robust Machine Value GUI 51
2.21 Aufbau einer Synchronmaschine, aus [19] 52
2.22 Zustandsdiagramm des Machine Control Protocol 60
2.23 Klassendiagramm MCP 64
2.24 RS232-Kabel 68
2.25 Gleichstromaggregate der PVA 75
2.26 Quellcode der Simulationsfunktion 76
2.27 Quellcode der Simulationsfunktion 77
2.28 Auswertung für Juli 2007 79
2.29 *Nachbildung einer Photovoltaikanlage* 83
2.30 Consumer Frontansicht 85
2.31 Messgeräte für Consumer 87

3.1 Beispiel für die Anwendung eines Newton-Raphson Verfahren 97
3.2 Graphischer Ablauf des Newton-Raphson Verfahren 98

3.3 Kein Konvergenzverhalten beim Newton-Raphson Verfahren 99
3.4 die Synchronmaschine 111
3.5 die Asynchronmaschine 112
3.6 die Klasse Edge 115
3.7 die Klasse Node 117
3.8 die Klasse SynchroGenerator 118
3.9 die Klasse AsynchroGenerator 119
3.10 die Klasse Inverter 120
3.11 die Klasse Consumer 121
3.12 die Klasse PowerNetDatastructure 122
3.13 die Klasse AbstractBalancer 123
3.14 die Klasse LFBalancer 124
3.15 die Klasse WLBalancer 125
3.16 Beispiel Wirkleistungsbilanzierung für drei verschiedene Systemzustände 128
3.17 Beispiel Blindleistungsbilanzierung für drei verschiedene Systemzustände 129
3.18 Klassendiagramm REAL-DEZENT 142
3.19 Sinuskurven für 50 Devices 145
3.20 Sinuskurve eines Devices 145
3.21 Ausschnitt 146
3.22 Klassendiagram von ContainerGUI 147
3.23 Erzeugen des neuen Knoten 148
3.24 die Knoten bei Simulation 149
3.25 die Kanten bei Simulation 150
3.26 Klassendiagramm des ConstantGenerator 152
3.27 Klassendiagramm RandomGenerator 154
3.28 Klassendiagramm LocalRandomGenerater 156
3.29 Klassendiagramm SinusGenerator 158
3.30 Klassendiagram WindGenerator 160
3.31 Klassendiagramm RealBehaviourGenerator 162

4.1 Asynchronmaschine (Windkraftanlage) geregelt 165
4.2 Gleichstromaggregat (Solaranlage) geregelt 165
4.3 Verbraucher 1 geregelt 166
4.4 Synchronmaschine 1 geregelt 166
4.5 Synchronmaschine 2 geregelt 167
4.6 Leistungsbilanz geregelt 168
4.7 Leistungsbilanz ungeregelt 169

Tabellenverzeichnis

2.1 Parametrierung des Synchrongenerator 1 21
2.2 Parametrierung des Synchrongenerator 2 21
2.3 Parametrierung des Asynchrongenerators 24
2.4 Datentypen 26
2.5 MODBUS-Funktioncodes 27
2.6 Aufbau einer SSP-Nachricht 32
2.7 Aufbau von `SSPValues` für `ValueCode0_SensorData` 33
2.8 Aufbau von `SSPValues` für `ValueCode1_State` 33
2.9 Werte für `MachineState` 34
2.10 Parameter des Bausteins SSP_sendMessage 37
2.11 Aufbau einer SSP Nachricht vom PC zur SPS 40
2.12 MCP Kommandos 54
2.13 Machine Control Protocol - Fehlercodes 57
2.14 Generatorzustände der Generatorsteuerung 58
2.15 Mapping Generatorzustand MCP ↔ SPS 59

3.1 Beispielverlauf des Newton-Raphson Verfahren 98
3.2 Gegebene und Gesuchte Werte der einzelnen Knotenarten 100
3.3 Beispielwerte für eine Lastflussberechnung 101
3.4 Errechnete Werte für eine Lastflussberechnung 104
3.5 Beispiel zur Wirkleistungsbilanzierung 127
3.6 Beispiel zur Blindleistungsbilanzierung 128

Kapitel 1

Einleitung

1.1 Einleitung

Durch die Verknappung und damit verbundene Preiserhöhung fossiler Rohstoffe, sowie durch die aktuelle umweltpolitische Stimmung, wird der Druck auf die Energieversorgung größer, die bisherige Primärenergiebasis (Erdöl, Erdgas, Kohle, etc.) um saubere und regenerative Energiequellen zu erweitern, bzw. bisherige Energiequellen effizienter zu nutzen. Charakteristisch für diese 'neue' Art von Energiequellen ist, dass sie verglichen mit ihren fossilen Konkurrenten nur bei großräumiger Nutzung nennenswerte Leistungen erreichen. Um einen wichtigen Beitrag zur aktuellen Versorgung leisten zu können, werden regenerative Energieumwandlungsanlagen (REAs) als Anlagen niedriger Anschlussleistungen in großer Zahl und weiträumig verteilt betrieben (Windparks, Solaranlagen auf den Hausdächern von Siedlungen, etc.). Die Verfügbarkeit dieser erneuerbaren Energiequellen lässt sich jedoch nicht präzise vorhersagen. Dieses unvorhersehbare Verhalten sorgt in gegenwärtigen Energieversorgungsmodellen damit für große Probleme: Die Leistungen regenerativer Anlagen variieren innerhalb der bisherigen Betriebsintervalle von 15-20 min Länge bzw. regional noch beträchtlich. Die vorgehaltene Reserveleistung und eingesetzte Regelenergie wird klassisch aus fossilen, (derzeit noch) hoch verfügbaren und damit planbaren Rohstoffen gewonnen. Die Regelenergie laufen im Standby, bis ihre Leistung eingesetzt wird. Mit vermehrter Einspeisung aus solchen REAs steigt somit die für den worst-case vorzuhaltende Reserveleistung rapide an. Ein derartiges Versorgungsmodell widerspricht nicht nur ganz offensichtlich der Forderung an eine effizientere Nutzung knapper werdender klassischer Energieträger, es zerstört vielmehr zugleich die Vorteile, die durch die Nutzung regenerativer Energieträger erreicht werden sollen.

Ansatz: Dezentrales Energiemanagement auf der Basis eines Realzeit Multiagentensystems

Im Rahmen des von der Deutschen Forschungsgemeinschaft geförderten F&E Projektes DEZENT, einem Gemeinschaftsprojekt zwischen dem Fachbereich Informatik und der Fakultät Elektrotechnik an der Universität Dortmund, wird eine neues Versorgungsmodell für elektrischen Strom aus verteilter, regenerativer Energieerzeugung entwickelt. In unserem Modell stellen verteilte REA selbst Reserveleistung zur Verfügung es wird aber durch lokale Koordination der Bedarf an vorzuhaltender Reserveleistung reduziert. So wird der, für eine gesicherte Versorgung notwendigen, Netzausgleich geschaffen. Um mit der praktischen Unvorhersehbarkeit in den individuellen Anschlussleistungen umgehen zu können, nutzt DEZENT Betriebsintervalle von nur 0,5s Länge. Software Agenten ver-

handeln innerhalb dieser Intervalle stellvertretend für individuelle Erzeuger und Verbraucher autonom und direkt miteinander und stimmen sich kontinuierlich über Leistungsfähigkeit und Bedarf ab. Ein derartiges Vorgehen stellt vollständig neuartige Anforderung an das elektrische Netz, was sowohl die Kommunikationsfähigkeit an die einzelnen Agenten als auch die Analyse der Betriebszustände des Versorgungssystems betrifft. Herkömmliche Analyseverfahren, wie Lastflussrechnungen und Zustandsabschätzungen stoßen bereits im klassischen Modell und wesentlich größeren Betriebsintervallen an ihre Grenzen. Um Störungen bzw. Unfälle im Betriebsablauf (Leitungsüberlastungen, Stromausfälle, Notabschaltungen etc.) zu vermeiden ist eine On-The-Fly Analyse des Versorgungsnetzes unverzichtbar. Im Gegensatz zu hergebrachten Regelungskonzepten, bei denen die Energie von oben nach unten (Top-Down) verteilt und global analysiert wird, um ein gewisses Maß an unvorhersehbaren Störungen in der Versorgungsleistung systemtechnisch aufzufangen, geschieht in DEZENT Verteilung und Abgleich der Energie von unten nach oben (Bottom-Up), ebenso weit verteilt wird auch die in Realzeit mitlaufende Analyse. Dieses System ist von vornherein hochskalierbar und inkrementell angelegt. Die bisherigen Ansätze und Untersuchungen Ansätze im Rahmen des DEZENT Projektes sollen nun im praktischen Einsatz mit realistischen Betriebsmittelkonfigurationen auf in einer verteilten Netzsimulationsanlage implementiert und getestet werden. Ein besonderer Schwerpunkt ist dabei die Realisierbarkeit der Netzanalyse und Zustandsbewertung in den vorgesehenen Zeitintervallen im Millisekundenbereich.

Im nachfolgenden Endbericht werden die Ergebnisse der Projektgruppe des Projektgruppenjahr vorgestellt. Der Endbericht ist in drei große Kapitel unterteilt.

- Hardwaregruppe
- Hardware-Simulationsgruppe
- Experimente

Im ersten ersten Kapitel stellt die Hardware-Gruppe ihre Ergebnisse vor. Die Hardware-Gruppe befasst sich grösten Teils um die SPS-Programmierung und die Ansteuerung der Generatoren. Weiterhin werden weitere Energienquellen wie eine Photovoltaikanlage ins Netz eingebunden sowie aktive Verbraucher.

Die Hardware-Gruppe hat die Laborumgebung genutzt und war für die Steuerung sämtlicher Geräte zuständig. Die Hardware-Gruppe bildet die unterste Ebene im Real-Dezent -System. Es musste zuerst eine Kommunikationsstruktur aufgebaut werden. Dazu wurde u.a. das Protokoll SSP entwickelt. Nach dem Ansteuern der einzelnen Geräte musste eine gemeinsame Koordination stattfinden lassen. Das Ziel bestand darin, alle Geräte gemeinsam zu steuern und koordinieren zu können. Zu den Geräten gehörten sowohl Synchrongeneratoren, Asynchrongeneratoren, sowie Verbraucher und eine Photovoltaik-Anlage. Die größte Herausforderung bestand darin, die Generatoren anzusteuern und zu regeln.

Im zweiten Kapitel werden die Arbeiten der Hardwaresimulationsgruppe dokumentiert. Dabei standen Lastflussberechnung und Simulation des Netzes im Blickpunkt. Die Kontrolle des elektrischen Energieübertragungsnetzwerks sowie die Leistungsverteilung stand dabei im Mittelpunkt. Da die Hardwaresimulation auf die Hardwaregruppe angewiesen war wurde zu Testzwecken eine eigene Simulationsumgebung entwickelt, so dass die Verhaltensmuster der Komponenten im Netzwerk abstrakt nachgebildet wurden und die entwickelten Verfahren getestet wurden, ohne die reale Hardware zu belasten.

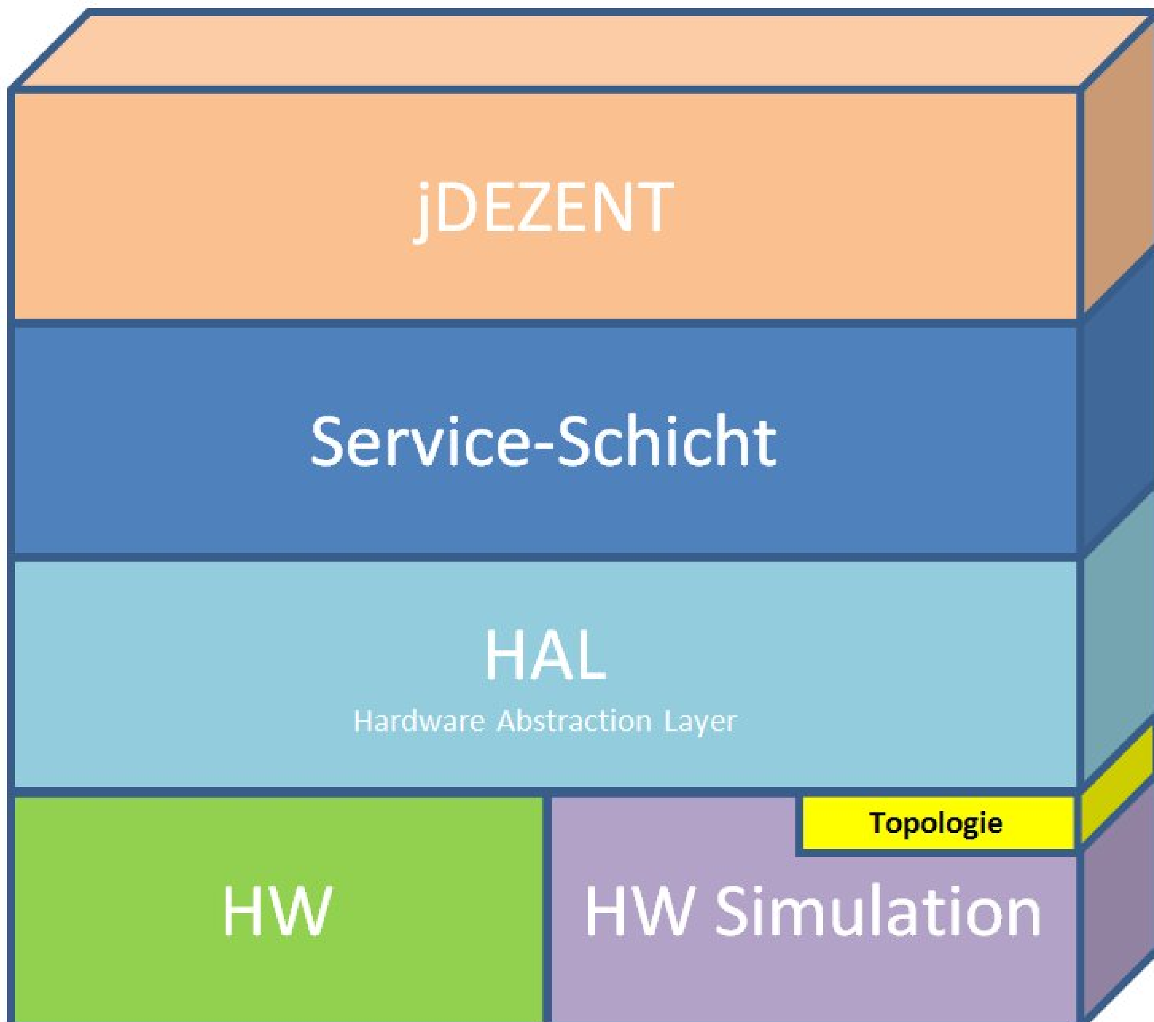


Abbildung 1.1: PG Organisation

Die Schnittstelle zwischen den beiden Gruppen bildet am Ende UPnP .

Im dritten Kapitel werden die Experimente vorgestellt. Es werden der Versuchsaufbau sowie die Experimente dokumentiert und intepretiert. Es wurden verschiedene Szenarien für die Experimente durchgeführt und anschließend analysiert. Zudem gibt es am Ende der Experimente ein Fazit über die geleistete Projektarbeit sowie einen Ausblick für zukünftige Aufgaben.

Eine Unterteilung des Projektes in zwei Teilgruppen war notwendig. Die erste Teilgruppe war für die Schnittstelle zu der Hardware, die uns in der Laborumgebung zu Verfügung stand, zuständig. Ein reibungsloses Ansteuern alle Maschinen war hier notwendig. Die andere Teilgruppe hatte Aufgaben zwischen der Hardware und dem Real-Dezent Algorithmus zu bewältigen. Sie stellte die Schnittstelle zum Real-Dezent Algorithmus dar und saß zwischen der Hardware und dem Real-Dezent . So stand bei dieser Gruppe die Netzsimulation und Netzanalyse im Vordergrund um einen reibungslosen Ablauf zwischen der Hardware und dem Netz zu erwirken. Zudem wurde im späteren Verlauf über UPnP

auf die Hardware zugegriffen, um ein Ansteuern aus der höheren Schicht zu ermöglichen. Durch die verschiedenen Aufgabenbereiche war es möglich, dass beide Gruppe unabhängig und parallel von einander arbeiten konnten. Kurzfristige Änderungen zwischen den einzelnen Schichten konnten somit rechtzeitig aufeinander abgestimmt werden und hinsichtlich der endgültigen Schnittstellen korrigiert werden.

Kapitel 2

Hardware

2.1 Einleitung

Der Projektgruppe steht eine Netzsimulationsanlage zur Verfügung, die u.a. über Synchron- und Asynchronmaschinen sowie steuerbare Gleichstromaggregate verfügt (s. 2.3 auf S. 16). Um auf dieser Anlage den Dezent-Algorithmus zu implementieren und zu testen, muss es möglich sein, die verfügbaren Geräte zu steuern und zu regeln.

Die Steuerung aller Geräte erfolgt über speicherprogrammierbare Steuereinheiten (SPS), die ihrerseits von einem PC gesteuert werden.

Die folgenden Kapitel beschäftigen sich mit der Entwicklung eines Kommunikationsprotokolls zwischen PCs und SPS sowie der Implementierung einer Steuerung für die Synchronmaschinen.

2.2 Übersicht

Auf der Hardwareseite stellen sich die folgenden Herausforderungen:

1. Die Generatoren müssen über SPS (Speicherprogrammierbare Steuerungen) gesteuert werden.
2. Die Kommunikation zwischen PC und SPS muss über Modbus geschehen, damit die Generatoren mittels SPS von einem Leitcomputer [1] gesteuert und beobachtet werden können.
3. Wichtige Werte der Generatoren müssen aufgenommen und die Wertänderungen dokumentiert werden, als Vorbereitung für spätere Visualisierungen und Automatisierungen.

Das Grundkonzept unseres Plans ist dargestellt in Abbildung 2.1.

[1] Ein Leitcomputer ist hier ein Computer, auf dem sämtliche Steuermechanismen implemeniert sind und der die Vermittlung zwischen SPS und PCs übernimmt.

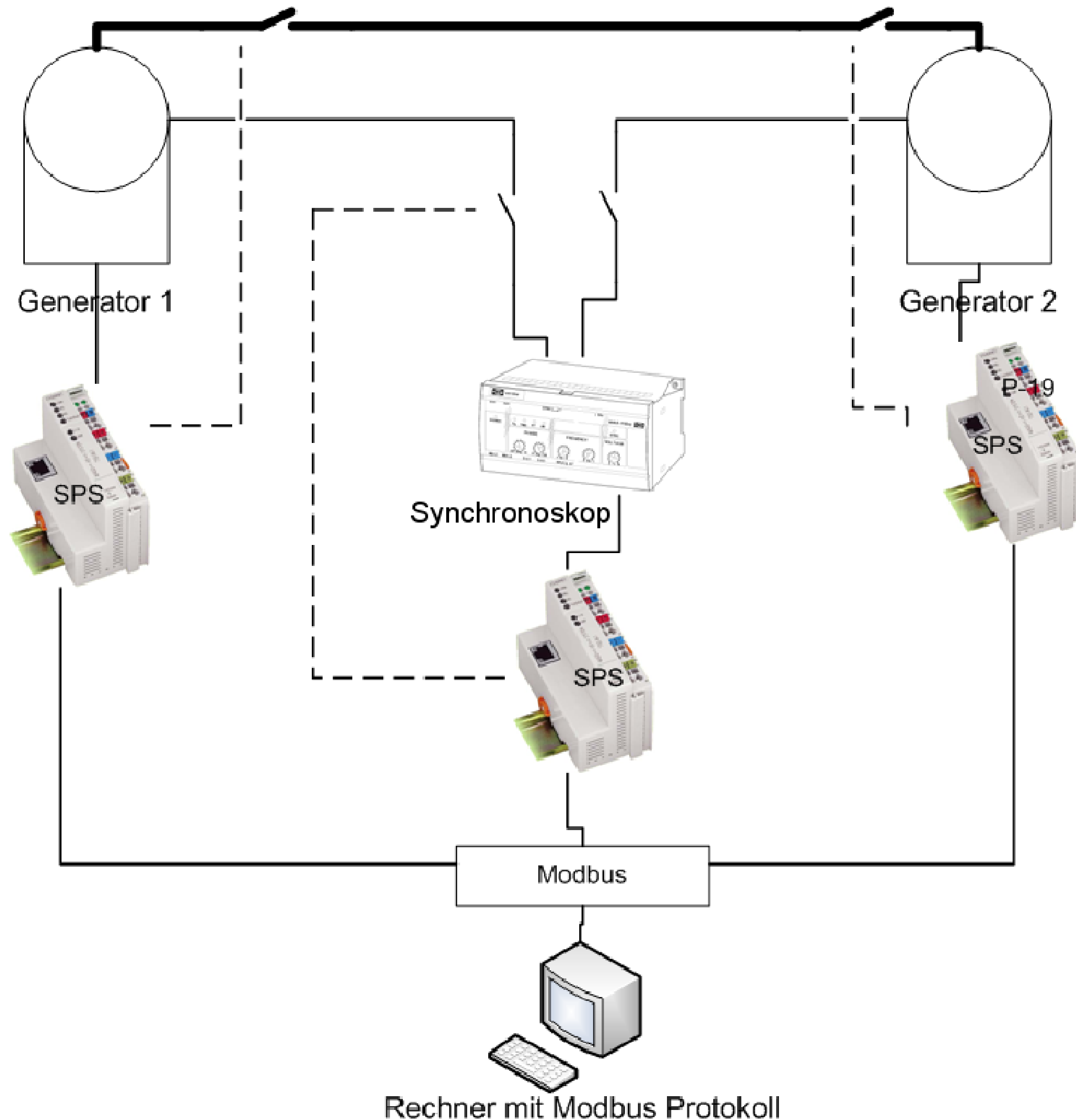


Abbildung 2.1: Konzeption

2.2.1 Generator Steuern mittels SPS

Die Steuerung der Generatoren erfordert jeweils eine SPS pro Geberator, die die Kommandos an den eigenen Generator gibt, Ergebnisse misst und ihn regelt, so dass der Generator das entsprechende Verhalten zeigt. Gewünschte Verhalten sind zum Beispiel Anfahren, Runterfahren, Not-Aus, oder das Synchronisieren mit den Anderen SPS/Generatoren. Man kann jeder Zeit entsprechende Programmparameter einer SPS ändern. Die Änderung wird dann im nächsten Regelungszylus [2] berücksichtigt.

[2]Ein Regelungszyklus folgt dabei einem deterministischem Eingabe-Verarbeitung-Ausgabe-Prinzip.

2.2.2 SPS-PC-Kommunikation über Modbus

Die Generatoren werden jeweils von einer SPS gesteuert, die dann wiederum durch einen Computer gesteuert werden. Dies geschieht, um grundlegende Funktionen garantieren zu können und für eine schnellere Reglung. Um die SPS zu Steuern und zu Überwachen, ist eine Kommunikationen zwischen SPS und Computer erforderlich. Es gibt zwei Wege, wie man SPS und Computer verbinden kann. Man kann entweder durch eine serielle Schnittstelle die Verbindung zwischen einer SPS und einem Computer aufbauen oder über Ethernet. Hier wurde Ethernet als Kommunikationsmedium ausgewählt, damit sich der Computer nicht unmittelbar in der Nähe der SPS befinden muss und damit mehere SPS leicht von einem einzelnen Computer angesprochen werden können. Außerdem wurde Modbus als Kommunikationsprotokoll eingesetzt, wobei die Modbus-Packete über TCP/IP versendet werden.

Der ganze Kommunikationsvorgang verhält sich wie folgt: Der Leitrechner meldet sich bei einem SPS [3] an, sobald der Rechner eine Rückmeldung von der SPS bekommt, kann er die Register der SPS lesen und schreiben. Außerdem kann der Rechner auch direkt von den Eingängen der SPS lesen oder auf Ausgänge schreiben. Die SPS bemerkt Veränderung an ihren Eingängen und nimmt die neuen Werte auf. Die SPS kann entsprechend ihrer Programmierung auf Veränderungen reagieren. Weiterhin kann der Client-Rechner die veränderten Werte aus den Registern der SPS auslesen. Eine grobe Darstellung der Beziehung von SPS-PC: ist in Abbildung 2.2 gegeben.

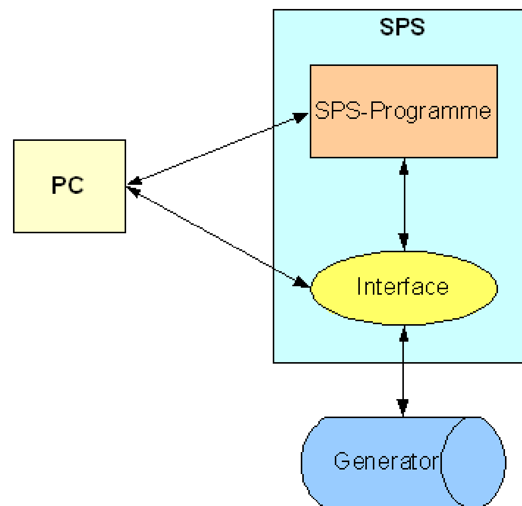


Abbildung 2.2: SPS-PC

2.2.3 Konzept des Netzmodells

Abbildung 2.1 stellt das Konzept des Netzmodells dar und zeigt, wie Generatoren, gesteuert über SPS, vernetzt werden können.

[3] Das SPS besitzt eine eigene IP-Adresse.

Jeder Generator wird von einer SPS gesteuert. Die Messwerte vom Generator können bei Bedarf an das Synchronisierrelais geliefert werden und dann mit den Werten einer anderen Synchronmaschine oder den Werten des Netzes verglichen werden. Sehe die Abbildung 2.3.

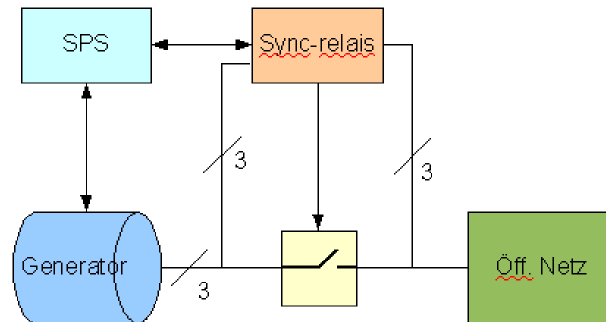


Abbildung 2.3: Darstellung des Netzmodells

Der Regelungs-Prozess funktioniert so: Die Synchronmaschine wird zuerst angefahren und möchte an das Netz angeschlossen werden. Aber bevor die Synchronmaschinen mit dem Netz verkoppelt werden, müssen zuerst die Synchronmaschinen mit dem Netz synchronisieret werden. Deshalb berichtet die zugehörige SPS über Ethernet dem Steuercomputer, dass die Maschine schon angefahren ist und jetzt mit dem Netz synchronisiert werden möchte. Der Steuercomputer erhält diese Anforderung und schickt dann eine Nachricht zur SPS des Synchronisierrelais, dann schaltet dieser SPS einen Schalter zwischen der Synchronmaschine und dem Synchronisierrelais und einen Schalter zwischen dem Netz und dem Synchronisierrelais ein. Das Synchronisierrelais liefert die Werte der Spannungsabweichung, Frequenzabweichung und Phasenabweichung zwischen der Sysnchronmaschine und dem Netz. Der Steuercomputer liest die Werte aus der SPS des Synchronisierrelais und analysiert diese, wenn Abweichungen existieren, dann weiß der Computer, dass er die Frequenzen oder die Phase der beiden Spannungen in Übereinstimmung bringen muss, durch Änderungen an der Drehzahl der Synchronmaschine. Um dies zu erreichen befiehlt der SPS der Synchronmaschine, dass sie die Drehzahl um den von der Synchronisierrelais-SPS vorgeschlagenen Betrag ändern soll. Anschließend bekommt der Computer die neuen Werte vom Synchronisierrelais. Der Vorgang wird wiederholt, bis die jeweiligen Frequenzen, Phasen und Spannungen, von Netz und Synchronmaschine, durch Änderung des Erregerstroms der Synchronmaschine, angeglichen worden sind. Ist eine Angleichung erreicht, so kann die Synchronmaschine auf das Netz, durch die Synchronmaschinen-SPS, aufgeschaltet werden.

2.3 Laborumgebung

Die Laborumgebung der Fakultät für Elektrotechnik an der Universität Dortmund umfasst mehrere Hardware- Komponenten, mit Hilfe derer ein Stromnetz sowie mehrere Verbraucher und Erzeuger "real" simuliert werden können. Die wichtigsten Komponenten sind:

- **Sineax Messumwandler**
  Der Sineax Messumwandler DME 442 ist ein programmierbarer Multi-Messumformer zur gleich-

zeitigen Erfassung mehrerer Größen in Starkstromnetzen. Die Analogausgänge des Messumformers können als Strom- bzw. Spannungsausgänge programmiert werden.

- **Speicherprogrammierbare Steuereinheit (SPS)**

- **Synchronoskop**

- **Simoreg Antriebssteuerung**
  Die Simoreg Antriebssteuerung ist ein programmierbarer Baustein, der vor dem eigentlichen Antrieb (Elektromotor) sitzt und diesen steuert. Über Analogeingänge können der Antriebssteuerung Drehzahl- bzw. Momentensollwerte vorgegeben werden, die die interne Regelung der Steuerung dann anfährt.

- **Wechselrichter**
  Die Wechselrichter dienen in dem späteren Versuchsaufbau dazu, den Gleichstrom, der durch die Gleichstromaggregate erzeugt wird, in Wechselstrom umzuwandeln, damit dieser für das öffentliche Netz nutzbar gemacht werden kann.

- **Gleichstromaggregate**
  Das Gleichstromaggregat kann durch die Vorgabe eines/einer Gleichstroms/-spannung Wirkleistung erzeugen. In dem späteren Versuchsaufbau wird es dazu benutzt, das Verhalten einer Photovoltaik-Anlage zu simulieren.

- **Synchrongenerator**
  Der Synchrongenerator erzeugt durch die Drehung eines Elektromagneten in einem äußeren Magnetfeld eine Wirk- und Blindleistung. Der genaue Aufbau sowie die Funktionsweise kann im Kapitel 2.7.4 auf S. 52 nachgelesen werden.

- **Asynchrongenerator** Der Asynchrongenerator erzeugt Wirkleistung in Abhängigkeit von der Drehzahldifferenz zwischen Rotor- und Statormagnetfeld. Der genaue Aufbau sowie die Funktionsweise kann im Kapitel 2.7.4 auf S. 53 nachgelesen werden.

- **Gleichstrommotor**

Mit Hilfe dieser Komponenten soll ein Test des DEZENT-Algorithmus im Inselnetzbetrieb sowie im Netzparallelbetrieb möglich sein. Dabei kann im Netzparallelbetrieb die Testumgebung um weitere virtuelle Erzeuger bzw. Verbraucher ergänzt werden. Virtuell bedeutet in diesem Zusammenhang, dass die Erzeuger und Verbraucher nur auf der Software-Seite vorhanden sind und über den Verhandlungsalgorithmus Energie anbieten oder anfordern können. Dadurch entsteht in der Laborumgebung ein Energiedefizit bzw. -überschuss, der durch das öffentliche Netz aufgenommen werden kann.
Im Inselnetzbetrieb werden die real vorhandenen Erzeuger und Verbraucher separat, vom öffentlichen Netz getrennt, betrieben. Dies bedeutet insbesonders, dass sich für einen stabilen Betrieb die jeweilige erzeugte und verbrauchte Energie ausgleichen muss.
In den folgenden Abschnitten werden die einzelnen Hardwarekomponenten sowie deren Verbindungen untereinander genauer beschrieben. Dabei wird an geeigneter Stelle auf die Besonderheiten der einzelnen Komponenten in den beiden Betriebsmodi eingegangen.

2.3.1 Schalttafel

Der Aufbau des Netzmodells ist so gestaltet, dass die Stromausgänge der beiden Synchronkraftwerke, des Asynchronkraftwerkes und des starren Netzes zentral an einer Schalttafel zusammenlaufen. Dies ermöglicht es sehr flexibel Szenarien mit unterschiedlichen Netztopologien zu erstellen und zusammen zu schalten. Durch Setzen der jeweiligen Verbindungskabel können somit wahlweise beide Synchronkraftwerke aufeinander, das Asynchronkraftwerk auf das starre Netz oder beliebig andere Kombinationen geschaltet werden. Auch bietet die Laborumgebung die Möglichkeit Effekte, die insbesondere bei sehr langen Stromleitungen auftreten, durch eine Leitungsnachbildung zu simulieren. Die beiden "Enden" dieser simulierten "Leitungen" liegen auch auf der Schalttafel auf, so dass sie Problemlos zwischen zwei Komponenten geschaltet werden kann.

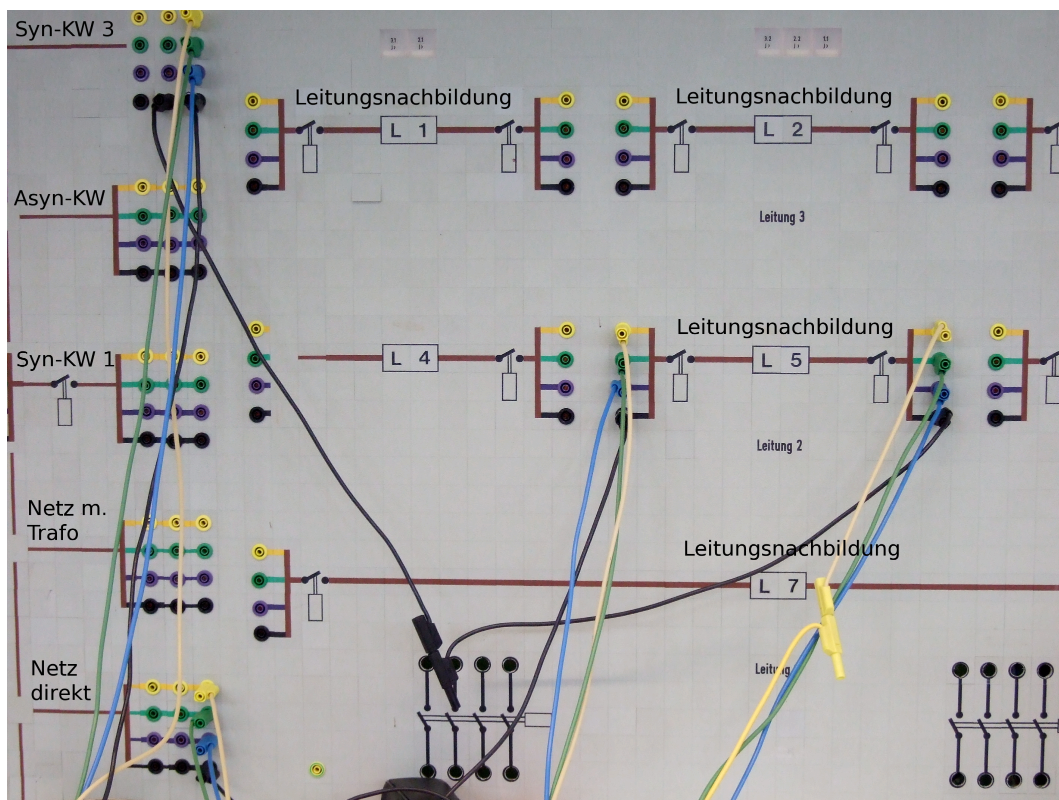


Abbildung 2.4: Schalttafel

Im Inselnetzbetrieb werden nur die zwei Synchronkraftwerke zusammen mit dem Asynchronkraftwerk und den Verbrauchern sowie den Wechselrichtern zusammengeschaltet. Im Netzparallelbetrieb wird zusätzlich das starre Netz hinzugeschaltet. Bei den Synchrongeneratoren und dem starren Netz muss vor dem Zuschalten noch ein Synchronisierprozess ausgeführt werden, der dafür sorgt, dass sowohl die Spannung, die Frequenz als auch der Phasenwinkel der jeweiligen Netze übereinstimmt.

2.3.2 Synchrongeneratoren

Die Synchrongeneratoren werden jeweils indirekt durch eine SPS gesteuert. In der Laborumgebung wird jede Synchronmaschine durch einen Elektromotor angetrieben, der durch einen Simoreg Regler gesteuert wird. Dem Regler kann über einen Analogeingang eine Spannung vorgegeben werden, die der Regler je nach Betriebsmodus als einen Drehzahl- oder Momentensollwert interpretiert. Die Analogeingänge des Reglers sind direkt mit der weiter oben erwähnten SPS verbunden. Die Zuordnung zwischen vorgegebener Spannung und einzustellender Drehzahl wird weiter unten unter dem Stichwort Umrechnungsgrößen behandelt.

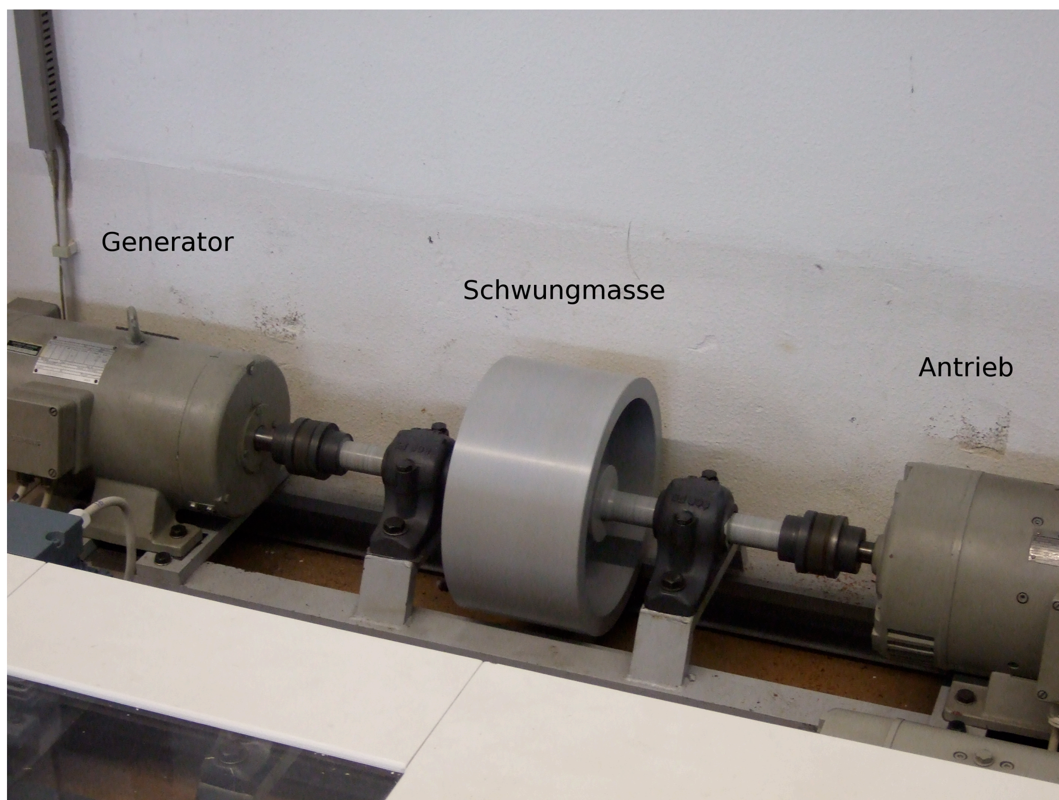


Abbildung 2.5: Synchrongenerator

Neben der Drehzahl und dem Moment kann auch der Erregerstrom direkt von der Generator-SPS gesetzt werden.
Zur Synchronisation auf ein anderes Netz ist an jedem Kraftwerk ein Synchronisierrelais vorhanden, das das Spannungs-/ Frequenzdelta der beiden Netze ermittelt und über einen Analogeingang an die SPS übergibt. Zusätzlich bekommt die SPS ein Zuschaltsignal vom Synchronoskop, sobald das Spannungs-/ Frequenzdelta sowie der Phasenwinkel klein genug sind. Die SPS kann somit ihren internen Zustand auf "Zugeschaltet" setzen, das Relais, welches die Maschine aufs Netz koppelt, wird vom Synchronoskop automatisch geschaltet.
Für die Kommunikation des Generators über Universal Plug & Play (UPnP) mit dem Real-Dezent System wird die Generator-SPS mit einem lokalen Rechner verbunden. Die Kommunikation läuft dabei vom Rechner zur SPS über das Modbus Protokoll, das von der SPS hardwaremäßig unterstützt

wird. Die SPS kommuniziert über das SSP-Protokoll mit dem Rechner, der empfangene Messwerte mittels UPnP an das Real-Dezent System übergibt. Das UPnP Protokoll wird ausführlich in Kapitel beschrieben.
Die folgende Abbildung zeigt den Aufbau des Synchrongenerators sowie die angesprochenen Sekundärgeräte:

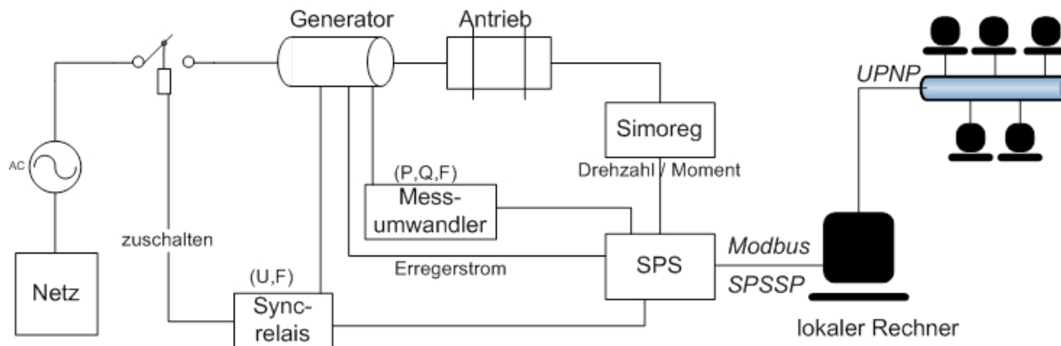


Abbildung 2.6: Generator Schema

Parametrierung der Synchrongeneratoren

In diesem Abschnitt sollen die Umrechnungsgrößen der im vorherigen Abschnitt spezifizierten Schnittstellen der Generator-SPS zu den direkten Hardwarekomponenten beschrieben werden. Bei den Umrechnungen handelt es sich in der Regel um lineare Funktionen, die einerseits von dem Messbereich der Messgeräte und Hardwarekomponenten auf den Strom-/Spannungsbereich der Analogausgänge der Messgeräte und andererseits vom Strom-/Spannungsbereich der Analogeingänge der Generator-SPS auf den Wertebereich der Analogeingänge schließen muss.
Ein Beispiel: Das Messgerät hat einen Messbereich von $3000V$ bis $-3000V$ und bildet diesen Bereich auf seinen Analogausgang auf den Spannungsbereich $5V$ bis $-5V$ ab. Dieser Analogausgang des Messgeräts ist mit dem Analogeingang der SPS verbunden, der einen Spannungsbereich von $10V$ bis $-10V$ auf einen Wertebereich von 32768 bis -32768 abbildet. Damit auf der SPS der Wert 4231 des Analogeingangs (im weiteren Verlauf als RAW-Wert bezeichnet) auf den ursprünglichen Messwert zurückgerechnet werden kann, muss zunächst die 4231 auf die gemessene Spannung am Analogeingang der SPS zurückgerechnet werden: $\frac{4231}{32768} \cdot 10.0V = 1.291V$. Der errechnete Spannungswert am Analogeingang der SPS wurde vom Analogausgang des Messgerätes erzeugt und muss nun auf den damit verbunden Messwert $\frac{1.291}{5} \cdot 3000 = 774.6$ projiziert werden. In diesem Beispiel wäre die vollständige Umrechnungsfunktion von einem in der SPS vorhanden RAW-Wert auf den am Messgerät anliegenden Messwert:

$$MESS_{Wert} = \frac{RAW_{Wert}}{32768} \cdot 10 \cdot \frac{3000}{5} + 0.$$

Für die folgende Parametertabelle kann die obige Formel in der allgemeinen Form

$$MESS_{Wert} = \frac{RAW_{Wert}}{RAW_{Max}} \cdot IN_{Max} \cdot \frac{MESS_{Max}}{OUT_{Max}} + MESS_{offset}$$

Messwert Bez	RAW_{Max}	IN_{Max}	$MESS_{Max}$	OUT_{Max}	$MESS_{offset}$
Blindleistung (Q)	32768	10	2000.0	10	0
Wirkleistung (P)	32768	10	1739.0	10	0
Spannung (U)	32768	10	250.0	10	0
Drehzahl	32768	10	1610	10	0
Moment (M)	32768	10	100	10	0
Erreger (I)	32768	10	2.0	10	0
Frequenz (F)	32768	10	2.0	10	50

Tabelle 2.1: Parametrierung des Synchrongenerator 1

Messwert Bez	RAW_{Max}	IN_{Max}	$MESS_{Max}$	OUT_{Max}	$MESS_{offset}$
Blindleistung (Q)	32768	10	3464.0	10	0
Wirkleistung (P)	32768	10	3464.0	10	0
Spannung (U)	32768	10	250.0	10	0
Drehzahl	32768	10	1610	10	0
Moment (M)	32768	10	100	10	0
Erreger (I)	32768	10	2.2	10	0
Frequenz (F)	32768	10	2.0	10	50

Tabelle 2.2: Parametrierung des Synchrongenerator 2

zugrunde gelegt werden:

Für den umgekehrten Fall, dass die SPS über einen ihrer Analogausgänge an den Analogeingang einer Hardwarekomponente wie zum Beispiel des Simoreg einen Sollwert kommunizieren will, muss die obige Formel anstatt nach $MESS_{Wert}$ nach RAW_{Wert} aufgelöst werden:

$$RAW_{Wert} = \frac{(MESS_{Wert} - MESS_{offset}) \cdot RAW_{Max}}{IN_{Max}} \cdot \frac{OUT_{Max}}{MESS_{Max}}$$

In diesem Fall entspricht die Variable $MESS_{Wert}$ dem vorzugebenden Sollwert und der berechnete Wert RAW_{Wert} dem Wert, der auf der SPS an den Analogeingang geschrieben werden muss, damit der entsprechende Sollwert an der Hardwarekomponente gesetzt wird.

Steuerungskonfiguration der Generator SPS

Die Steuerungskonfiguration und die Verdrahtung der Module der Generator SPS'en ist sowohl für die Synchronmaschine als auch für die Asynchronmaschine (bis auf die Erregerstromeinrichtung) dieselbe. Dies hat vor allem den Vorteil, dass auf allen drei Generator-SPS'en das gleiche SPS-Programm laufen kann. Die Verwendung von baugleichen Modulen in der gleichen Reihenfolge führt dazu, dass hinter den in der Steuerungskonfiguration definierten Variablen und Speicherbereichen die gleichen Module bzw. die gleichen Ausgänge der jeweiligen Module stehen.
Für die Asynchronmaschine bedeutet dies allerdings, dass Stellgrößen wie der Erregerstromsollwert definiert werden, die real nicht vorhanden sind. Aus Zeitgründen hat sich die Projektgruppe aber für diesen Entwurf entschieden, auch weil ansonsten zwei verschiedene Versionen des eigentlich gleichen SPS-Programms gepflegt werden müssten.

Es folgt zunächst die Auflistung aller Module des “K-Bus[FIX]”-Teil der Steuerungskonfiguration der SPS Programme für die Generatoren:

1. **0750-0530 8 DO 24 V DC 0.5A** *(8 × 24 V Digital Ausgang)*:

Ausgangsnr.	Variablen Name
0	SimoregFreigabe
1	SimoregEin
2	ErregerFreigabe
3	SimoregMomentRegelung
4	ErregerEin
5	SimoregStatik
6	keine Belegung
7	keine Belegung

2. **0750-0430 8 DI 24 V DC 3.0ms** *(8 × 24 V Digital Eingang)*:

Ausgangsnr.	Variablen Name
0	NotAusGlobal
1	AntriebBereit
2	AntriebStoerung
3	UxN_Unterschreitung
4	UxN_Ueberschreitung
5	GenSchalterGeschlossen
6	Synchronisieren
7	keine Belegung

3. **0750-0550 2 AO 0-10 V DC** *(2 × 0-10 V Analog Ausgang)*:

Ausgangsnr.	Variablen Name
0	SimoregDrehzahlSollwert
1	SimoregMomentSollwert

4. **0750-0550 2 AO 0-10 V DC** *(2 × 0-10 V Analog Ausgang)*:

Ausgangsnr.	Variablen Name
0	ErregungSollwert
1	keine Belegung

5. **0750-0456 2 AI 0-10 V S.E.** *(2 × 0-10 V Analog Eingang)*:

Ausgangsnr.	Variablen Name
0	DrehZahlAnalog
1	IErreger

6. **0750-0457 2 AI -10-10 V S.E.** *(2 × -10-10 V Analog Eingang)*:

Ausgangsnr.	Variablen Name
0	P
1	Q

7. **0750-0457 2 AI -10-10 V S.E.** *(2 × -10-10 V Analog Eingang)*:

Ausgangsnr.	Variablen Name
0	P_Antrieb
1	keine Belegung

8. **0750-0457 2 AI -10-10 V S.E.** *(2 × -10-10 V Analog Eingang)*:

Ausgangsnr.	Variablen Name
0	`Delta_F_Sync`
1	`Delta_U_Sync`

9. **0750-0457 2 AI -10-10 V S.E.** *(2 × -10-10 V Analog Eingang)*:

Ausgangsnr.	Variablen Name
0	`F`
1	`U`

10. **0750-0509 2 DO 230 V AC 300 mA SolidState** *(2 × 230 V Digital Ausgang)*:

Ausgangsnr.	Variablen Name
0	`SimoregVersorgung`
1	`ErregerVersorgung`

11. **0750-0509 2 DO 230 V AC 300 mA SolidState** *(2 × 230 V Digital Ausgang)*:

Ausgangsnr.	Variablen Name
0	`ZuschaltenAufNetzmodell`
1	`keine Belegung`

12. **0750-0650/0000-0012 RS 232 C Interface** *(Serielle Schnittstelle RS 232)*:

Ausgangsnr.	Variablen Name
0	`rs232_ByteGesendet`
1	`rs232_ByteWartet`
2	`rs232_initAck`
3	`keine Belegung`
4	`readByte1`
5	`readByte2`
6	`readByte3`
7	`rs232_inByte1`
8	`rs232_inByte2`
9	`rs232_inByte3`
10	`rs232_req`
11	`rs232_ack`
12	`rs232_init`
13	`sendByte1`
14	`sendByte2`
15	`sendByte3`
16	`rs232_OutByte1`
17	`rs232_OutByte2`
18	`rs232_OutByte3`

2.3.3 Asynchronmaschine

Der Aufbau Asynchronmaschine ist ähnlich wie bei den beiden Synchronmaschinen. Der wesentliche Unterschied ist, dass der Rotor über eine kurgeschlossene Dreiphasenwicklung verfügt. Anstatt durch einen Erregerstrom baut sich das Magnetfeld des Rotors auf, wenn die Drehzahl des Rotors anders

ist als die des Drehstromnetzes im Stator. Für eine detailliertere Beschreibung sei an dieser Stelle auf das Kapitel 2.7.4 auf S. 53 verwiesen. Durch die unterschiedliche Bauart ergeben sich einige Besonderheiten. Die Asynchronmaschine muss zur Synchronisation auf ein anderes Netz nur grob auf die Netzfrequenz gebracht werden, bevor sie zugeschaltet werden kann. Das bedeutet insbesondere, dass das Synchronisierrelais bei der Asynchronmaschine überflüssig wird. Da das Magnetfeld des Rotors durch einen Kurzschlussstrom entsteht, entfällt bei der Asynchronmaschine der Erregerstrom. Eine weitere Konsequenz dieser Bauart ist, dass bei der Asynchronmaschine nur auf die Wirkleistung Einfluss genommen werden kann. Die von der Asynchronmaschine erzeugte Blindleistung kann nur durch zusätzliche Schaltungselemente beeinflusst werden. Die Verdrahtung der Asynchrongenerator-SPS mit den primären Steuerungsbausteinen (Simoreg, etc.) wurde eins zu eins von der Verdrahtung der Synchronmaschinen übernommen. Dabei werden die bei der Asynchronmaschine überflüssigen Ein- und Ausgänge, wie zum Beispiel der Ausgang für den Erregerstrom, einfach ignoriert. In der folgenden Tabelle sind alle Parameter der Umrechnungsgrößen, die für die Asynchronmaschine sinnvoll sind aufgelistet:

Messwert Bez	RAW_{Max}	IN_{Max}	$MESS_{Max}$	OUT_{Max}	$MESS_{offset}$
Blindleistung (Q)	32768	10	3464.0	10	0
Wirkleistung (P)	32768	10	3464.0	10	0
Spannung (U)	32768	10	250.0	10	0
Drehzahl	32768	10	1610	10	0
Moment (M)	32768	10	100	10	0
Frequenz (F)	32768	10	2.0	10	50

Tabelle 2.3: Parametrierung des Asynchrongenerators

2.4 MODBUS TCP/IP Steuerung über UPnP

Die PG benutzt das MODBUS TCP/IP-Protokoll zur Kommunikation mit der obrigen Peripherie. In den ersten Ansätzen sollte die Kommunikation direkt über MODBUS TCP/IP erfolgen, welche hinterher in UPnP eingekapselt werden sollte. Dabei geschieht die Abfrage durch Polling und ist nicht Event-basiert. Die nächsten Verbesserungen haben diesen Nachteil aufgegriffen und die Polling orientierte Kommunikation durch Event-basierte Kommunikation ersetzt, wodurch die Einkapselung des MODBUS TCP/IP-Protokolls in den UPnP Standard verworfen wurde.
Im Folgenden werden die für die Umsetzung benötigten Protokolle und deren Einsatz innerhalb der PG erläutert.

2.4.1 MODBUS

Das MODBUS-Protokoll wurde im Jahre 1979 von Modicon für industrielle Automatisierungssysteme entwickelt und ist mittlerweile ein Industriestandard zum Übertragen von Digitalen/Analogen E/A auf der einen Seite und zum Erfassen von Daten zwischen Kontrolleinheiten und Ausgabegeräten auf der anderen Seite. Die Kommunikation basiert auf dem Master-/Slave- bzw Client-/Server-Prinzip, wobei nur der Master bzw. der Client eine Anforderung übermitteln kann, wöhrend der Slave die

Anfrage erfüllt und dem Master antwortet. Ein Slave kann jede mögliche Hardware sein, die Informationen verarbeitet und ihre Ausgabe anschließend mit Hilfe des MODBUS-Protokolls an den Master sendet.
Der Unterschied zwischen Master und Slave liegt darin, dass der Master den Slave ansprechen kann, den er möchte. Der Slave hingegen kann nur den Mastern antworten, und zwar denen, von welchen dieser angesprochen wurde, d.h. die Slaves können keine eigene Anfrage schicken. Zudem kann der Master mit Hilfe eines Broadcast-Aufrufs alle Slaves ansprechen. Slaves, die von mehreren Mastern angesprochen wurden, können nicht mit Hilfe eines Broadcasts antworten, sie können aber allen, von denen sie angesprochen wurden, antworten.

Kommunikation

Die Anfrage eines Masters enthält

- die Slave oder Broadcast Adresse
- einen Funktionscode, der die Aktion definiert
- die benötigten Daten
- ein Error Checking Feld

Die Antwort des Slaves enthält

- die Bestätigung, dass die Aktion ausgeführt wurde
- die Rückgabedaten
- ein Error Checking Feld

Falls es zu einem Fehler kommt, z.B. bei der Master-Anfrage oder der Slave-Antwort, dann wird eine entsprechende Exception ausgegeben. Das Error-Checking-Feld der Slave-Nachricht dient der Gültigkeitsprüfung. Dieses Feld erlaubt es dem Master den Inhalt der Nachricht zu prüfen. Festzuhalten ist, dass MODBUS ein Anwendungsprotokoll ist, das definiert wie Daten zu intepretieren und organisieren sind.Es ist dabei unabhängig von der darunter liegenden Hardware.

2.4.2 TCP/IP

Das MODBUS-Protokoll wurde anfänglich zur seriellen Kommunikation verwendet. Die Kombination von MODBUS und TCP/IP ist erforderlich, um die Übertragung in einem Netzwerk zu realisieren. TCP/IP ermöglicht im Grunde genommen das blockweise Übertragen von Daten, die zwischen 2 Computern in binärer Form liegen. TCP sichert, dass der Dateninhalt korrekt Übertragen wird, wobei das IP sichert, dass die Daten auch am richtigen Computer ankommen.
MODBUS TCP/IP kombiniert einen Netzwerkübertragungsstandard und ein Anwendungsprotokoll. Die MODBUS-Kommunikation wird also in ein TCP-Wrapper eingegliedert. Dabei fallen beim MODBUS-Protokoll die Adresse und das Error-Checking-Feld weg, da diese Informationen im TCP-Frame eingegliedert sind. Eine MODBUS TCP/IP Application Data Unit hat einen 7 Byte großen Header. Die Bytes sind folgendermaßen verteilt:

- Transaction/Invocation Identifier, der zur Paarung von Nachrichten dient. Wenn mehrere Nachrichten eines Masters über die selbe TCP/IP-Verbindung versendet werden, braucht der Master/Client nicht auf eine Antwort des Slaves/Servers warten. Für dieses Feld werden 2 Bytes verwendet.
- Der Protocol Identifier benutzt 2 Bytes. Der Wert ist hier für das MODBUS-Protokoll auf Null beschränkt.
- Length werden 2 Bytes zur Verfügung gestellt. Es ist ein Byte-Zähler der angibt, welche Felder übrig sind und beinhaltet den Unit Identifier Byte, Function Code Byte und die Datenfelder.
- Unit Identifier Byte dient zur Identifizierung eines entfernten Servers für non TCP/IP Netzwerke. Das Feld ist meistens auf 00 oder FF gesetzt und eine protokollspezifische Einheit, die den Funktionscode und die Daten enthält.

Die ganze MODBUS TCP/IP Application Data Unit ist in das Datenfeld des TCP-Frames eingegliedert und wird über Port 502 versendet. Dieser Port ist für MODBUS reserviert. Das MODBUS-Protokoll kennt im Kern nur 4 Grunddatentypen:

Datentype	Länge	Description
Discrete Inputs	1 Bit	Digitale Eingänge
Coils	1 Bit	Digitale Ausgänge
Input Register	16 Bit	Analoge-Eingangsdaten
Holding Register	16 Bit	Analoge-Ausgangsdaten

Tabelle 2.4: Datentypen

Die meisten MODBUS-Register eines Gerätes sind so organisiert, dass die vier Grunddatentypen wie folgt adressiert werden können:

Adresse	Beschreibung
0xxxx	Lesen/Schreiben von Digitalen Ausgängen oder Ausgangsdaten
1xxxx	Lesen von Digitalen Eingängen
3xxxx	Lesen von Eingabe Registern
4xxxx	Lesen/Schreiben von Ausgängen oder Halte Registern

Dabei sind für jeden Grunddatentyp ein oder mehrere Functioncodes definiert. Die Functioncodes legen die Aufgabe des Slaves fest, z.B. Lesen/Schreiben von Eingangs-/Ausgangsbits, Lesen/Schreiben von Eingangs-/Ausgangs-Registern.

Funktions - code	**Funktions - name**	**Zugriffsart und - beschreibung**	**Zugriff auf Ressourcen**
FC1: 0x01	Read Coils	Lesen eines einzelnen Bits	R: Prozessabbild, PFC-Variablen
FC2: 0x02	Read Input Discretes	Lesen mehrerer Eingangs-Bits	R:Prozessabbild, PFC-Variablen
FC3: 0x03	Read Multiple Registers	Lesen mehrer Eingangs-Register	R:Prozessabbild, PFC-Variablen,Interne Variable, NOVRAM
FC4: 0x04	Read Input Registers	Lesen mehrerer Eingangs-Register	R:Prozessabbild, PFC-Variablen,Interne Variable, NOVRAM
FC5: 0x05	Write Coil	Schreiben eines einzelnen Ausgangsbits	W:Prozessabbild, PFC-Variablen
FC6: 0x06	Write Single Register	Schreiben eines einzelnen Ausgangs-Registers	W:Prozessabbild, PFC-Variablen,Interne Variable, NOVRAM
FC7: 0x07	Read Execption Status	Lesen der ersten 8 Eingangsbits	R:Prozessabbild, PFC-Variablen
FC11: 0x0B	Get Comm Event Counter	Kommunikationsereigniszähler	keine
FC15: 0x0F	Force Multiple Coils	Schreiben mehrerer Ausgangsbits	W:Prozessabbild, PFC-Variablen
FC16:0x0010	Write Multiple Registers	Schreiben mehrerer Ausgangsregister	W:Prozessabbild, PFC-Variablen,Interne Variable, NOVRAM
FC23: 0x0017	Read/Write Registers	Lesen und Schreiben mehrerer Ausgangsregister	W:Prozessabbild, PFC-Variablen,NOVRAM

Tabelle 2.5: MODBUS-Funktioncodes

Um eine Operation auszuführen wird der entsprechende Functioncode und die Adresse des ausgewählten Ein- oder Ausgangskanals angegeben. Das Format und der Beginn der Adressierung kann variieren.

Mit Hilfe dieser Funktionscodes stellt der MODBUS TCP/IP Master eine entsprechende Anfrage an den WAGO Feldbusknoten. Dieser gibt dann eine Antwort-Nachricht an den Master zurück. Ist diese Nachricht fehlerhaft, so wird eine Exception an den Master zurückgegeben. Das MODBUS TCP/IP Protokoll wurde im Rahmen des Real-Dezent Projektes unter Java realisiert. In dieser ersten Realisierung werden die Anfragen permanent an einen WAGO Feldbusknoten gesendet und man erhält auf diese Anfragen eine Antwort-Nachricht und kann dadurch entsprechende Werte ablesen bzw. weiter verarbeiten. Der WAGO Feldbusknoten wird also in unbestimmten Intervallen vom Rechner gepollt.

2.4.3 MODBUS TCP/IP über UPnP

Um Real-Dezent so vernetzt wie möglich zu gestalten, müssen einige Grundlegende Punkte beachtet werden. Zum Einen darf die Überprüfung von Daten, die Rückschlüsse auf die Energieerzeugung bzw. -Verbrauch benötigt werden nicht nur lokal vorliegen, sondern der globale Verbund zwischen den Erzeugern und Verbrauchern muss berücksichtigt werden. Es werden diverse Informationen des globalen Verbundes von Teilnehmern des Energienetzes benätigt um diese in Produzenten und Konsumenten einzuteilen, die die Grundlage für die BGM's darstellen. Die BGM's kümmern sich dann um die Energieverteilung. Diese Informationen werden auch zu Kontrollzwecken benötigt, um z.B. die Phasenwinkel, Frequenzen, etc. anzupassen. Zum anderen muss es auch möglich sein, die Verbraucher und Erzeuger eines einzelnen Teilnehmers mit zu berücksichtigen, um die Energiebilanz des Teilnehmers zu ermitteln. Dazu wird UPnP im Rahmen des Projektes benutzt.

UPnP dient zur Herstellungsübergreifenden Kommunikation zwischen Geräten in einem Netzwerk über das TCP/IP-Protokoll. Dabei exisitiert mindestens ein Control Point in einem Netzwerk, so dass sich Geräte an einem Control Point anmelden können. Wenn sich ein Gerät anmeldet, dann werden die Services, welche das Gerät anbietet, abgefragt und bereitgestellt. Die Services haben wiederum Variablen und Actions, womit sie Aktionen durchführen können. Daher liegt es nahe, das MODBUS TCP/IP Protokoll in UPnP einzukapseln und so Anfragen an das SPS der Firma WAGO zu stellen, um Zugriffe auf die SPSen zu realisieren. Mit dieser Umsetzung wird die Kontrollfunktion und die Berechnung der Bilanzen über UPnP geleistet.

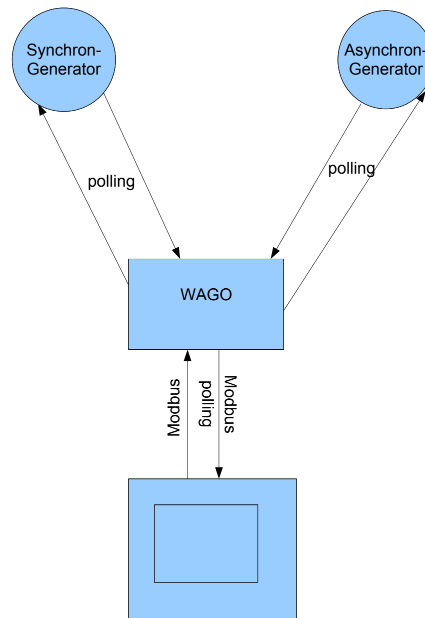


Abbildung 2.7: Polling basierte MODBUS Kommunikation

Bei diesem Ansatz wird das SPS der Firma WAGO auf ein UPnP Device, genannt Wago Device, abgebildet. Dabei werden alle Funktionscodes die MODBUS bereitgestellt, als Services abgebildet um die SPSen zu steuern. Das Übergeben der Parameter für die jeweiligen Funktionscodes wird durch die Actions, die ein Bestandteil des UPnP Standards sind, ermöglicht. Durch diesen Ansatz wird anstatt direkt über die MODBUS API zu kommunizieren, das Protokoll in UPnP eingekapselt. Der Vorteil UPnP zu benutzen liegt in der Hardwareunabhängigkeit. Wenn also ein UPnP Device bereits erstellt worden ist, kann innerhalb eines Netzwerks jeder Control Point auf dieses Device zugreifen, indem der Control Point sich für das Device subscribed. Dadurch wird die begrenzende Lokalität des Gerätes aufgehoben und man kann von überall aus auf das Gerät zugreifen. Bei mehreren Devices ist der Vorteil natürlich noch grösser, damit ist es möglich die Devices, die auf verschiedenen Computern gestartet sind, nur durch ein oder mehrere Control Points zu steuern. Es ist zu beachten, dass bisher das MODBUS TCP/IP Protokoll in der Java-Implementierung, die SPS permanent pollt, d.h. die Anfragen werden in einem Intervall an die Slaves geschickt und diese antwortet jeder Anfrage aufgrund des MODBUS-Protokolls. Eine naheliegende Verbesserung ist, dass der Rechner zwar seine Anfrage über MODBUS verschickt und die zugehörige Antwort erhält, aber jede weitere Kommunikation von dem Slave ausgeht. Dies wird so realisiert, dass die SPS im Fall einer Wertänderung der Daten einer SPS, die Daten dann zum Master schickt. Damit dies funktioniert, wurde ein Protokoll entwickelt, genannt SSP 2.5.1, das in den SPSen implementiert ist. Das Polling bleibt zwar weiterhin bestehen, jedoch auf lokaler Ebene, zwischen WAGO und SPS. Durch diese Verbesserung kann der Netzwerktraffic drastisch reduziert werden, da nicht mehr komplette, teilweise unnötige Messages verschickt werden, sondern nur in den Fällen, wo es auch etwas zu melden gibt.

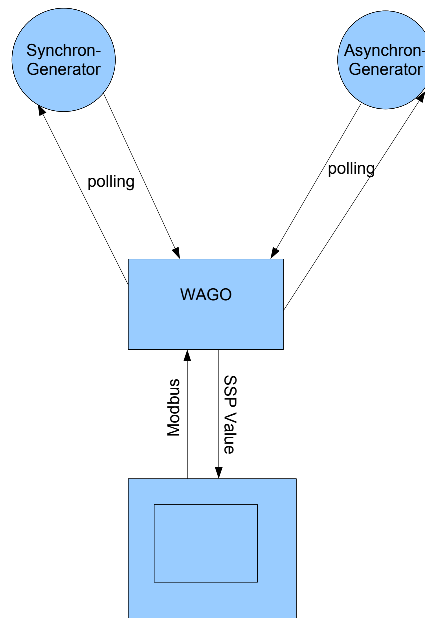


Abbildung 2.8: Eventbasierte Kommunikation mittels SSP n

Um diese Art der Kommunikation zwischen Rechner und WAGO zu realisieren, wird das Entwurfsmuster des Observers benutzt. Dabei werden die Daten, von der SPS kommen in Instanzen von Klassen weitergeleitet, die Observable sind. Die Werte werden dann in UPnP abgebildet, so dass die Kontrollinstanz noch weiterhin bestand hat.

2.5 Das SPS Subscription Protocol (SSP)

2.5.1 Einleitung

Jedes Gerät - Generatoren, Netzgeräte zur Simulation von Solarzellen, etc. - wird über eine SPS gesteuert und kontrolliert. Die SPS sammelt dazu relevante Messdaten und legt sie in ihren internen Speicherregistern ab. Neben den Messwerten können die Register auch Informationen über die aktuelle Tätigkeit der SPS bzw. der angeschlossenen Maschine enthalten. Eine SPS, die eine Synchronmaschine steuert, könnte beispielsweise „Tätigkeiten“ wie „Generator anfahren“, „Netzparallel-Betrieb“, „Standby“ etc. ausführen.

Der Inhalt der SPS-Register sei im Folgenden als *Zustand der SPS* definiert. Dieser Zustand soll in Echtzeit über UPnP zur Verfügung gestellt werden. Da die Implementierung des UPnP -Protokolls auf den SPSen zu aufwändig wäre, ist jeder SPS ein PC als Proxy vorgelagert, der ihren Zustand überwacht und so stets ein aktuelles Abbild der SPS liefern kann.

Wie funktioniert diese Überwachung? Die SPS und der PC sind über TCP/IP miteinader verbunden, somit fällt die Entscheidung über den Kommunikationskanal leicht. Für die konkrete Überwachung des Zustandes bieten sich zwei Konzepte an: zum einen könnte der PC die Zustände in regelmäßigen Abständen abfragen, oder aber die SPS sendet aktiv Nachrichten an den PC, um Zustandsänderungen zu signalisieren.

Da im Allgemeinen davon auszugehen ist, dass die Zustände sich nicht sehr oft verändern, man zum anderen aber in der Lage sein will, auf jede Zustandsänderung schnell zu reagieren, wurde der zweite Ansatz gewählt. Das bedeutet, dass die SPS bei jeder Zustandsänderung eine Nachricht an den PC sendet. Die Form dieser Nachrichten wurde in einem Protokoll spezifiziert, dem SPS Subscription Protocol, im Folgenden als SSP bezeichnet. Bei der Spezifizierung des Protokolls wurde darauf Wert gelegt, dass die Benachrichtigung unabhängig von der Maschine funktioniert, die von der SPS gesteuert wird, und unabhängig von dem PC, der die SPS überwacht. Außerdem besteht die Möglichkeit, nicht den gesamten Speicher zu überwachen, sondern lediglich relevante Bereiche.

Um die Nachrichtenübermittlung unabhängig von der konkreten Implementierung auf der SPS zu machen, werden diese Bereiche in einer Maschinen-spezifischen Zusatz-Spezifikation definiert und können daraufhin über sogenannte *Value-Codes* referenziert werden. Somit muss der überwachende PC nicht die genauen Speicher-Adressen der relevanten Werte kennen, sondern lediglich den Value-Code eines Wertes bzw. von Werte-Gruppen. Falls sich nun Speicherbereiche auf der SPS durch Änderungen in der Implementierung verschieben, muss lediglich SPS-seitig das Mapping der Value-Codes auf Speicherzellen aktualisiert werden. Die Implementierung auf dem PC kann unverändert bleiben.

Weiterhin wird die Adresse des überwachenden PCs nicht hart in die SPS einkodiert, sondern die SPS verwaltet eine dynamische Liste der PCs, die sie im Falle eines Zustandswechsels benachrichtigen soll. Um in die Liste aufgenommen zu werden, schreibt der PC über Modbus seine Adresse in ein bestimmtes Register der SPS, und schreibt zusätzlich den Value-Code, für den er sich interessiert. Dieser Vorgang heißt *Subscription*; wird er erfolgreich beendet, so ist der PC ab jetzt *Subscriber* für einen Value-Code.

2.5.2 Details

Wie bereits beschrieben, muss sich ein PC zunächst subscriben, bevor er Nachrichten erhält. Natürlich kann er sich auch wieder von der Liste streichen lassen, d.h. unsubscriben. Für diese Vorgänge schreibt er seine IP-Adresse über Modbus in ein bestimmtes Register der SPS. Jeden dieser Vorgänge quittiert die SPS mit Nachrichten, die sie über TCP/IP verschickt. Auch Benachrichtigungen über Zustandsänderungen werden so verschickt. Der Aufbau dieser Nachrichten hat stets die in Tab. 2.6 angegebene Form.

Byte	Bedeutung
0	`SSPCode`
1 - 4	`SPSId`
5 - 6	`Length`
7 - `Length`+6	`SSPData`

Tabelle 2.6: Aufbau einer SSP-Nachricht

Zur Zeit sind vier Nachrichten-Typen definiert, die sich jeweils durch einen eindeutigen `SSPCode` auszeichnen.

`SPSId` gibt die Id der sendenden SPS an. Jeder SPS ist eine Id zugewiesen. Diesen Ids kann an zentraler Stelle eine IP zugeordnet werden. Durch diese Abstraktion der Netzwerkebene kann leicht auf Änderungen der Adressvergabe im Netzwerk reagiert werden, da lediglich die Zuordnungstabelle aktualisiert werden muss.

`Length` gibt die Länge der Nachricht, genauer die Länge von `SSPData` in Words, an.

`SSPData` enthält die eigentlichen zu übermittelnden Daten. Die Struktur ist abhängig vom jeweiligen SSP-Code.

SSP-Codes

`SSPCode1_Values` leitet eine Nachricht ein, die bei Zustandsänderungen an den PC übermittelt wird. `SSPData` enthält dann einen ValueCode eine Liste von WORDs (`SSPValues`), die den zugehörigen Speicherhinhalt enthält. Die Interpretation ist Maschinen-spezifisch und muss in einer zusätzlichen Spezifikation für jede Maschine definiert werden.

`SSPCode2_Unsubscribe` und `SSPCode3_Subscription` quittieren die entsprechenden Anfragen eines PCs über Modbus und enthalten ein Status-WORD, welches Aufschluss über den Erfolg oder Misserfolg der Subscription/Unsubscription gibt.

`SSPCode0_PONG` ist als Antwort auf eine Ping-Nachricht seitens des PCs gedacht. Damit soll abgefragt werden können, ob die SPS SSP unterstützt. Dies ist jedoch noch nicht implementiert.

Weitere Details sind der Spezifikation des SSP-Protokolls zu entnehmen.

2.5.3 Maschinen-spezifische Spezifikationen

Zur Zeit sind lediglich Value-Codes für SPSen spezifiziert, die Synchronmaschinen steuern. Da die möglichen Zustände einer Asynchronmaschine eine Untermenge der Zustände einer Synchronmaschine darstellt, gelten diese Value-Codes ebenfalls für SPSen an Asynchronmaschinen.

Value-Codes für Synchronmaschinen

Es sind zwei Value-Codes definiert.

- `ValueCode0_SensorData` wird getriggert, wenn sich die von den Sensoren gemessenen Werte über einen Grenzwert hinaus verändern. `SSPValues` hat den in Tab. 2.7 angegebenen Aufbau. Dabei ist zu beachten, dass die übermittelten Werte lediglich Rohdaten sind, die durch geeignete Algorithmen in verwertbare Größen umgewandelt werden müssen.

Word	Bedeutung
0	Frequenz der erzeugten Spannung
1	erzeugte Blindleistung
2	Ausgangsspannung
3	erzeugte Wirkleistung
4	Erregerstrom
5	Drehzahl des Antriebs
6	Leistung des Antriebs

Tabelle 2.7: Aufbau von `SSPValues` für `ValueCode0_SensorData`

- `ValueCode1_State` wird getriggert, wenn sich der Betriebszustand der überwachten Maschine ändert. `SSPValues` hat den in Tab. 2.8 angegebenen Aufbau.

Word	Bedeutung
0	`MachineState`

Tabelle 2.8: Aufbau von `SSPValues` für `ValueCode1_State`

`MachineState` kann die in Tab. 2.9 aufgelisteten Werte annehmen.

Wert	Bedeutung
1	KW_Aus
2	KW_Init
3	KW_Stillstand
4	KW_AntriebHochfahren
5	KW_Hochgefahren
6	KW_Runterfahren
7	Abschalten
8	KW_Ausregeln
9	ErregungEinschalten
10	Erregung_Abschalten
11	Zuschaltbereit
12	Synchronisieren
13	Zugeschaltet
14	TrennenVomNetzmodell

Tabelle 2.9: Werte für `MachineState`

2.5.4 Einschränkungen

Aufgrund eines Bugs in der Implementierung des Netzwerk-Sockets der verwendeten SPSen kann sich nur jeweils ein PC bei einer SPS subscriben. Ist dieser PC subscribed, können keine Statusmeldungen an andere PCs versendet werden, die ebenfalls versuchen sich zu subscriben (siehe auch Kap. 2.6.3).

2.6 Realisierung des SSP auf einer SPS

2.6.1 Einleitung

Bei der Implementierung eines Protokolls zur Steuerung der Kommunikation zwischen einer SPS und einem PC gilt es sich zunächst zu überlegen, auf welchen Protokollen das neue Protokoll aufsetzen soll. In den vorangegangen Abschnitten wurde schon erwähnt, dass sich für die Kommunikation vom PC zur SPS das eigens für diese Kommunikation entworfene Modbus-Protokoll als ideal herausstellt. Mit Hilfe des Modbus-Protokolls lassen sich auf einfache Weise vom Programmierer vorher festgelegte Register auf der SPS schreiben und auslesen. Damit unterstützt das Modbus-Protokoll die zugrunde liegende SPS-Hardware und deren Struktur auf der Empfängerseite vollkommen. Das Modbus-Protokoll bietet somit die ideale Grundlage für die Kommunikation vom PC zur SPS.
Für die Kommunikation von der SPS zum PC ist das Modbus-Protokoll als Sub-Protokoll auf Grund zu hohen Overheads nicht geeignet. Ein PC müsste die Register einer SPS simulieren bzw. die Modbus-Nachrichten entsprechend parsen um anschließend die eigentliche Nachricht entnehmen zu können. Der Vorteil, dass die Modbus-Nachrichten die interne SPS Hardware unterstützen, führt auf der Seite des PC's beim Empfangen solcher Nachrichten zu mehr Aufwand. Deshalb wird für die Kommunikation in Richtung SPS zum PC das TCP/IP-Protokoll genutzt. Das TCP/IP-Protokoll bietet für die Kommunikation von der SPS zum Rechner den entscheidenden Vorteil, dass die damit konstruierten

Nachrichten kleiner, der Aufwand beim Empfangen somit geringer und die Kommunikation effizienter und schneller durchgeführt werden kann.
Eine effiziente Kommunikation zwischen PC und SPS im Rahmen des Real-Dezent Projektes besonders von Bedeutung, da die SPS'en komplexere Anlagen wie zum Beispiel Kraftwerke steuern können. Die Daten der Kraftwerke müssen also auf besonders schnellen Wege den PC's mitgeteilt werden, damit der Dezent Algorithmus sie in seine Berechnung mit einbeziehen kann.

Im Folgenden wird in diesem Abschnitt gezeigt, wie die Implementierung des SSP Protokolls basierend auf dem TCP/IP-Protokolls auf der SPS durchgeführt wurde. Dazu werden zunächst die zur Kommunikation über das TCP/IP-Protokoll verwendeten Bausteine der Bibliothek Ethernet.lib vorgestellt. Im Anschluss daran wird sich dem SPS-Code zur Umsetzung des SSP-Protokolls zugewandt und die Basisfunktionen genauer erläutert.

2.6.2 Die Bibliothek Ethernet.lib

Die Bibliothek "Ethernet.lib", die standardmäßig bei der Wago-Vollversion mitgeliefert wird, enthält Bausteine für die Kommunikation über ein Netzwerk. Dabei wird unter anderem sowohl die Kommunikation über TCP als auch über UDP unterstützt. Für die Kommunikation über TCP oder UDP stehen die Bausteine ETHERNET_CLIENT_OPEN und ETHERNET_CLIENT_CLOSE zum Verbindungsauf- und -abbau sowie der Baustein ETHERNET_WRITE zum Versenden entsprechender Nachrichten zur Verfügung. Da diese Bausteine die Grundlage für das SSP-Protokoll liefern, soll im Folgenden kurz erläutert werden, wie mit Hilfe dieser Bausteine eine Kommunikation durchgeführt werden kann. Des Weiteren wird auf einige technische Eigenheiten hingewiesen, die sich bei der Arbeit mit den Bausteinen ergeben und die den Entwurf des SSP-Protokolls beeinflusst haben.

Der Baustein "ETHERNET_CLIENT_OPEN"

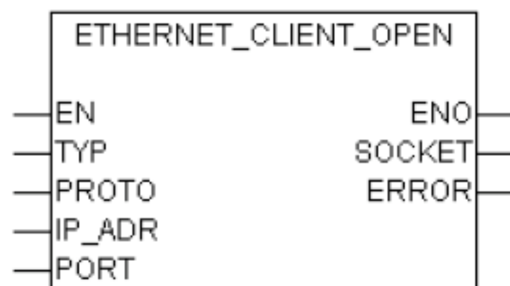


Abbildung 2.9: Graphische Darstellung des Bausteins Ethernet_Client_Open

Der Baustein "ETHERNET_CLIENT_OPEN" ist ein Funktionsblock und verfügt über die fünf Eingabeparameter `EN, PROTO, IP_ADR, PORT, TYP`. Dabei definiert der Parameter `PROTO` (Protokoll) das zu verwendende Protokoll (TCP o. UDP etc.), der Parameter `TYP` legt die Kommunikationsart fest. Bei der Kommunikationsart unterscheidet der Baustein zwischen den drei Arten SOCK_STREAM, ein kontinuierlicher Bytestrom, SOCK_DGRAM, Unterstützung von Datagrammen und SOCK_RAW, welche den Zugang zu den internen Netzwerkprotokollen bereitstellt.

[1] Durch den Wechsel des Eingabeparameter `EN` von FALSE auf TRUE wird, bei gleichzeitiger Eingabe einer IP-Adresse über den Parameter `IP_ADR`, ein Socket erzeugt und der Verbindungsaufbau zu der angegebenen IP-Adresse eingeleitet. Dabei ist zu beachten, dass nie mehr als zwei Sockets gleichzeitig geöffnet werden können. Die Beschränkung auf zwei Sockets gleichzeitig wird in dem nachfolgenden Entwurf des SSP-Protokolls noch einige Konsequenzen haben, so muss bei der Kommunikation mit mehreren unterschiedlichen PCs jeweils die Verbindung wieder geschlossen und erneut aufgebaut werden.
Der Verbindungsaufbau ist erfolgreich beendet, wenn die Rückgabevariable `ENO` auf TRUE wechselt und die Variable `ERROR` den Wert Null annimmt. Nachdem die Verbindung erfolgreich aufgebaut wurde, enthält die Rückgabevariable SOCKET die Socket-ID, an Hand derer die Verbindung eindeutig identifiziert und den anderen Bausteinen bekannt gemacht werden kann.

Der Baustein "ETHERNET_CLIENT_CLOSE"

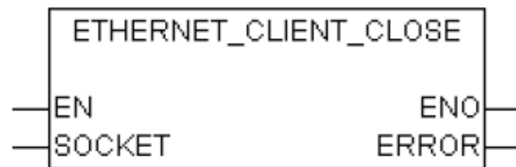


Abbildung 2.10: Graphische Darstellung des Bausteins Ethernet_Client_Close

Der Baustein "ETHERNET_CLIENT_CLOSE" dient zum Schließen eines durch den Baustein "ETHERNET_CLIENT_OPEN" geöffneten Sockets. Dazu muss an dem Eingabeparameter `SOCKET` die Socket-ID des geöffneten Sockets anliegen und der Parameter `EN` von FALSE auf TRUE wechseln.
Ist die Verbindung erfolgreich abgebaut, wird der Socket geschlossen und der Ausgabeparameter `ENO` wechselt auf TRUE. Tritt während des Abbaus ein Fehler auf, kann die Fehler-ID über den Rückgabeparameter `ERROR` ausgelesen werden.

Der Baustein "ETHERNET_WRITE_PT"

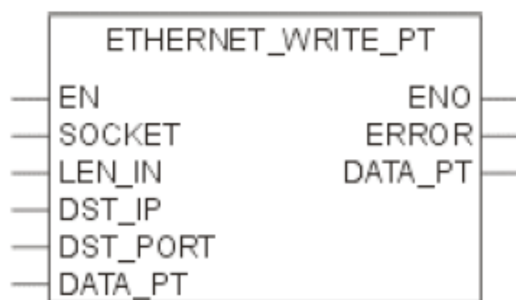


Abbildung 2.11: Graphische Darstellung des Bausteins Ethernet_Write_Pt

Der Baustein "ETHERNET_WRITE_PT" der Bibliothek Ethernet.lib wird schließlich zum Senden von Nachrichten über die geöffneten Sockets benutzt. Dabei identifiziert der Eingabeparameter SOCKET

den vorher geöffneten Socket.
Die zu schreibenden Daten werden als Pointer auf ein Byte Array übergeben, das vorher im Speicher angelegt und mit den entsprechenden Daten versehen werden muss. Dabei kann die Anzahl der zu schreibenden Bytes zwischen 0 und 1500 variieren. Die tatsächliche Länge muss daher durch die Variable `LEN_IN` übergeben werden.
Wie bei den Bausteinen "ETHERNET_CLIENT_OPEN" und "ETHERNET_CLIENT_CLOSE" beginnt das Senden mit dem Setzten der EN Variable von FALSE auf TRUE. Ist die Nachricht erfolgreich gesendet worden, wird die Rückgabevariable `ENO` auf TRUE wechseln. Tritt während der Kommunikation ein Fehler auf, enthält die Variable Error die Fehler-ID.

2.6.3 Kommunikation zwischen einer SPS und einem TCP Client

Der Baustein SSP_sendMessage (FB) kombiniert die drei Bausteine "ETHERNET_CLIENT_OPEN", "ETHERNET_CLIENT_CLOSE" und "ETHERNET_WRITE_PT", so dass die Bausteine für die Kommunikation zwischen SPS und PC effizient genutzt werden können.
Da der Baustein ETHERNET_CLIENT_OPEN nur maximal zwei Sockets gleichzeitig öffnen kann, muss die Verbindung jedesmal geschlossen werden, bevor eine Nachricht zu einer andern IP-Adresse geschickt werden kann. Gleichzeitig haben Tests mit der "Ethernet.lib" Bibliothek ergeben, dass häufiger Verbindungsauf- und -abbau zu Fehlern auf der SPS führen, so dass die SPS nur noch mit einem manuellen Reset (Strom aus) zurückgesetzt werden kann.
Diese beiden Einschränkungen haben weitreichende Folgen auf den Entwurf des Bausteins sowie auf den Entwurf des SSP Protokolls. Die Beschränkung auf maximal zwei Sockets gleichzeitig und das Problem, dass eine Verbindung nicht ständig ab- und wieder aufgebaut werden darf, macht es unmöglich mehr als einen PC durch das SSP-Protokoll zu unterstützen. Das Protokoll wurde daraufhin so geändert, dass sich nun nur maximal ein PC gleichzeitig auf der SPS subscriben kann.
Für die Kommunikation zwischen SPS und PC und speziell für den Baustein SSP_Message bedeutet dies, dass mit hoher Wahrscheinlichkeit mehrere Nachrichten über einen Socket an die gleiche IP-Adresse verschickt werden. Es ist daher sinnvoll einen initiierten Socket nicht gleich wieder zu schließen, sondern solange geöffnet zu lassen, bis eine Nachricht zu einer anderen IP-Adresse verschickt werden soll.

Die folgende Tabelle gibt Aufschluss über die Ein- und Ausgabeparameter des neuen Bausteins:

Eingabeparameter	**Typ**	**Bedeutung**
`start_sending`	BOOL	Startet den Sendevorgang
`ipAdresse`	STRING	die IP Adresse des Empfängers
`length`	WORD	Anz. zu sendender Bytes
`data`	POINTER TO BYTE	Pointer auf Byte Array mit Sendedaten
Ausgabeparameter	**Typ**	**Bedeutung**
`sending_finish`	BOOL	TRUE falls Sendevorgang erfolgreich
`gotConnectionError`	BOOL	TRUE im Fehlerfall

Tabelle 2.10: Parameter des Bausteins SSP_sendMessage

Beim Wechseln des Parameters von FALSE auf TRUE wird das Senden durchgeführt. Falls noch kein Socket erstellt wurde, wird zunächst ein Socket erstellt und danach gesendet. Wurde die Nachricht

erfolgreich gesendet, d.h. trat kein Fehler auf, so wird die Rückgabevariable `sending_finish` gesetzt. Der Socket bleibt geöffnet, solange die gleiche IP-Adresse anliegt. Wird eine andere IP-Adresse angelegt, so wird der Socket zunächst geschlossen, ein neuer Socket zu dem neuen Empfänger aufgebaut und die Nachricht verschickt. Im Falle eines Fehlers wird zunächst versucht die Verbindung abzubauen und anschließend die Flagvariable `gotConnectionError` sowie `sending_finished` gesetzt. Der Benutzer hat dann die Möglichkeit auf den Fehler zu reagieren und bei Bedarf erneut eine Verbindung aufzubauen.
Das Nachfolgende Zustandsdiagramm soll dieses Verhalten noch einmal genauer erläutern:

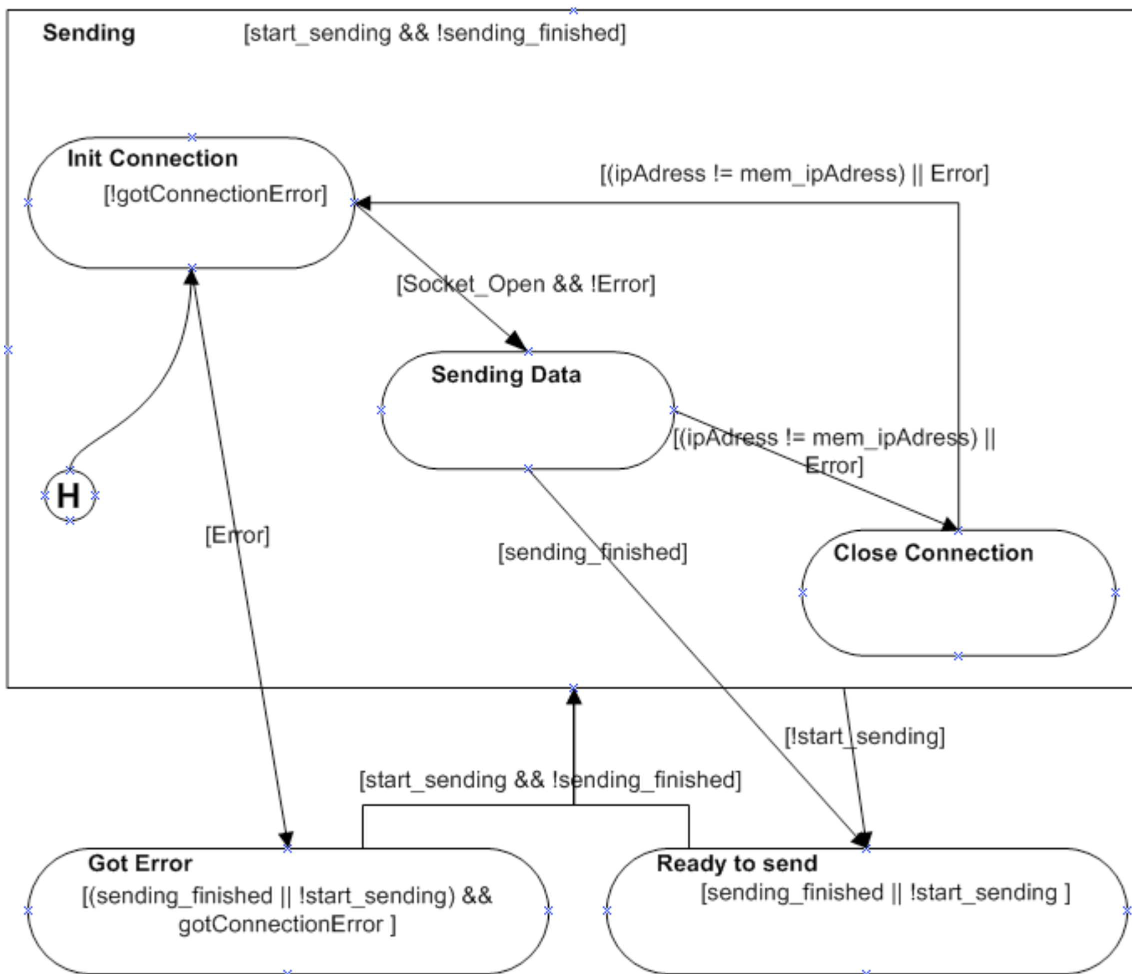


Abbildung 2.12: Zustandsdiagramm des SSP_sendMessage Blocks

2.6.4 Der Notify-Mechanismus des SSP-Protokolls

Das Ziel des SSP-Protokolls ist, einen außenstehenden PC darüber in Kenntnis zu setzten, wenn sich Register oder Ein-/Ausgangsparameter der SPS ändern. Dazu müssen die entsprechenden Register auf der SPS von Seiten der SPS überwacht und ggf. Nachrichten geschickt werden. Des Weiteren sollen sich die einzelnen PC's für unterschiedliche Register einschreiben können. Die PCs sollen nur

dann benachrichtigt werden, wenn die entsprechenden Register, für die sich die PCs registiert haben, verändert werden.
In Java gibt es für solche Zwecke entsprechende Ereignis-Routinen, die ein Event auslösen, sobald eine ausgewählte Variable den Wert ändert und den Benutzer somit darüber informieren. Da die SPS-Sprache einen vergleichbaren Mechanismus nicht implementiert, muss dieser von Hand nachgerüstet werden.
Der Baustein SSP_NotifyObservers(PRG) implementiert einen ähnlichen Mechanismus, mit dem man Variablen auf Veränderungen hin überwachen kann. Der grobe Ablauf ist dem folgenden Zustandsdiagramm in der Ablaufsprache (AS) der SPS-Programmiersprache zu entnehmen:

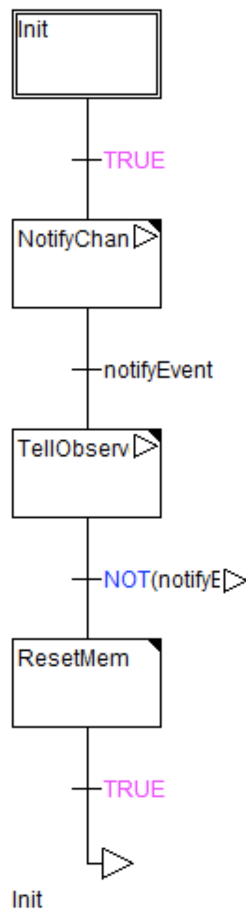


Abbildung 2.13: Grober Ablauf des Notify-Mechanismus’

Dabei überwacht der Block “NotifyChanges” die Veränderungen der Registerwerte. Zu diesem Zweck wird für jedes zu überwachende Register x eine zusätzliche Variable mem_x eingeführt, die den letzten Wert der Variable x enthält. Für die Feststellung einer Veränderung reicht der Vergleich

$$x \neq mem_x$$

In der Praxis, vor allem dann, wenn analoge Eingänge überwacht werden, die von Natur aus mit einem gewissen Rauschen hinterlegt sind, ist diese Art des Vergleichs aber zu ungenau. Zu diesem Zweck wird ein δ Wert definiert, so dass, wenn sich der Wert des Registers x um mehr als δ von mem_x

unterscheidet, eine Benachrichtigung ausgelöst wird.
Wird eine Änderung eines Registers auf diesem Wege bemerkt, muss sich zusätzlich der zugehörige "Value Code" des geänderten Registers gemerkt werden, so dass nur die PCs benachrichtigt werden, die sich auch für das entsprechende Register eingeschrieben haben. Das folgende Code-Stück zeigt einen charakteristischen Ausschnitt des "NotifyChanges" Block:

```
IF (mem_DrehZahlAnalog  < 65535-ssp_ABSSchwankung) THEN
IF (mem_DrehZahlAnalog + ssp_ABSSchwankung < DrehZahlAnalog) THEN
   notifyEvent := TRUE;
   notifiedValueCode := 0;
END_IF
END_IF

IF (mem_DrehZahlAnalog > ssp_ABSSchwankung) THEN
IF (mem_DrehZahlAnalog  - ssp_ABSSchwankung > DrehZahlAnalog) THEN
   notifyEvent :=TRUE;
   notifiedValueCode := 0;
END_IF
END_IF
```

Beim Setzen der `notifyEvent` Variable auf TRUE wird der Block "TellObservers" betreten, der nun sukzessive den in der SPS eingetragenen PC mit entsprechendem "ValueCode" eine Werteänderungsnachricht (SSPCode 1) schickt.

2.6.5 Die "Input loop"

Damit ein PC über Modbus SSP-Nachrichten an die SPS verschicken kann, muss zunächst ein entsprechender Speicherbereichs auf der SPS definiert werden, in den der Rechner die Nachrichten schreiben kann. Dieser Speicherbereich muss vorher fest definiert und beiden Parteien bekannt gemacht werden, damit sowohl der PC, der die Nachrichten in die festgelegten Register schreiben muss, als auch die SPS, die die festgelegten Register "abhorchen" muss, miteinander kommunizieren können. In diesem Abschnitt soll der SPS Code und die interne Speicherstruktur der SPS im Hinblick auf die Implementierung des SSP Protokolls genauer erläutert werden.

Reservierte Speicherbereiche

In der folgenden Tabelle ist der Aufbau einer SSP-Nachricht und der definierte zugehörige Speicherbereich des SSP-Protokolls aufgelistet.

Sp.-bereich (auf SPS)	Sp.-bereich (Modbus)	Länge	Funktion
%MW0	0x3000	1 Word	`SSPCommandCode`
%MW1	0x3001	2 Words	`SenderIP`
%MW3	0x3003	1 Word	`SSPCmdData`

Tabelle 2.11: Aufbau einer SSP Nachricht vom PC zur SPS

Die Reservierung des entsprechenden Speicherbereich kann auf der SPS wie folgt aussehen:

```
(*----------------- SSP Input ---------------------*)
  sspInputCode AT %MB0 : WORD;
  sspInputData AT %MB2 : ARRAY [0..10] OF BYTE;
(*-----------------------------------------------*)
```

Die Adressierung %MB2 (Byte-Adressierung) deckt sich dabei mit der in der Tabelle aufgeführten Adressierung %MW1 (Word-Adressierung).

Abhandlung einer SSP-Nachricht

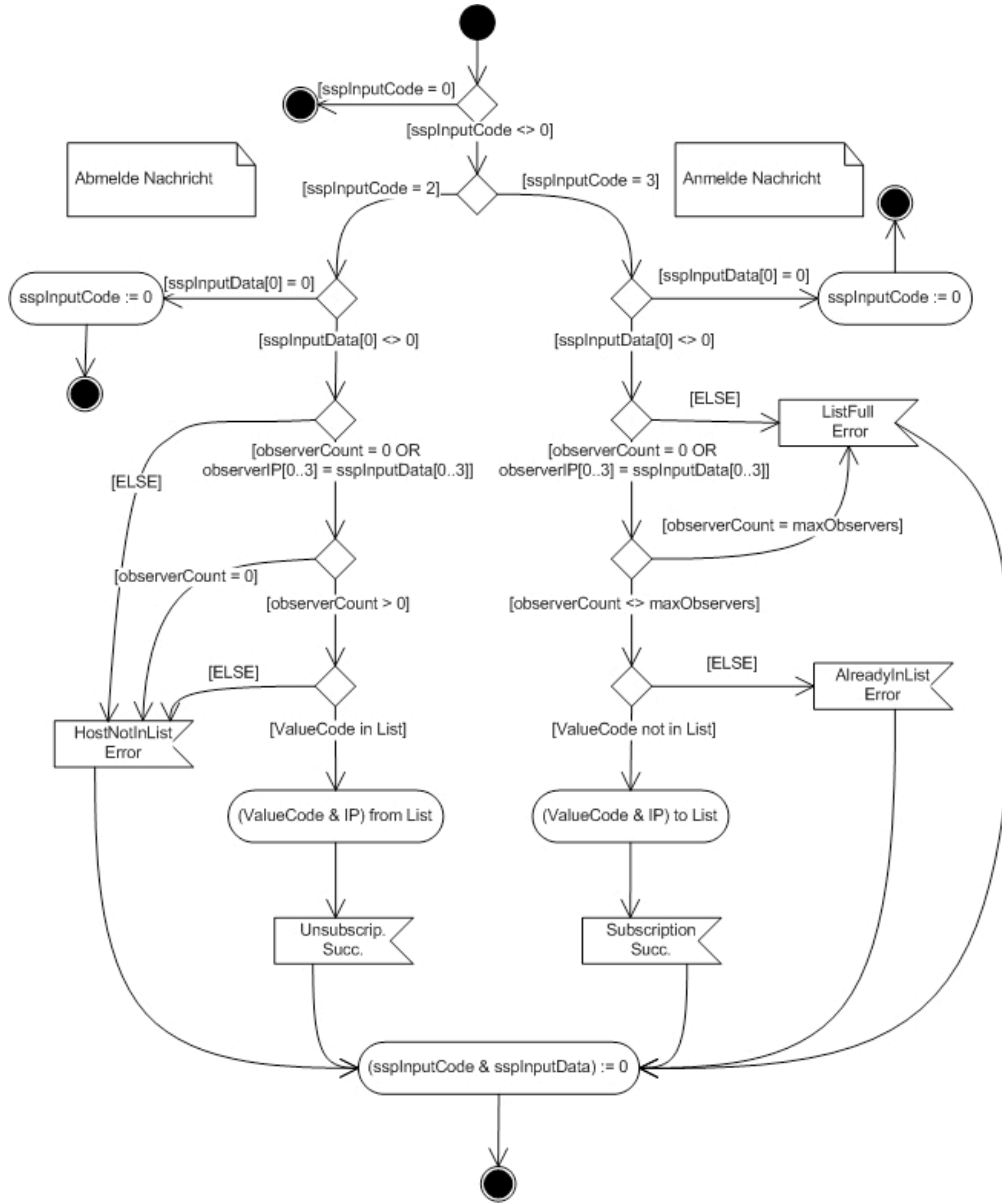


Abbildung 2.14: Abhandlung einer SSP-Nachricht

Das oben aufgeführte Aktivitätsdiagramm zeigt die Abhandlung einer SSP-Nachricht. Die Sendeaktionen `ListFullError`, `AlreadyInListError`, `HostNotInListError`, `SubscriptionSucc` und `UnsubscriptionSucc` stellen SSP-Nachrichten mit den entsprechenden Fehler bzw. Erfolgsnummern da.

2.6.6 Die Taskkonfiguration

In der Taskkonfiguration einer SPS kann man mehrere “Tasks” konfigurieren, die in bestimmten Mustern aufgerufen werden. Zur Auswahl des Aufrufsmusters für eine Task stehen die Optionen:

- `zyklisch`
- `freilaufend`
- `Ereignis gesteuert`

`Zyklisch`: Der Benutzer kann ein Zeitintervall angeben, nach dessen Ablauf der Task aufgerufen wird. Der Aufruf wiederholt sich zyklisch jeweils nach Ablauf des angegebenen Zeitintervalls.

`Freilaufend`: Der Task wird immer dann aufgerufen, wenn die SPS “nicht beschäftigt” (“idle”), d.h. der Prozessor der SPS keine weiteren Aufgaben ausführt.

`Ereignis gesteuert`: Der Benutzer muss eine Boolesche Variable angeben, die beim Wechseln von FALSE auf TRUE den Aufruf der Task auslöst.

Die einzeln definierten Tasks werden nach den definierten Mustern aufgerufen. Der Benutzer kann an jede Task ein oder mehrere Programme (Bausteine) anhängen, die bei dessen Aufruf *sequentiell* abgearbeitet werden. Dabei kann jeweils nur eine Task und innerhalb der Tasks nur ein Programm *gleichzeitig* aufgerufen werden.
Abhängigkeiten zwischen einzelnen Tasks kann der Benutzer durch Angabe einer Prioritätsnummer definieren. Die Skala der Nummern reicht von 0 bis 31, wobei 0 die höchste Priorität repräsentiert.

Da der Subscription-Mechanismus nur im Hintergrund eines Hauptprogramm wie beispielsweise einer Generatorsteuerung laufen soll, werden für die “Input Loop”, den “Notify Mechanismus” und das Senden von Nachrichten eigene Tasks definiert. Die “Input Loop” wird dabei nur jede Sekunde einmal aufgerufen. Dies sollte ausreichen um die notwendigen Subscription- und Unsubsription-Nachrichten zu empfangen und zu interpretieren (im Normalfall soll nur zu Beginn/Ende der Überwachung eine Subscription/Unsubscription Nachricht verschickt werden). Der Notify-Mechanismus und das Senden von Nachrichten wird hingegen jede 20 ms (Notify) bzw. 40 ms (Senden) aufgerufen.

2.7 Benutzerschnittstellen der Laborumgebung

In diesem und den folgenden Abschnitten sollen die GUI’s vorgestellt werden, mit Hilfe derer die Kraftwerke der Laborumgebung bedient werden können. Die wichtigsten Klassen sind dabei die SMControlGUI, ASMControl, MachineValueGUI sowie die RobustMachineValueGUI. Die SMControlGUI/ASMControlGUI bietet dem Benutzer die Möglichkeit den Zustand des Generators zu beeinflussen. Über Textfelder und Buttons kann der Benutzer z.B. die Drehzahl des Generators beeinflussen, den Erregerstrom einschalten oder den Generator automatisch auf ein anderes Netz synchronisieren. Die SMControlGUI ist also im Wesentlichen eine Visualisierung des MC-Protokolls.
Zusätzlich zu den Textfeldern, die dem Zweck dienen den Generator zu beeinflussen, kann der Benutzer über zwei Buttons die MachineValueGUI bzw. die RobustMachineValueGUI starten. Diese beiden GUI’s dienen dazu die aktuellen Messwerte des Generators anzuzeigen. Die RobustMachineValueGUI

unterscheidet sich von der MachineValueGUI dadurch, dass sie neben den aktuellen Messwerten auch noch aufbereitete Werte wie zum Beispiel den Median, Durchschnitt, etc. anbietet. Die MachineValueGUI wirkt dadurch etwas aufgeräumter und bietet Aufschluss über die Blind-/ Wirkleistung, den Erregerstrom, die Spannung, die Drehzahl und die Frequenz. Die Werte werden bei beiden GUI's über das SSP-Protokoll aktualisiert, d.h. es werden nur dann neue Messwerte angezeigt, wenn sich die Messwerte über einen auf der SPS einstellbaren Mindestabstand hinaus ändern.
Das folgende Klassendiagramm in Abbildung 2.15 enthält alle erwähnten Klassen sowie deren wichtigsten Methoden und Attribute. Dabei wurden vor allem die Standardkomponenten (JTextField, JButton, etc.) zu Zwecken der besseren Übersicht weggelassen. Die einzelnen Klassen und deren Beziehungen werden danach in den nächsten Abschnitten genauer vorgestellt.

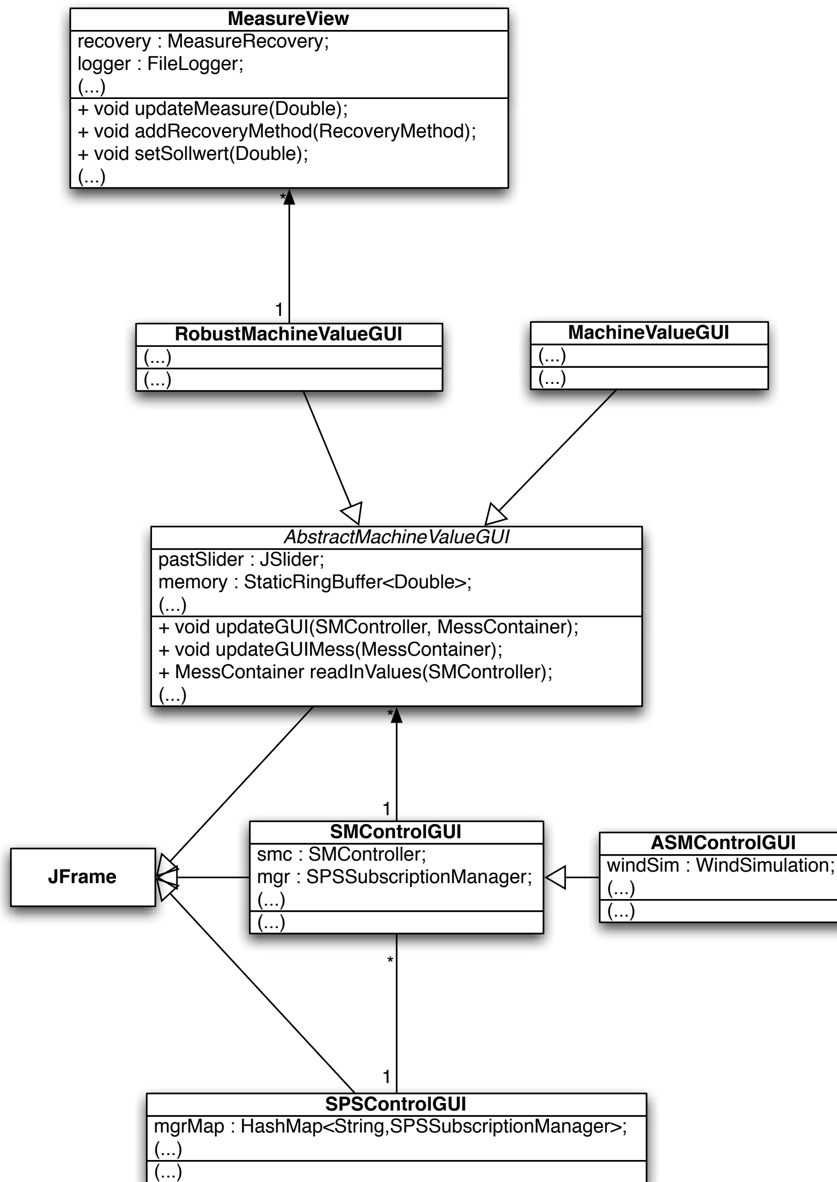


Abbildung 2.15: Klassendiagramm

2.7.1 SPS Control GUI

Die SPSControlGUI bietet eine einfache Benutzerschnittstelle, über die der Benutzer die SMControlGUI's der einzelnen Kraftwerke starten kann. Die GUI besteht aus zwei Comboboxen und einem "Verbinde"-Button.

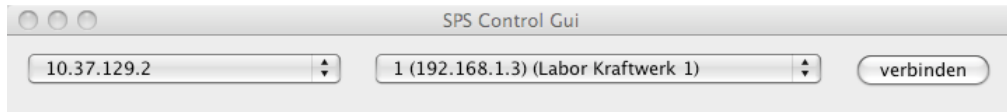


Abbildung 2.16: SPS Control GUI

Der Benutzer kann die SMControlGUI für ein Kraftwerk starten, indem er in der linken Combobox die IP-Adresse [4] des Rechners, auf dem die GUI gestartet wurde, auswählt. In der rechten Combobox muss die IP-Adresse der Generator-SPS des Kraftwerks, zu dem er sich verbinden will, ausgewählt werden. Ein Klick auf den Button startet eine Instanz der SMControlGUI/ASMControlGUI zu dem ausgewählten Kraftwerk. Der Verbindungsaufbau zu der Generator-SPS wird dann von einer Instanz der Klasse SSPSubscriptionManager übernommen, die die SPSControlGUI der SMControlGUI bzw. ASMControlGUI mit übergibt.

2.7.2 (A)Synchron Machine Control GUI

Wenn eine neue Instanz der Klasse (A)SMControlGUI erzeugt wird, versucht die GUI zunächst eine Verbindung zur zugehörigen Generator-SPS aufzubauen. Der Verbindungsaufbau erfolgt über die von der SPSControlGUI übergegeben Instanz der Klasse SSPSubscriptionManager, die sich für die Value-Codes 0, 1 und 3 im Generator registriert. Misslingt einer dieser Versuche, wird das Fenster wieder geschlossen und eine entsprechende Fehlermeldung wird ausgegeben.
Das Subscriben für die drei ValueCodes erfüllt mehrere Zwecke:

- Durch das erfolgreiche Subscriben auf der jeweiligen SPS wird sichergestellt, dass das richtige Programm auf der SPS gestartet wurde und sich kein anderer PC auf der SPS registriert hat.
- Der ValueCode 3 teilt der Generator-SPS mit, dass der Rechner über Zustandsänderungen des Generators informiert werden will.
- Über den ValueCode 1 wird der Rechner nun über Änderungen der Messwerte des Generators informiert.

Wurde die Anmeldung an der SPS erfolgreich ausgeführt, wird eine Instanz der Klasse NodeAdapter erzeugt, der den jeweiligen Generator als UPnP Device im Netzwerk zur Verfügung stellt.
Der Aufbau der (A)SMControlGUI ist wie folgt: im Kopf der (A)SMControlGUI wird der aktuelle Generatorzustand angezeigt. Die Steuerung des Generators erfolgt über die einzelnen Textfelder und Buttons der GUI. Intern wird bei einem Klick auf einen Button die jeweilige Methode des SMController's aufgerufen, der die entsprechende MCP-Nachricht generiert und diese an die Generator-SPS sendet.

[4] Es stehen eine Handvoll möglicher IP Adressen zur Auswahl

Abbildung 2.17: Synchron Machine Control GUI

Über die beiden unteren Buttons kann jeweils eine Instanz der Klassen MachineValueGUI bzw. der RobustMachineValueGUI gestartet werden.

Unterschiede der Asynchron Machine Control GUI

Wie der Abbildung 2.15 zu entnehmen ist, ist die ASMControlGUI eine Unterklasse der SMControlGUI. Damit gleichen sich der Aufbau sowie die Grundfunktionen (setzen der Blind- / Wirkleistung etc.) der SMControlGUI. Konzeptionell ergibt sich hieraus zunächst ein Problem:

- Der Aufbau der Asynchronmaschine unterscheidet sich grundlegend von dem Aufbau einer Synchronmaschine und unterstützt somit beispielsweise das Setzten des Erregerstroms sowie der Blindleistung nicht.

Dieser Entwurf ist an dieser Stelle aber nur die konsequente Weiterführung der Modellierung auf der SPS-Ebene, da sich die Projektgruppe vor allem aus zeitgründen dafür entschieden hat, für die Steuerung der Asynchronmaschine das gleiche SPS-Programm wie für die Synchronmaschinen zu verwenden. Für den Benutzer bedeutet dies, dass er die in der GUI vorhanden Funktionalitäten (setzen der Blindleistung / des Erregerstroms) zwar benutzen darf, aber die entsprechenden Parameteränderungen von der Maschine ignoriert werden.
Zusätzlich zu den eingeschränkten Funktionen der Synchron Machine Control GUI besitzt die Asynchron Machine Control GUI die Möglichkeit ein in Form einer CSV Datei vorhandenes Leistungsprofil abzufahren. Zu diesem Zweck wurde die ASMControlGUI um einige weitere Steuerelemente

ergänzt, die sich in Form einer Zeile am unteren Ende der ASMControlGUI befinden. Der Benutzer hat die Möglichkeit die CSV-Datei mit den Simulationsdaten zu laden, einen geeigneten Zeitraum aus den Simulationsdaten auszuwählen, die Dauer der Simulation festzulegen sowie das Zeitintervall, nach dem die Simulation Sollwerte an den Generator schickt, zu setzen. Mit den drei Buttons “Set”, “Start” und “Stopp” kann die Simulation manuell gestartet sowie gestoppt werden und zusätzlich mittels “Set” für den synchronen Start über UPnP vorbereitet werden. Das Label in der rechten unteren Ecke zeigt den aktuellen Status der Simulation an.

Abbildung 2.18: Asynchron Machine Control GUI

2.7.3 Machine Value GUI

Die Machine Value GUI ist in drei Teile aufgeteilt. Im Kopf wird der aktuelle Zustand des Generators[5] angezeigt. Der Hauptteil der GUI wird dazu benutzt, die Messwerte des Generators anzuzeigen. Die Messwerte sind in drei Gruppen zu je zwei Messwerten eingeteilt:

1. Leistung
    - Wirkleistung
    - Blindleistung
2. IErreg. / Spannung
    - Erregerstrom
    - Spannung
3. Drehzahl / Frequenz

[5] Der Wertebereich des Generatorzustandes bezieht sich auf die im MC-Protokoll definierten Zustände

- Drehzahl
- Frequenz

Am Fuß der GUI befindet sich ein Slider mit deren Hilfe sich der Benutzer die letzten 100 Messwerte anzeigen lassen kann.
Damit die Messwerte von der GUI angezeigt werden können, muss die Instanz der Klasse SMCon-

Abbildung 2.19: Machine Value GUI

troller überwacht werden, die sich wiederum bei der Generator-SPS für den ValueCode 3 subscriben muss. Nach erfolgreichem Subscriben werden die Messwerte bei jeder Änderung erneuert, und in den dafür vorgesehenen Feldern angezeigt.

Robust Machine Control GUI

Die Robust Machine Value GUI ist ähnlich aufgebaut wie die Machine Value GUI. Sie unterscheidet sich im Wesentlichen dadurch, dass sie neben den Messwerten auch aufbereitete Messwerte zur Verfügung stellt. Die Messwertaufbereitung bezieht sich standardmäßig auf die letzten fünf Messwerte. Diese Anzahl kann aber vom Benutzer jeder Zeit durch einen Rechtsklick auf einen Messwertbereich geändert werden. Die Messwerte werden mit unterschiedlichen Methoden aufbereitet und die aufbereiteten Werte werden rechts vom jeweiligen Messwert angezeigt. Die Aufbereitung erfolgt mit folgenden Methoden:

- **Minimum Recovery (Min):**
  Berechnet das Minimum über die letzten n Werte.

- **Maximum Recovery (Max):**
  Berechnet das Maximum über die letzten n Werte.

- **Average Recovery (Avg):**
  Berechnet den Durchschnitt über die letzten n Werte.

- **Median Recovery (Med):**
  Berechnet den Median über die letzten n Werte.

- **MedianAverage Recovery (M&A):**
  Sortiert die Wertefolge und berechnet dann den Durchschnitt über eine variabel einstellbare Anzahl von Werten aus der Mitte der sortierten Folge.

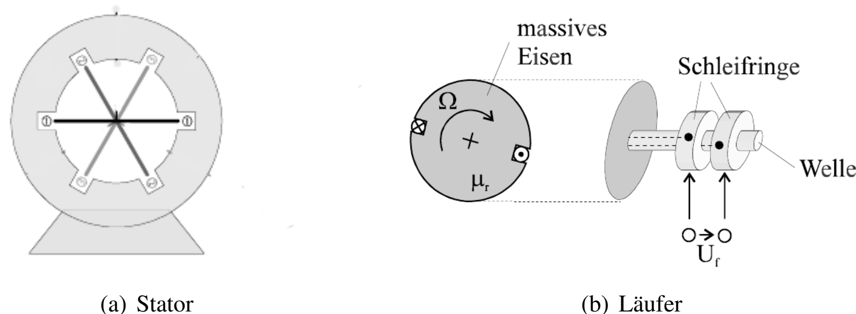
- **Sollwert Recovery (SWR):** Sortiert die Wertefolge nach dem Abstand zum aktuellen Sollwert. Anschließend wird der Durchschnitt über eine variabel einstellbare Anzahl von k Werte mit dem geringsten Abstand zum Sollwert, bestimmt.

Abbildung 2.20: Robust Machine Value GUI

Zusätzlich zu den einzelnen aufbereiteten Messwerten befindet sich an jedem Messwert eine Checkbox, die bei Aktivierung den jeweiligen Messwert, den Sollwert sowie alle aufbereiteten Messwerte in eine Datei mitloggt. Beim erstmaligen aktivieren der Checkbox wird eine neue Datei mit dem Namen des Messwertes sowie einem Zeitstempel erstellt, der das eindeutige Zuordnen von Datei zu Messwert bzw. Messlauf gewährleistet. Die Datei wird im "edu.udo.cs.ls3.MachineControl.gui"-Paket abgelegt.

2.7.4 Abriss über das Verhalten elektrischer Maschinen

Die Synchronmaschine



(a) Stator (b) Läufer

Abbildung 2.21: Aufbau einer Synchronmaschine, aus [19]

Eine Synchronmaschine besteht aus einem feststehenden Stator, der über eine Drehstromwicklung verfügt, d.h. drei verschiedene Spulen, die um 120 ° versetzt an dem Stator angebracht sind (s. Abb. 2.21(a)). Im Stator befindet sich eine Welle, der so genannte *Läufer* (s. Abb. 2.21(b)). Dieser enthält einen Elektromagneten. Wird der Läufer nun in Rotation versetzt, induziert das rotierende Magnetfeld eine Spannung in die Statorwicklungen. Deren Frequenz ist direkt proportional zur Drehfrequenz des Läufers: Läufer und Drehfeld sind *synchron*.

Die Synchronmaschine ist *Inselnetz-fähig*, d.h. sie kann selbstständig eine bestimmte Frequenz und Spannung erzeugen. Im Inselbetrieb ist die Amplitude der Spannung abhängig von der Stärke des Läufer-Magnetfeldes. Diese wiederum wird durch den *Erregerstrom* geregelt, mit dem der Elektromagnet betrieben wird.

Im Gegensatz dazu werden beim Betrieb am *starren Netz*[6] Frequenz und Spannung vom Netz vorgegeben. Man kann jedoch beeinflussen, wieviel Leistung die Synchronmaschine erzeugt. Diese ist abhängig vom Drehmoment des mechanischen Antriebs: je stärker der Antrieb, desto größer die abgegebene Wirkleistung. Außerdem kann die Synchronmaschine Blindleistung erzeugen bzw. aufnehmen. Blindleistung wird von Asynchronmaschinen zur Erzeugung von Wirkleistung benötigt, außerdem können Blindleistungsverluste an den Stromleitungen ausgeglichen werden. Die Erzeugung bzw. Aufnahme von Blindleistung wird über den Erregerstrom gesteuert. Zusätzlich wirkt sich ein hoher Erregerstrom positiv auf die Stabilität der Maschine gegenüber Drehmomentänderungen des Antriebs aus.

Die Synchronmaschine muss vor dem Aufschalten auf ein bestehendes Netz synchronisiert werden, d.h. Frequenz, Spannung und Phasenlage zwischen Netz und Maschine müssen übereinstimmen. Um dies zu erreichen geht man wie folgt vor:

[6] Das Netz heißt starr, weil es so stabil ist, dass eine einzelne Maschine nicht die Möglichkeit hat, seine Parameter merklich zu verändern. Diese Stabilität kommt durch die große Zahl zusammengeschalteter Energieerzeuger zustande, die bereits alle synchron laufen.

1. Zunächst wird die Frequenz der Maschine an die Netzfrequenz angeglichen. Dazu wird der mechanische Antrieb des Läufers so geregelt, dass die gewünschte Frequenz erreicht wird. Man spricht dabei von *Frequenzregelung*.
2. Ist die benötigte Frequenz stabil erreicht, muss die *Klemmspannung*, also die in den Stator induzierte Spannung, an die Netzspannung angeglichen werden. Dies geschieht durch Veränderung des Erregerstroms.
3. Zuletzt muss der Phasenwinkel zwischen Maschine und Netz minimiert werden. Sobald der Winkel ausreichend klein ist, kann die Maschine aufgeschaltet werden. Nun sind Frequenz, Spannung und Phase fest an das Netz gekoppelt. Der Antrieb ist sofort von Frequenzregelung auf *Momentenregelgung* umzustellen, d.h. es wird nicht mehr versucht, den Läufer auf einer bestimmten Frequenz zu halten, sondern er wird mit einer vorgegebenen „Kraft“ dem *Drehmoment* gedreht.

Die Asynchronmaschine

Eine Asynchronmaschine besteht ebenso wie die Synchronmaschine aus einem feststehenden Stator mit Drehstromwicklung. Der Läufer besitzt im Gegensatz zur Synchronmaschine jedoch keinen Magneten, sondern lediglich eine weitere Drehstromwicklung. Daher bewirkt eine Rotationsbewegung des Läufers zunächst gar nichts. Wird an die Statorwicklungen jedoch eine Spannung angelegt, so baut sich ein rotierendes elektromagnetisches Feld auf. Falls sich die Drehzahl des Läufers von der Drehzahl des Magnetfeldes unterscheidet, induziert dieses wiederum einen Strom in die Läuferwicklungen, die ihrerseits ein Magnetfeld aufbauen. Falls nun die Drehzahl des Läufers höher ist als die des Stator-Drehfeldes, so wird Wirkleistung erzeugt.

Festzuhalten ist also, dass eine Asynchronmaschine Blindleistung aus dem Netz benötigt, um den Läufer zu magnetisieren, und Wirkleistung erzeugt, falls die Drehzahl des Läufers hinreichend groß ist. Entspricht die Frequenz des Läufers der des Netzes, nimmt die Maschine weder Wirkleistung auf, noch erzeugt sie welche. Ist die Frequenz des Läufers größer als die Netzfrequenz, geht die Maschine in den Generatorbetrieb über und erzeugt Wirkleistung.

Um eine Asynchronmaschine zu synchronisieren muss also lediglich der mechanische Antrieb so eingestellt werden, dass die Läuferfrequenz der Netzfrequenz entspricht. Dadurch wird sichergestellt, dass im Moment des Zuschaltens keine großen Ströme auftreten. Nach dem Zuschalten sollte der Läufer etwas beschleunigt werden, um Leistung zu erzeugen.

2.8 Generatorsteuerung

2.8.1 Machine Control Protokoll

Das MC-Protokoll ist entwickelt worden, um den Generator kontrollieren und von einem externen Rechner aus hochfahren und steuern zu können. Das MC-Protokoll benutzt dabei für die Kommunikation zwischen Rechner und SPS das Modbus Protokoll, welches von den SPS’en hardwareseitig unterstützt wird. In diesem Abschnitt soll die Implementierung des MC-Protokolls auf den Generator-SPS’en beschrieben werden. Zur Steuerung der Generatoren wird das von Michael Kleemann ent-

wickelte SPS-Programm benutzt [11]. Die Implementierung des MC-Protokolls nutzt die Steuerungskommandos des Generatorprogramms um den Generator Zustand zu verändern und die vom MC-Protokoll vorgegebenen Aktionen durchzuführen.
Weiter oben wurde bereits erwähnt, dass das MC-Protokoll unter anderem den automatischen Anlauf der Generatoren ermöglichen soll. Des Weiteren kann mit Hilfe des MC-Protokolls der Generator manuell, das heißt Schritt für Schritt, hochgefahren und danach auf ein anderes Netz synchronisiert werden. Dabei kann der Benutzer Einfluss auf den Erregerstrom, die Drehzahl sowie die erzeugte Wirk- und Blindleistung des Generators nehmen.
Zu diesem Zweck enthält jede MCP-Nachricht vom Rechner an die Generator-SPS einen Controlcode, der angibt, in welchen Zustand der Generator wechseln soll bzw. welche Parameter verändert werden sollen. Die Nachricht wird von der Generator-SPS empfangen, interpretiert, und anschließend werden je nach geforderter Aktion die nötigen Steuerungskommandos gesetzt. Falls ein Befehl nicht ausgeführt werden kann, weil z.B. das Setzen der geforderten Parameter die Kopplung des Generators zum Netz gefährdet, wird eine Fehlernachricht generiert und dem Rechner mitgeteilt.
In der folgenden Tabelle sind die möglichen Aktionen aufgelistet, die das MC-Protokoll dem Benutzer für die Interaktion mit dem Generator bereitstellt.

MCPCommand	Wert	#Param.	Funktion
SWITCH_OFF	1	0	Kraftwerk ausschalten
STAND_BY	2	0	Kraftwerk in Standby-Modus setzen
ANTRIEB_EIN_DEFAULT	3	0	Antrieb auf Standard-Drehzahl regeln
ANTRIEB_EIN_VALUE	4	1	Antrieb auf beliebige Drehzahl regeln
ERREGERSTROM_EIN	5	1	Erregerstrom einschalten
ERREGERSTROM_AUS	6	0	Erregerstrom abschalten
ZUSCHALTEN	7	0	Aufs Netz aufschalten
TRENNEN	8	0	Vom Netz trennen
SET_TORQUE	9	1	Drehmoment einstellen
NOT_AUS	10	0	Notaus
SPANNUNGSREGELUNG	11	1	In Spannungsregelung wechseln
FREQUENZREGELUNG	12	0	In Frequenzregelung wechseln
MOMENTENREGELUNG	13	0	In Momentenregelung wechseln
SET_LEISTUNG	20	2	Wirk- und Blindleistung einstellen

Tabelle 2.12: MCP Kommandos

SWITCH_OFF

Funktion: Schaltet das Kraftwerk vollständig und kontrolliert aus, d.h. das Kraftwerk wird ggf. zunächst vom Netz getrennt, der Erregerstrom ausgeschaltet und die Drehzahl runtergefahren.
Parameter: keine.

STAND_BY

Funktion: Wie SWITCH_OFF.
Parameter: keine.

ANTRIEB_EIN_DEFAULT

Funktion: Regelt den Antrieb auf die Standard-Drehzahl von 1500 U/min.
Parameter: keine.

ANTRIEB_EIN_VALUE

Funktion: Regelt den Antrieb auf eine beliebige Drehzahl.
Parameter:

1. $Drehzahl_{raw}$: Soll-Drehzahl als Raw-Wert. Die Umrechnung einer realen Drehzahl in den Raw-Wert erfolgt bei beiden Maschinen wie folgt:

$$Drehzahl_{raw} = Drehzahl_{soll}/Drehzahl_{max} \cdot raw_{max} \tag{2.1}$$

mit $Drehzahl_{soll}$ Soll-Drehzahl in U/min, $Drehzahl_{max} = 1610$ U/min, $raw_{max} = 32767$

ERREGERSTROM_EIN

Funktion: Setzt den Sollwert für den Erregerstrom. Ggf. wird der Erregerstrom zunächst eingeschaltet.
Parameter:

1. $IErr_{raw}$: Soll-Erregerstrom als Raw-Wert. Die Umrechnung eines realen Erregerstroms in den Raw-Wert erfolgt bei beiden Maschinen wie folgt:

$$IErr_{raw} = IErr_{soll}/IErr_{max} \cdot raw_{max} \tag{2.2}$$

mit $IErr_{soll}$ Soll-Erregerstrom in A, $IErr_{max} = 1$ A für KW1 bzw. $IErr_{max} = 1.2$ A für KW3, $raw_{max} = 32767$

ERREGERSTROM_AUS

Funktion: Schaltet den Erregerstrom ab.
Parameter: keine.

TRENNEN

Funktion: Trennt das Kraftwerk vom Netz, falls es zugeschaltet ist.
Parameter: keine.

ZUSCHALTEN

Funktion: Schaltet das Kraftwerk aufs Netz auf. Zuvor wird es automatisch synchronisiert.
Parameter: keine.

SET_TORQUE

Funktion: Setzt das gewünschte Drehmoment des Antriebs.
Parameter:

1. $Torque_{raw}$: Soll-Drehmoment als Raw-Wert. Die Umrechnung eines realen Drehmoments in den Raw-Wert erfolgt bei beiden Maschinen wie folgt:

$$Torque_{raw} = Torque_{soll}/Torque_{max} \cdot raw_{max} \tag{2.3}$$

mit $Torque_{soll}$ Soll-Drehmoment in Prozent des maximalen Drehmoments, $Torque_{max} = 100\ \%$, $raw_{max} = 32767$

NOT_AUS

Funktion: Schaltet sofort das Kraftwerk aus, setzt alle Spannungsquellen außer Betrieb.
Parameter: keine.

SPANNUNGSREGELUNG

Funktion: Aktiviert die Spannungsregelung. In diesem Modus kann der Benutzer einen Spannungssollwert vorgeben und die SPS regelt den Erregerstrom entsprechend bis das Kraftwerk die gewünschte Spannung erzeugt.
Parameter:

1. U: Sollwert der Spannung als Raw-Wert (0...32767). Die Interpretation ist maschinenabhängig. Der Wert muss in entsprechenden Java-Klassen vorbereitet werden.
2. U_{raw}: Soll-Spannung als Raw-Wert. Die Umrechnung einer realen Spannung in den Raw-Wert erfolgt bei beiden Maschinen wie folgt:

$$U_{raw} = U_{soll}/U_{max} \cdot raw_{max} \tag{2.4}$$

mit U_{soll} Soll-Spannung in V, U_{max} = 250, $raw_{max} = 32767$

FREQUENZREGELUNG

Funktion: Das Kraftwerk wechselt in die Frequenzregelung. In diesem Modus kann der Benutzer eine Sollfrequenz vorgeben, die das Kraftwerk dann versucht anzufahren.
Parameter: keine.

MOMENTENREGELUNG

Funktion: Das Kraftwerk wechselt in die Momentenregelung. In diesem Modus kann der Benutzer ein Sollmoment vorgeben, welches das Kraftwerk dann versucht anzufahren.
Parameter: keine.

SET_LEISTUNG

Funktion: Setzt die gewünschte Wirk- und Blindleistung.
Parameter:

1. P: Soll-Wirkleistung in Watt ($-2000 \cdot 0.98 < Q < +2000 \cdot 0.98$)
2. Q: Soll-Blindleistung in Watt ($-2000 \cdot 0.98 < Q < +2000 \cdot 0.98$)

2.8.2 MCP Fehlercodes

Da ein Teil der MCP-Kommandos nicht in allen Zuständen der SPS' bzw. des Kraftwerks Sinn macht, es gleichzeitig aber vorteilhaft wäre eine Reaktion auf das Senden eines Kommandos zu bekommen, sendet die SPS Fehlercodes in Form von SSP-Value-Messages.
Ein Beispiel für eine solche Nachricht könnte in der folgenden Situation generiert werden. Die Generator SPS befindet sich im Zustand Standby. In diesem Zustand macht es wenig Sinn eine Nachricht mit dem Kommando TRENNEN an die SPS zu versenden. Sollte dies trotzdem getan werden, generiert die SPS eine SSP-Value-Message, die den fest definierten ValueCode 3 besitzt und zusätzlich einen Parameter, der die Beschreibung des Fehlers enthält. Eine detaillierte Liste alles Fehlercodes mit den dazu gehörigen Fehlerbeschreibungen kann der folgenden Tabelle entnommen werden:

MCP Fehler Code	Fehler Beschreibung
1	Der Antrieb muss erst hoch gefahren werden
2	Das Kraftwerk ist nicht ans Netz angeschlossen
3	Das Kraftwerk muss zunächst synchronisiert werden
21	Kein gültiger Betriebszustand $(P < P_{min})\|\|(P > P_{max})$
22	Kein gültiger Betriebszustand $(E > E_{max})$
23	Kein gültiger Betriebszustand $(\delta < \delta_{min})\|\|(\delta > \delta_{max})$
24	Kein gültiger Betriebszustand $(S > S_{max})$

Tabelle 2.13: Machine Control Protocol - Fehlercodes

Reservierte Register für MCP-Nachrichten

Wie weiter oben bereits erwähnt baut die Kommunikation vom Rechner zur SPS auf dem Modbus Protokoll auf. Der Rechner schreibt den Controlcode sowie die jeweiligen Parameter einer MCP-

Nachricht über Modbus in dafür extra reservierte Register der SPS. Der dafür vorgesehenen Speicherbereich beginnt ab Register `MB 50` und reserviert 2 Bytes für den Controlcode sowie weitere 8 Bytes für die jeweiligen Parameter.

2.8.3 SPS - Implementierung der Steuerung mit MCP

Die Implementierung des MC-Protokolls auf der SPS ist im wesentlichen davon geprägt die MC-Nachrichten zu interpretieren, und dementsprechend die verschiedenen Steuerungskommandos und Sollwertvorgaben anzupassen. Als einfachstes Konzept bietet sich daher die Implementierung als Zustandsautomat an. Die Zustände des Automaten sollen dabei die Zustände des Generators darstellen. Die folgende Aufzählung enthält alle definierten Zustände des Generators:

Bezeichnung	**Wert FB_Zustand**
SPS initialisieren	0
Kraftwerk aus	1
Kraftwerk initialisieren	2
Kraftwerkstillstand	3
Antrieb hochfahren	4
Antrieb hochgefahren	5
Antrieb runtergefahren	6
Kraftwerk abschalten	7
Drehzahl ausregeln	8
Erregung einschalten	9
Erregung abschalten	10
zuschaltbereit	11
synchronisieren	12
Kraftwerk zugeschaltet	13
trennen vom Netz	14

Tabelle 2.14: Generatorzustände der Generatorsteuerung

Wie oben schon erwähnt, steuert die MCP-Implementierung den Generator nur indirekt durch die Steuerungskommandos der Generatorsteuerung. Daher ist es sinnvoll, als Variable für den Zustand des MCP-Zustandsautomaten auch den Generatorzustand der Implementierung der Generatorsteuerung mit zu nutzen. Die Programm-Variable, in der der Generatorzustand der Generatorsteuerung festgehalten wird, ist `FB_Zustand`. Die Generatorsteuerung unterscheidet dabei einige Zustände, die für das MC-Protokoll nur von untergeordneter Bedeutung sind. Deswegen wird der Generatorzustand des MC-Protokoll durch das folgende Mapping aus der Variable `FB_Zustand` gewonnen:

Alle Zustände, die in der obigen Tabelle nicht definiert wurden, sind in der Generatorsteuerung nur Übergangszustände, die die Generatorsteuerung für interne Prozesse verwendet.
Die Entscheidung, die Implementierung des MCP-Zustandsautomaten indirekt auf der Generatorsteuerung aufsetzen zu lassen, bringt einige Vorteile mit sich:

1. Die Zustandsübergänge erfolgen nur indirekt über die Zustandswechsel der Generatorsteuerung. Da ein Zustandswechsel bei der Generatorsteuerung erst dann ausgelöst wird, wenn sich

Wert FB_Zustand	Generatorzustand MC-Protokoll
0	Kraftwerk aus
1	Kraftwerk aus
2	Standby
3	Standby
4	Antrieb hochfahren
5	Antrieb hochfahren
9	Erreger ausregeln
11	Erreger ausregeln
12	Zuschalten
13	Zuschalten

Tabelle 2.15: Mapping Generatorzustand MCP ↔ SPS

der nachfolgende Zustand durch die Messwerte bestätigen lässt, besteht eine enge Bindung zwischen dem MCP-Generatorzustand und dem realen Generatorzustand der jeweiligen Maschine.

2. Externe Ereignisse, die nicht durch das MC-Protokoll ausgelöst worden sind und die trotzdem zu einem Zustandswechsel des Generators führen, werden automatisch vom MC-Protokoll registriert.

3. Das indirekte Auslösen der Zustandswechsel ermöglicht es, dass die Sicherheitsmechanismen der Generatorsteuerung weiterhin bestehen bleiben.

Die folgende Abbildung 2.22 zeigt den konzeptionellen Entwurf des MC-Protokolls als Zustandsdiagramm:

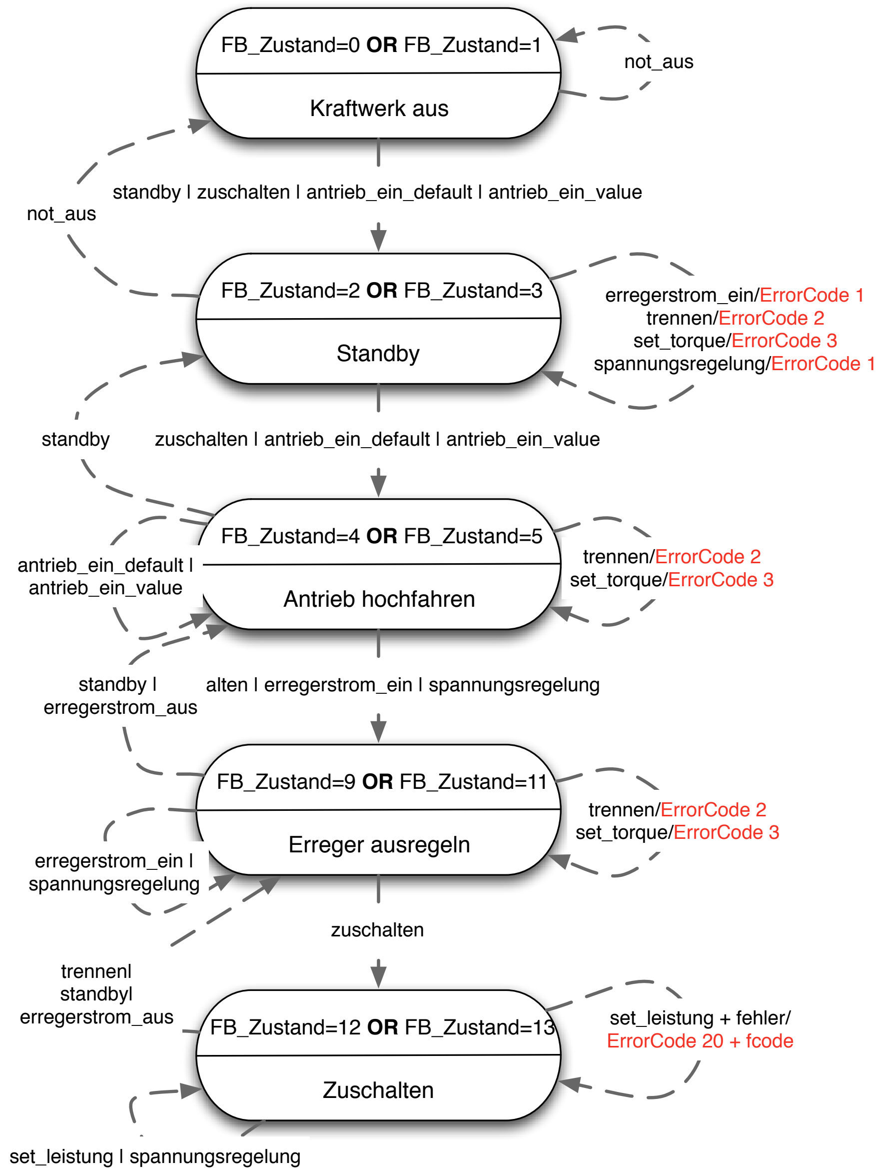


Abbildung 2.22: Zustandsdiagramm des Machine Control Protocol

Dieser Entwurf wurde auf der SPS durch den folgenden Code Abschnitt realisiert:

```
IF ((FB_Zustand = 0) OR (FB_Zustand = 1)) THEN (* "Kraftwerk aus" Modus*)
  DEBUG_state := 0;
  IF ((controlCode = 2) OR (controlCode = 7) OR
     (controlCode = 3)OR (controlCode = 4)) THEN
     (* standby | zuschalten | antrieb fest | antrieb + zahl - Befehl *)
    K_KwEin := TRUE;        (* setze Steuerungskommando *)
    debugControlCode := controlCode;      (* Debug Info *)
  ELSIF (controlCode = 10) THEN (*Notaus Befehl*)
    NotAus := TRUE;        (* Steuerungskommando *)
    debugControlCode := controlCode;
    controlCode := 0;     (* loesche den controlCode *)
```

```
  END_IF
  K_MomentRegelung := FALSE;
ELSIF ((FB_Zustand = 2) OR (FB_Zustand = 3)) THEN (* "Standby" Modus*)
  DEBUG_state := 1;
  IF (controlCode = 10) THEN (*Notaus Befehl*)
    debugControlCode := controlCode;
    IF (FB_Zustand = 2) THEN
      NotAus := TRUE;
      controlCode := 0;
    ELSIF (FB_Zustand = 3) THEN
      NotAus := TRUE;
    END_IF
  END_IF
  IF ((controlCode = 7) OR (controlCode = 3)) THEN (* zuschalt | antrieb fest  - Befehl*)
    FB_DrehzahlSollwert := 28000; (*1500 Umdrehungen *)
    K_AntriebEin := TRUE;
    debugControlCode := controlCode;
  ELSIF (controlCode = 4) THEN    (*"antrieb + zahl - Befehl"*)
    FB_DrehzahlSollwert := controlParameterW1; (* setze Drehzahl auf uebergebenen Wert*)
    K_AntriebEin := TRUE;
    debugControlCode := controlCode;
  ELSIF (controlCode = 5) THEN    (*Erreger + Zahl*)
    MCP_Protokoll(1);       (*Der Antrieb muss erst hochgefahren werden*)
    debugControlCode := controlCode;
    controlCode := 0;
  ELSIF (controlCode = 8) THEN    (*Vom Netz Trennen*)
    MCP_Protokoll(2);       (*Das Kraftwerk ist nicht an das Netz angeschlossen*)
    debugControlCode := controlCode;
    controlCode := 0;       (*loesche controlCode*)
  ELSIF (controlCode = 9) THEN    (* Moment + Zahl *)
    MCP_Protokoll(3);       (*Das Kraftwerk muss zunaechst synchronisiert werden*)
    debugControlCode := controlCode;
    controlCode := 0;       (*loesche controlCode*)
  ELSIF(controlCode = 11) THEN    (*Spannungsregelung*)
    MCP_Protokoll(1);       (*Das Kraftwerk muss zunaechst synchronisiert werden*)
    debugControlCode := controlCode;
    controlCode := 0;       (*loesche controlCode*)
  END_IF
ELSIF ((FB_Zustand = 4) OR (FB_Zustand = 5)) THEN (* "Antieb hochfahr" Modus *)
  DEBUG_state := 2;
  K_PQRegelung := FALSE;
  IF (FB_Zustand = 4) THEN
    IF (controlCode = 5) THEN     (*Erreger + Zahl*)
      MCP_Protokoll(1);    (*Der Antrieb muss erst hochgefahren werden*)
      debugControlCode := controlCode;
      controlCode := 0;    (*loesche controlCode*)
    ELSIF ((controlCode = 7) OR (controlCode = 3)) THEN
    (* zuschalt | antrieb fest  - Befehl*)
      FB_DrehzahlSollwert := 28000;  (*1500 Umdrehungen*)
      K_AntriebEin := TRUE;
      debugControlCode := controlCode;
    ELSIF(controlCode = 11) THEN  (*Spannungsregelung*)
      MCP_Protokoll(1);      (*Das Kraftwerk muss zunaechst synchronisiert werden*)
      debugControlCode := controlCode;
      controlCode := 0;    (*loesche controlCode*)
    END_IF
  END_IF
```

```
  IF (controlCode = 2) THEN       (*standby - Befehl*)
    K_AntriebEin := FALSE;
  ELSIF (controlCode = 10) THEN   (*notaus - Befehl*)
    NotAus := TRUE;
    K_AntriebEin := FALSE; (*verhindert das erneute starten des Gen.*)
  ELSIF (controlCode = 9) THEN    (* Moment + Zahl *)
    MCP_Protokoll(3);       (*Das Kraftwerk muss zunaechst synchronisiert werden*)
  ELSIF (controlCode = 4) THEN  (*"antrieb + zahl - Befehl"*)
    FB_DrehzahlSollwert := controlParameterW1;
  END_IF
  IF (FB_Zustand = 5) THEN
    IF (controlCode = 7) THEN     (* zuschalt - Befehl*)
      K_ErregerEin := TRUE;
      FB_ErregungSollwert := 23000;
    ELSIF (controlCode = 5) THEN  (*Erreger + Zahl - Befehl*)
      FB_ErregungSollwert := controlParameterW1;
      K_ErregerEin := TRUE;
      K_URegelung := FALSE;
    ELSIF (controlCode = 11) THEN (*Spannungsregelung*)
      K_URegelung := TRUE;
      K_ErregerEin := TRUE;
      FB_USollwert := controlParameterW1;
    END_IF
  END_IF
ELSIF ((FB_Zustand = 9) OR (FB_Zustand = 11)) THEN (* "Erreger ausregeln" Modus *)
  DEBUG_state := 3;
  IF (controlCode = 2 ) THEN      (*Standby - Befehl*)
    K_ErregerEin := FALSE;
  ELSIF (controlCode = 10) THEN   (*Notaus - Befehl*)
    NotAus := TRUE;
    K_AntriebEin := FALSE;  (*verhindert das erneute starten des Gen.*)
  ELSIF (controlCode =  7)  THEN  (*Zuschalt - Befehl*)
    K_Zuschalten := TRUE;
  ELSIF(controlCode = 6) THEN     (* ErregerStrom Aus *)
    K_ErregerEin := FALSE;
  ELSIF (controlCode = 5) THEN    (*Erreger + Zahl - Befehl*)
    FB_ErregungSollwert := controlParameterW1;
    K_ErregerEin := TRUE;
    K_URegelung := FALSE;
  ELSIF (controlCode = 11) THEN   (*Spannungsregelung*)
    K_URegelung := TRUE;
    K_ErregerEin := TRUE;
    FB_USollwert := controlParameterW1;
  END_IF
ELSIF ((FB_Zustand = 12) OR (FB_Zustand = 13)) THEN (* "Zuschalten" Modus *)
  DEBUG_state := 4;
  IF (FB_Zustand = 12) THEN
    IF (controlCode =  10)  THEN  (* Notaus *)
      NotAus := TRUE;
      K_AntriebEin := FALSE;  (*verhindert das erneute starten des Gen.*)
      K_KwEin := FALSE;
      K_Zuschalten := FALSE;
      K_ErregerEin := FALSE;
    ELSIF ((controlCode = 2 ) OR (controlCode = 8)) THEN (* Standby / VomNetzTrennen *)
      K_Zuschalten := FALSE;
    ELSIF (controlCode = 5) THEN  (*Erreger + Zahl - Befehl*)
      FB_ErregungSollwert := controlParameterW1;
```

```
      K_ErregerEin := TRUE;
      K_URegelung := FALSE;
    ELSIF (controlCode = 11) THEN (*Spannungsregelung*)
      K_URegelung := TRUE;
      K_ErregerEin := TRUE;
      FB_USollwert := controlParameterW1;
    END_IF
  ELSIF (FB_Zustand = 13) THEN
    IF (controlCode =  10) THEN   (*Not Aus*)
      NotAus := TRUE;
      K_AntriebEin := FALSE;      (*verhindert das erneute starten des Gen.*)
    ELSIF ((controlCode = 2) OR (controlCode = 8)) THEN
      K_Zuschalten := FALSE;
    ELSIF (controlCode = 9) THEN  (* Moment setzen*)
      FB_MomentSollwert := controlParameterW1;
    ELSIF (controlCode = 5) THEN  (*Erreger + Zahl - Befehl*)
      FB_ErregungSollwert := controlParameterW1;
      K_ErregerEin := TRUE;
      K_URegelung := FALSE;
    ELSIF (controlCode = 11)THEN  (*Spannungsregelung*)
      K_URegelung := TRUE;
      K_ErregerEin := TRUE;
      FB_USollwert := controlParameterW1;
    ELSIF (controlCode = 20) THEN (* setLeistung *)
      _error_code :=
         CheckValidStateConfiguration(_Q:=controlParameterR2,_P:=controlParameterR1);
      IF(_error_code = 0) THEN
        P_sollwert := controlParameterR1;
        Q_sollwert := controlParameterR2;
        temp_result := QPToMI(_Q:=Q_sollwert,_P:=P_sollwert); (*Umrechnung QP->MI*)
        FB_MomentSollwert :=  temp_result.soll_moment;
        FB_ErregungSollwert := temp_result.soll_erregung;
        debug_rawErreger := temp_result.soll_erregung;
        debug_rawMoment := temp_result.soll_moment;
      ELSE
        MCP_Protokoll(20 + _error_code);
      END_IF
      debugControlCode := controlCode;
      controlCode := 0;
    END_IF
  END_IF
  IF (controlCode = 6) THEN
    MCP_Protokoll(4);        (* Gen. zugeschaltet -> !erreger aus*)
  ELSIF ((controlCode = 3)OR (controlCode = 4)) THEN
    MCP_Protokoll(5);        (* Gen. zugeschaltet -> drehzahl fest*)
  END_IF
END_IF
```

2.8.4 Implementierung in Java

Den Kern der Java-Implementierung des MC-Protokolls bildet die Klasse `SMController`. Diese Klasse sendet die MCP-Commands über Modbus an die SPSen. Sie macht Gebrauch von der Enumeration `MCPCommands`, welche alle definierten MCP-Commands mit den entsprechenden Integer-Werten enthält. Für jeden möglichen Befehl implementiert `SMController` eine Funktion, welche über Modbus das MCP-Command und eventuell erforderliche Parameter in ein Register auf der SPS

schreibt.

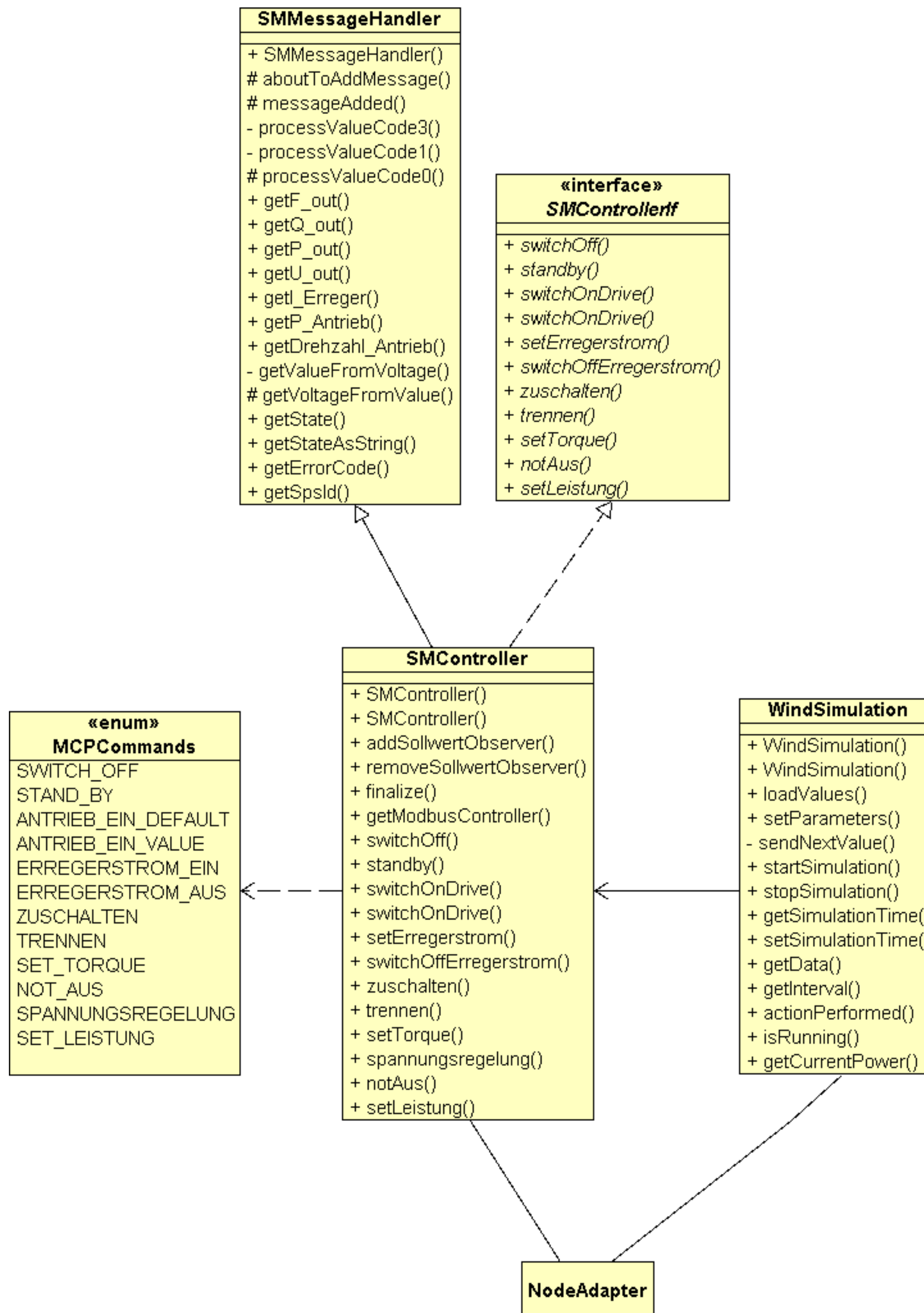


Abbildung 2.23: Klassendiagramm MCP

Funktionen, die physikalische Werte erwarten, wie bspw. `setErregerstrom(float i)`, erwarten als Eingangsparameter den gewünschten Wert als Fließkommazahl. Da die SPS aber bei einigen Parametern den Wertebereich durch einen WORD approximiert müssen die Fließkommazahlen zunächst umgerechnet werden.

Für die Simulation von Windkraftanlagen besteht die Möglichkeit, ein Profil mit Zeit-Wirkleistungs-Paaren aus einer csv-Datei einzulesen und dieses Profil beliebig gerafft abzufahren. Dazu dient die Klasse `WindSimulation`. Über die Funktion `loadValues()` wird eine Datei eingelesen. Diese enthält per definitionem Werte für ein ganzes Jahr. Die Leistungswerte in der Datei sind so normiert, dass die maximale Leistung dem Wert 1.0 entspricht. `WindSimulation` berechnet daraus automatisch die Anzahl Werte pro Tag. Über die Funktion `setParameters()` kann das zu simulierende Zeitintervall (bspw. Tag 348 bis 350), die Simulationszeit (bspw. 30 Minuten) sowie ein Leistungsfaktor, mit dem die Werte aus der Datei multipliziert werden, eingestellt werden. Sobald das Profil mit `startSimulation()` gestartet wurde, sendet `WindSimulation` die aktuellen Leistungswerte an einen `SMController`.

Über die Klasse `NodeAdapter` besteht schließlich die Möglichkeit, über UPnP Werte an den `SMController` zu übermitteln und aktuelle Werte abzufragen.

2.9 Netzregelung

2.9.1 Konzept

Die hardwareseitige Netzregelung ist dazu gedacht, dass Werte für Erregerstrom, Wirkleistung und Bindleistung vorgegeben werden können. Mit Hilfe der Netzregelung sollen die Vorgaben des Verhandlungsalgorithmus mit der Einheit Leistung umgesetzt werden, damit die zur Verfügung stehende Leistung geregelt werden kann.

2.9.2 Netzregelung

Die Antriebssteuerung wird erst während des Hochfahrens der Maschine, nämlich wenn diese sich im Zustand `Kraftwerksstillstand` (FB_Zustand 3) befindet, freigeschaltet. Zu diesem Zeitpunkt wird die Drehzahlregelung eingeschaltet, um den Antrieb hochzufahren und das Kraftwerk zuzuschalten.

Während der Verhandlungen werden Wirkleistung und Blindleistung gesetzt, die Regelung setzt diese Vorgaben um. Bei der Asynchronmaschine kann nur auf die Wirkleistung Einfluss genommen werden, während bei den Synchronmaschinen auch die Blindleistung beeinflusst werden kann.

Die eigentliche Ansteuerung der Maschinen übernimmt ein Steuerungsbaustein, die so genannte Simoreg Antriebssteuerung. Genaueres dazu ist in Kapitel 2.3 zu finden.

2.9.3 Spannungsregelung

Die Spannungsregelung dient in Energieversorgungsnetzen der Anpassung der Netzspannung auf die aktuelle Lastsituation im Netz. Dabei werden Stellbefehle vom Spannungsregler zum Transformator gegeben, d.h. in diesem Aufbau werden die Vorgaben vom DEZENT Algorithmus gemacht. Der Stufenschalter für Leistungstransformatoren schaltet dann ein neues Windungsverhältnis. Aus diesem

ergibt sich die resultierende neue Netzspannung. In unserem Fall wird die Spannung direkt an den Synchronmaschinen durch Änderung des Erregerstroms gesteuert.
Der zulässige Regelbereich für Umspanner (Netzspannung) der öffentlichen Energieversorgung ist in der IEC-Norm 60150 angegeben. Die Spannungsregelung wird meist zur Synchronisation des Kraftwerks mit dem Netz oder eines anderen Kraftwerks genutzt. Diese kann aber auch in den folgenden Zuständen mit dem controlCode 11 eingeschaltet werden.

- FB_Zustand 2: Kraftwerk initialisieren
- FB_Zustand 3: Kraftwerk Stillstand
- FB_Zustand 5: Antrieb hochgefahren
- FB_Zustand 9: Erregung einschalten
- FB_Zustand 11: Zuschaltbereit
- FB_Zustand 12: Synchronisieren
- FB_Zustand 13: Kraftwerk zugeschaltet

2.9.4 Momentenregelung

Im Fall der Momentenregelung ist das Drehmoment die Regelgrösse. Die Drehzahl stellt sich dabei aufgrund der Lastkennlinie ein.
Wenn das Kraftwerk zugeschaltet ist, wird umgeschaltet von Drehzahlregelung zur Momentenregelung. Sollte keine Momentenregelung gemacht werden, dann bleibt es bei der Drehzahlregelung. Wird das Kraftwerk heruntergefahren, so wird auch die Momentenregelung ausgeschaltet.

Das vom Verhandlungsalgorithmus vorgegebene Moment muss zunächst, wie im Codesegment zu sehen, in den Sollwert für die Maschine umgerechnet werden, ehe dieser an die Antriebssteuerung weitergegeben wird.

```
1  const_X_d := 15;
2  const_a := 199;
3  const_b := 50;
4
5  (*Umrechnung von vorgegebenes Moment in soll-Moment *)
6
7  (*Das Moment wird in Prozent von 0 bis 100 angegeben  *)
8  QPToMI.soll_moment :=REAL_TO_WORD(((_P + 280.0)/2700.0) *32768)  ; (*  (((_P + 310.0)/27/100.0) *32768)  *)
9
10
11 _U := (FB_U /32768.0) * 250.0 * 1.002;
12 _Qk := 2500.0*234*234/_U/_U;
13
14 _iErreg := 0.82 / (COS(ATAN(_P/p_bezug))) *(1 + (_Q-120.0)*1.176/_Qk) ;
15
16 IF (NOT((_iErreg > 1.0) OR (_iErreg < 0.4))) THEN
17         QPToMI.soll_erregung := REAL_TO_WORD( (_iErreg / 1.0)*32768); (* ierreg *)
18 END_IF
```

Da die vom Algorithmus vorgegebenen Werte Wirkleistung und Blindleistung sind, muss mit diesen vorgegebenen Werten ein Moment errechnet werden, das durch die Abhängigkeiten der Werte die geforderten Größen ergibt. Dies wird in folgendem Codesegment realisiert.

```
1  IF (K_PQRegelung) THEN
2          pRegler.SET_POINT :=  P_sollwert;
3          pRegler(ACTUAL :=  ((CAST_WORD(FB_P)/32678.0) * 1739.0), KP := pReglerKP,TN := pReglerTN,MANUAL := FALSE);
4          qRegler.SET_POINT := Q_sollwert;
5          qRegler(ACTUAL := ((CAST_WORD(FB_Q)/32678.0) *1739.0), KP := qReglerKP,TN := qReglerTN,MANUAL := FALSE);
```

```
6         IF (NOT(PQ_ReglerReset)) THEN
7                 regelVerzoegerung(IN := TRUE);
8                 IF (regelVerzoegerung.Q) THEN
9                         IF (reglerInit) THEN
10                                pRegler.Y_MANUAL := P_sollwert;
11                                qRegler.Y_MANUAL := Q_sollwert;
12                                IF (pRegler.RESET) THEN
13                                        pRegler(RESET := FALSE, MANUAL:=TRUE);
14                                        qRegler(RESET := FALSE, MANUAL:=TRUE);
15                                        reglerInit := FALSE;
16                                ELSE
17                                        pRegler(RESET:=TRUE);
18                                        qRegler(RESET:=TRUE);
19                                END_IF
20                        ELSE
21                                temp_result := QPToMI(_Q:=qRegler.Y , _P:=pRegler.Y);
22                                FB_MomentSollwert := temp_result.soll_moment;
23                                FB_ErregungSollwert := temp_result.soll_erregung;
24                        END_IF
25                END_IF
26        ELSE
27                PQ_ReglerReset := FALSE;
28                reglerInit := TRUE;
29                regelVerzoegerung(IN := FALSE);
30        END_IF
31 ELSE
32        regelVerzoegerung(IN := FALSE);
33 END_IF
```

2.10 Digitales Auslesen des SINEAX Multimeters

Die Messwerte, wie sie bisher ausgelesen wurden, waren sehr verrauscht und daher für die Verhandlung recht ungeeignet. Als Ursache für die starke Verrauschung der Messwerte wurde das Umwandeln der Messwerte in ein Analogsignal sowie das spätere digitalisieren dieses Signals festgemacht. Daher wurde beschlossen den Messumformer Sineax DME442 direkt digital auszulesen. Zu diesem Zweck verfügt der Messumformer über eine RS-232 Schnittstelle die

2.10.1 Pinbelegung

Der Sineax Messumformer benutzt für die Datenübertragung ein einfaches asynchrones serielles Verfahren. Seriell bedeutet, dass die einzelnen Bits des zu übertragenden Bytes nacheinander über eine einzige Datenleitung geschoben werden. Asynchron heißt, dass es keine Taktleitung gibt, die dem Datenempfänger genau sagt, wann das nächste Bit auf der Datenleitung liegt. So ein Verfahren kann nur funktionierten, wenn Sender und Empfänger mit genau dem gleichen internen Takt arbeiten, und wenn der Empfänger gesagt bekommt, wann das erste Bit genau anfängt, also durch Synchronisation.

Alle RS232-Leitungen (mit Ausnahme der Masseleitung) arbeiten mit den Spannungspegeln +12V (für eine logische '0') und -12V (für eine logische '1'). (Erlaubt sind jeweils 5V..15V.) Der Datenempfänger erwartet eine Spannung von über +3V für eine 0 und von unter -3V für eine 1.
Um Daten minimalistisch von einer Datenquelle zu einem Datenempfänger zu übertragen, werden eigentlich nur 2 Drähte benötigt - eine Masseleitung und eine Datenleitung. Für eine bidirektionale Verbindung reichen also 3 Leitungen.

Zur Übertragung von der SPS zum Sineax Messumformer wird ein serielles Kabel benutzt. Hierbei wurden die Pins wie folgt belegt.

Die Daten-Eingangsleitung des Senders wird als **TxD in** bezeichnet, die Daten-Ausgangsleitung dementsprechend als **TxD out**. Die Daten-Ausgangsleitung des Empfängers wird als **RxD out** und die Daten-Eingangsleitung analog als **RxD in** bezeichnet. Die Belegung der Pins ist Abb. 2.24 zu entnehmen. Zusätzlich mussten die Leitungen 4, 6 und 8 im seriellen Kabel zusammengelötet werden, damit die korrekte Übertragung gewährleistet ist.

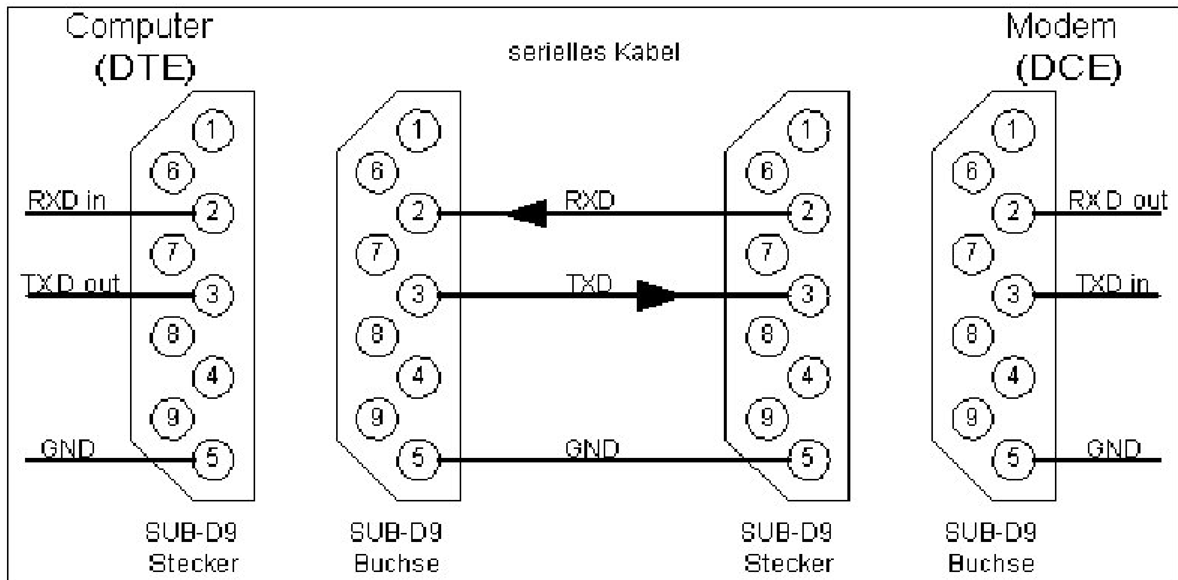


Abbildung 2.24: RS232-Kabel

SPS-Programmierung

Die Steuerung durch die SPS bereitete jedoch erhebliche Schwierigkeiten, da die SPS öfters abstürzte.
Später stellte sich heraus, dass dafür die offiziell zur Verfügung gestellte Bibliothek `Serial_Interface_01.lib` verantwortlich war.
Dieses Problem wurde behoben, indem die offizielle Bibliothek durch eine selbstgeschriebene ersetzt wurde.
Die vorgesehenen Übertragungsprotokolle sind in der Schnittstellen-Definition des Sineax DME442 aufgeführt. Für die Übertragungssicherheit wird das CRC8-Protokoll genutzt.

```
1
2  crc := 0;
3
4  FOR i := 0 TO dataLength-1 DO
5          IF (i <> 0) THEN
6                  IF (ptData = endPointer) THEN
7                          ptData := startPointer;
8                  ELSE
9                          ptData := ptData +1;
10                 END_IF
11         END_IF
12         data := ptData^;
13
14         Reg8 := crc;
15         Reg8 := Reg8 XOR data;
16
17         FOR j := 0 TO 7 DO
18                 LSB := Reg8 AND 1;
19                 Reg8 := Reg8 / 2;
```

```
20                  IF  (LSB<>0) THEN
21                          Reg8  := Reg8 XOR Poly8;
22                  END_IF
23          END_FOR
24          crc  := Reg8;
25
26
27  END_FOR
28
29  CRC8:=crc;
```

Der Messumformer kann in verschiedene Modi versetzt werden, um spezielle Aufgaben auszuführen. Die verschiedenen Modi sind:

- Normalbetrieb
- Abgleichmodus
- Kundenabgleich
- Simulation
- Systemcheck
- Multimeter

Zur Messwertausgabe können nur die drei Modi Simulation, Systemcheck und Multimeter genutzt werden. Im Modus Simulation werden Analog- und Digitalausgänge simuliert. Im Modus Systemcheck werden die Werte, aller für die aktuelle Betriebsart möglichen Messgrössen, periodisch jeweils nach Abschluss von 3 Messdurchgängen über die serielle Schnittstelle gesendet. Hierbei sind die gelesenen Grössen auf Nennwerte bezogen, d.h. es muss ein Nennstrom und eine Nennspannung gegeben sein. Im Multimeter Modus werden die Werte aller für die aktuelle Betriebsart möglichen Messgrössen gemessen. Es müssen keine Messbereiche programmiert werden und auch Nennwerte sind nur in für uns nicht relevanten Sonderfällen notwendig. Um die Messauflösung optimal auszunutzen, beziehen sich die übertragenen Raw-Daten jedoch, je nach gemessenem Wert, auf unterschiedliche Bezugsgrößen. Diese Bezugsgrößen werden stets mit den Messwerten übertragen.
Daher schien der Multimeter Modus zum Auslesen der geeignetste. Die ausgelesen Werte waren auch nicht verrauscht, allerdings wurde die Bezugsgröße fehlerhaft übertragen, so dass die umgerechneten Werte unbrauchbar wurden.
Auch der Systemcheck Modus war keine Alternative, da es nicht möglich ist Bezugswerte zu bestimmen. Daher mussten wir weiterhin auf die Analogwerte zurückgreifen. Der Vollständigkeit halber seien dennoch im Folgenden Details zum Nachrichtenaustausch mit dem Messumformer dargestellt.

Um die im Multimeter Modus empfangenen Werte umzurechnen, wurde folgender Code benutzt. Die Umrechnung wurde aus der Schnittstellendefinition Sineax DME4 Abschnitt 6.7 entnommen.

```
1   dmeData := multiMsg.data^;
2
3   ir[0]:= 0;
4   ir[1]:= 1;
5   ir[2]:= 2;
6   ir[3]:= 5;
7
8   ib[0]:= 0.5;
9   ib[1]:= 1;
10  ib[2]:= 2;
11  ib[3]:= 5;
12  ib[4]:= 10;
13
14  ub[0]:= 50;
15  ub[1]:= 100;
16  ub[2]:= 200;
17  ub[3]:= 500;
18  ub[4]:= 1000;
19
```

```
20 rangeUI := dmeData.Data[ (multiMsg.start_index + indexShift + 2) MOD 256];
21 anschlArt := dmeData.Data[ (multiMsg.start_index + indexShift + 1) MOD 256];
22
23
24 ir_index := rangeUI / 64;
25 ib_index := (rangeUI MOD 64) / 8;
26 ub_index := rangeUI MOD 8;
27
28 u13raw := dmeData.Data[ (multiMsg.start_index +indexShift + 23) MOD 256]*256 \
29         + dmeData.Data[ (multiMsg.start_index +indexShift + 24) MOD 256];
30 u13 := CAST_WORD(u13raw) /10000 * ub[ub_index] ;
31
32
33 uNetzRaw := dmeData.Data[ (multiMsg.start_index +indexShift + 11) MOD 256]*256 \
34         + dmeData.Data[ (multiMsg.start_index +indexShift + 12) MOD 256];
35 uNetz := CAST_WORD(uNetzRaw) /10000 * ub[ub_index] ;
36
37 u1NRaw := dmeData.Data[ (multiMsg.start_index +indexShift + 13) MOD 256]*256 \
38         + dmeData.Data[ (multiMsg.start_index +indexShift + 14) MOD 256];
39 u1N := (CAST_WORD(u1NRaw) /10000 * ub[ub_index])/1.732 ;
40
41 i1Raw := dmeData.Data[ (multiMsg.start_index +indexShift + 27) MOD 256]*256 \
42         + dmeData.Data[ (multiMsg.start_index +indexShift + 28) MOD 256];
43 i1 := CAST_WORD(i1Raw) /10000 * ib[ib_index] ;
44
45 ib1Raw := dmeData.Data[ (multiMsg.start_index +indexShift + 89) MOD 256]*256 \
46         + dmeData.Data[ (multiMsg.start_index +indexShift + 90) MOD 256];
47 ib1 := CAST_WORD(ib1Raw) /10000 * ir[ir_index] ;
48
49
50 iNetzRaw := dmeData.Data[ (multiMsg.start_index +indexShift + 25) MOD 256]*256 \
51         + dmeData.Data[ (multiMsg.start_index +indexShift + 26) MOD 256];
52 iNetz := CAST_WORD(iNetzRaw) /10000 * ib[ib_index];
53
54 p1Raw := dmeData.Data[ (multiMsg.start_index +indexShift + 35) MOD 256]*256 \
55         + dmeData.Data[ (multiMsg.start_index +indexShift + 36) MOD 256];
56 p1 := CAST_WORD(p1Raw) /10000 * ub[ub_index]*ib[ib_index] ;
57
58 p2Raw := dmeData.Data[ (multiMsg.start_index +indexShift + 37) MOD 256]*256 \
59         + dmeData.Data[ (multiMsg.start_index +indexShift + 38) MOD 256];
60 p2 := CAST_WORD(p2Raw) /10000 * ub[ub_index]*ib[ib_index] ;
61
62
63 p3Raw := dmeData.Data[ (multiMsg.start_index +indexShift + 39) MOD 256]*256 \
64         + dmeData.Data[ (multiMsg.start_index +indexShift + 40) MOD 256];
65 p3 := CAST_WORD(p3Raw) /10000 * ub[ub_index]*ib[ib_index] ;
66
67
68 pNetz := (p1 + p2 + p3)*10.0;
69
70 q1Raw := dmeData.Data[ (multiMsg.start_index +indexShift + 43) MOD 256]*256 \
71         + dmeData.Data[ (multiMsg.start_index +indexShift + 44) MOD 256];
72 q1 := CAST_WORD(q1Raw) /10000 * ub[ub_index]*ib[ib_index] ;
73
74 q2Raw := dmeData.Data[ (multiMsg.start_index +indexShift + 45) MOD 256]*256 \
75         + dmeData.Data[ (multiMsg.start_index +indexShift + 46) MOD 256];
76 q2 := CAST_WORD(q2Raw) /10000 * ub[ub_index]*ib[ib_index] ;
77
78
79 q3Raw := dmeData.Data[ (multiMsg.start_index +indexShift + 47) MOD 256]*256 \
80         + dmeData.Data[ (multiMsg.start_index +indexShift + 48) MOD 256];
81 q3 := CAST_WORD(q3Raw) /10000 * ub[ub_index]*ib[ib_index] ;
82
83 qNetz := (q1+q2+q3)*10.0;
84
85 fRaw := dmeData.Data[ (multiMsg.start_index +indexShift + 65) MOD 256]*256 \
86         + dmeData.Data[ (multiMsg.start_index +indexShift + 66) MOD 256];
87 f := fRaw/1000;
```

Zunächst initialisiert die SPS eine Verbindung zum Messumwandler. Danach können Nachrichten gesendet werden. Als erstes muss geprüft werden in welchem Modus sich der Messumwandler befindet und diesen, falls notwendig, in den passenden Modus setzen. Daraufhin können die eigentlichen Nachrichten (zum Empfangen und Anfragen der Messwerte) gesendet und empfangen werden. Empfangene Nachrichten werden zunächst in einen Ring Buffer geschrieben und erst von dort ausgelesen. Bei jeder empfangenen Nachricht wird das CRC Bit kontrolliert und danach die Nachricht gemäß des Übertragungsprotokolls ausgewertet. Ebenso muss für jede Nachricht, die gesendet werden soll, ein CRC Bit berechnet werden.

```
1 IF (oInit) THEN
2         IF (init_state = 0) THEN
3                 timer.PT := t#500ms;
4                 timer(IN := TRUE);
```

```
5                 outInterface.EN := FALSE;
6                 IF (NOT(rs232_init)) THEN (*initialisiere Verbindung*)
7                         rs232_init := TRUE;
8                         rs232_req := FALSE;
9                         rs232_ack := FALSE;
10                ELSE
11                        IF ((rs232_initAck)AND(timer.Q)) THEN (*initialisierung beendet*)
12                                rs232_init := FALSE;
13                                timer(IN := FALSE);
14                                init_state := 1;  (*wechsel in den naechsten Schritt*)
15                        END_IF
16                END_IF
17                RETURN; (*vor dem ersten Verbindungsreset soll nicht gesendet bzw. empfangen werden*)
18        ELSIF(init_state = 1) THEN (*schicke in zustand 0 wechseln nachricht*)
19                IF(outInterface.ENO) THEN
20                        outInterface.EN := FALSE;
21                ELSE
22                        dmeOutData[0] := 68; (* 'D' *)
23                        dmeOutData[1] := 0;
24                        dmeOutData[2] := 7;
25                        dmeOutData[3] := 16;
26                        dmeOutData[4] := 47;
27                        dmeOutData[5] := DME442_Modi(TRUE,FALSE,0); (*in modus normalbetrieb wechseln mit reset*)
28                        dmeOutData[6] := CRC8(ADR(dmeOutData),ADR(dmeOutData),ADR(dmeOutData[32]),6);
29                        outInterface.Data := ADR(dmeOutData);
30                        outInterface.nBytes := 7;
31                        outInterface.EN := TRUE;
32                        init_state := 2;
33                        last_recMsgCount := oRecMsgCount;
34                        timer.PT := t#8s;
35                END_IF
36        ELSIF(init_state = 2) THEN
37                timer(IN := TRUE);
38                IF (timer.Q) THEN
39                        timer(IN := FALSE);
40                        IF ((oRecMsgCount < last_recMsgCount)  OR (oRecMsgCount-last_recMsgCount > 2) OR (oDMEMsg.length <> 4)) THEN
41                                (*modus wechseln hat noch nicht geklappt*)
42                                init_state := 0;
43                        ELSE
44                                timer.PT := t#2s;
45                                init_state := 3;
46                        END_IF
47                END_IF
48        ELSIF(init_state = 3) THEN
49                timer(IN := TRUE);
50                IF (timer.Q) THEN
51                        oRecMsgCount := 0;
52                        last_recMsgCount := 0;
53                        oInit := FALSE;
54                END_IF
55        END_IF
56 END_IF
57
58 IF (NOT(oInit)) THEN
59 IF (NOT(outInterface.ENO)) THEN
60        IF (ioAction = SET_DME_MODUS) THEN
61                dmeOutData[0] := 68; (* 'D' *)
62                dmeOutData[1] := 0;
63                dmeOutData[2] := 7;
64                dmeOutData[3] := 16;
65                dmeOutData[4] := 47;
66                dmeOutData[5] := iDME_Modus;
67                dmeOutData[6] := CRC8(ADR(dmeOutData),ADR(dmeOutData),ADR(dmeOutData[32]),6);
68                outInterface.Data := ADR(dmeOutData);
69                outInterface.nBytes := 7;
70                outInterface.EN := TRUE;
71        ELSIF (ioAction = GET_REGISTERS) THEN
72                dmeOutData[0] := 82; (* 'R' *)
73                dmeOutData[1] := 0;
74                dmeOutData[2] := 8;
75                dmeOutData[3] := WORD_TO_BYTE(iStartAd / 256);
76                dmeOutData[4] := WORD_TO_BYTE(iStartAd MOD 256);
77                dmeOutData[5] := WORD_TO_BYTE(iEndAd / 256);
78                dmeOutData[6] := WORD_TO_BYTE(iEndAd MOD 256);
79                dmeOutData[7] := CRC8(ADR(dmeOutData),ADR(dmeOutData),ADR(dmeOutData[32]),7);
80                outInterface.Data := ADR(dmeOutData);
81                outInterface.nBytes := 8;
82                outInterface.EN := TRUE;
83        ELSIF (ioAction = 42) THEN
84                dmeOutData[0] := 65;
85                dmeOutData[1] := 84;
86                dmeOutData[2] := 68;
87                dmeOutData[3] := 84;
88                dmeOutData[4] := 48;
89                dmeOutData[5] := 49;
90                dmeOutData[6] := 50;
91                dmeOutData[7] := 13;
92                outInterface.Data := ADR(dmeOutData);
93                outInterface.nBytes := 8;
94                outInterface.EN := TRUE;
```

```
95          ELSIF (ioAction = SET_REGISTERS) THEN
96                  dmeOutData[0] := 68; (* 'D' *)
97                  dmeOutData[1] := WORD_TO_BYTE((iDataLength+6) / 256);
98                  dmeOutData[2] := WORD_TO_BYTE((iDataLength+6) MOD 256);
99                  dmeOutData[3] := WORD_TO_BYTE(iStartAd / 256);
100                 dmeOutData[4] := WORD_TO_BYTE(iStartAd MOD 256);
101                 FOR i := 0 TO iDataLength-1 DO
102                         dmeOutData[i+5] := iData^;
103                         iData := iData + 1;
104                 END_FOR
105                 dmeOutData[iDataLength+5] := CRC8(ADR(dmeOutData),ADR(dmeOutData),ADR(dmeOutData[32]) ,iDataLength+5);
106                 outInterface.Data := ADR(dmeOutData);
107                 outInterface.nBytes := iDataLength+6;
108                 outInterface.EN := TRUE;
109         END_IF
110         ioAction := 0;
111 ELSE (*ENO TRUE*)
112         outInterface.EN := FALSE;
113 END_IF
114 END_IF
115 outInterface();
116 inInterface(buffer:=dmeInBuffer);
117 oSending := outInterface.EN;
118
119  (*Werte Empfangen*)
120 messageWaiting := TRUE;
121 WHILE (messageWaiting) DO
122         IF (dmeInBuffer.Index <> recDMEMsg.end_index) THEN
123                 IF (dmeInBuffer.Index > recDMEMsg.end_index) THEN
124                         ring_dif :=     INT_TO_BYTE(dmeInBuffer.Index -recDMEMsg.end_index);
125                 ELSE
126                         ring_dif := INT_TO_BYTE(256 - recDMEMsg.end_index + dmeInBuffer.Index);
127                 END_IF
128                 IF (recDMEMsg.length <> 0) THEN (*akt Message receiving*)
129                         IF (recDMEMsg.length-recMsgDataCount <= ring_dif) THEN (*MSG ready*)
130                                 recDMEMsg.end_index := WORD_TO_BYTE((recDMEMsg.start_index + recDMEMsg.length) MOD 256);
131                                 IF (recDMEMsg.end_index=0) THEN
132                                         crc_rec := dmeInBuffer.Data[255];
133                                 ELSE
134                                         crc_rec := dmeInBuffer.Data[recDMEMsg.end_index -1];
135                                 END_IF
136                                 crc_comp := CRC8(ptData:=ADR(dmeInBuffer.Data[recDMEMsg.start_index]),startPointer:= \
137                                         ADR(dmeInBuffer.Data),endPointer:=ADR(dmeInBuffer.Data[255]),dataLength:=recDMEMsg.length -1);
138                                 IF (crc_rec = crc_comp)THEN
139                                         oDMEMsg.data := ADR(dmeInBuffer);
140                                         oDMEMsg.start_index := recDMEMsg.start_index;
141                                         oDMEMsg.end_index := recDMEMsg.end_index;
142                                         oDMEMsg.length := recDMEMsg.length;
143                                         recDMEMsg.length := 0;
144                                         recDMEMsg.start_index := oDMEMsg.end_index;
145                                         recMsgDataCount := 0;
146                                         recDMEMsg.end_index := recDMEMsg.start_index;
147                                         oRecMsgCount := DWORD_TO_WORD((oRecMsgCount + 1) MOD 65536);
148                                 ELSE
149                                         recDMEMsg.length := 0;
150                                         recDMEMsg.start_index := recDMEMsg.end_index;
151                                         recMsgDataCount := 0;
152                                 END_IF
153                         ELSE (*Receiving Msg still not ready*)
154                                 recMsgDataCount := ring_dif + recMsgDataCount;
155                                 recDMEMsg.end_index := WORD_TO_BYTE((recDMEMsg.end_index + ring_dif) MOD 256);
156                                 messageWaiting := FALSE;
157                         END_IF
158                 ELSE
159                         IF (ring_dif > 2) THEN
160                                 recDMEMsg.length := dmeInBuffer.Data[(1+recDMEMsg.start_index)MOD 256]*256 \
161                                         + dmeInBuffer.Data[(2+recDMEMsg.start_index)MOD 256];
162                                 IF (recDMEMsg.length-recMsgDataCount <= ring_dif) THEN (*MSG ready*)
163                                 recDMEMsg.end_index := WORD_TO_BYTE((recDMEMsg.start_index + recDMEMsg.length) MOD 256);
164                                 IF (recDMEMsg.end_index=0) THEN
165                                         crc_rec := dmeInBuffer.Data[255];
166                                 ELSE
167                                         crc_rec := dmeInBuffer.Data[recDMEMsg.end_index -1];
168                                 END_IF
169                                 crc_comp := CRC8( \
170                                         ptData:=ADR(dmeInBuffer.Data[recDMEMsg.start_index]), \
171                                         startPointer:=ADR(dmeInBuffer.Data), \
172                                         endPointer:=ADR(dmeInBuffer.Data[255]), \
173                                         dataLength:= recDMEMsg.length-1 \
174                                 );
175                                 IF (crc_rec = crc_comp)THEN
176                                         oDMEMsg.data := ADR(dmeInBuffer);
177                                         oDMEMsg.start_index := recDMEMsg.start_index;
178                                         oDMEMsg.end_index := recDMEMsg.end_index;
179                                         oDMEMsg.length := recDMEMsg.length;
180                                         recDMEMsg.length := 0;
181                                         recDMEMsg.start_index := oDMEMsg.end_index;
182                                         recMsgDataCount := 0;
183                                         recDMEMsg.end_index := recDMEMsg.start_index;
184                                         oRecMsgCount := DWORD_TO_WORD((oRecMsgCount + 1) MOD 65536);
```

```
185                                     ELSE
186                                             recDMEMsg.length := 0;
187                                             recDMEMsg.start_index := recDMEMsg.end_index;
188                                             recMsgDataCount := 0;
189                                     END_IF
190                             ELSE (*Receiving Msg still not ready*)
191                                     recMsgDataCount := ring_dif + recMsgDataCount;
192                                     recDMEMsg.end_index := WORD_TO_BYTE((recDMEMsg.end_index + ring_dif) MOD 256);
193                                     messageWaiting := FALSE;
194                             END_IF
195                             ELSE
196                                     messageWaiting := FALSE; (*Es sind noch nicht genug Daten vorhanden*)
197                             END_IF
198                     END_IF
199             ELSE
200                     messageWaiting := FALSE;
201             END_IF
202 END_WHILE
203
204
205 (*Ausgabe Variablen Aktualisieren*)
206 oSending   :=  NOT(outInterface.ENO);
```

2.11 Photovoltaik-Anlage

2.11.1 Einleitung

Neben den Synchronmaschinen und der Asynchronmaschine steht uns als weitere Energiequelle die Hardware-Simulation einer Photovoltaikanlage zur Verfügung.

Zuerst eine kurze Einführung zu Photovoltaikanlagen.

Eine Photovoltaikanlage ist eine Solaranlage. Mit Hilfe von Solarzellen wird ein Teil der Sonnenstrahlung in elektrische Energie umgewandelt. Die Umwandlung solcher Energie, wird mit Photovoltaik bezeichnet. Im Gegensatz dazu gibt es noch weitere Umwandlungsformen wie Wärmeenergie in mechanische Energie.

Die Photovoltaikanlage besteht aus mehreren Komponenten. Zum einen gibt es die sichtbaren Solarzellen, die auf Dächern montiert sind oder frei im Gelände aufgestellt werden. Über diese Solarzellen wird die Strahlungsenergie in elektrische Energie in Form von Gleichstrom umgewandelt. Oft werden mehrere Solarzellen gebündelt, da eine einzelne Solarzelle nur über eine geringe Spannung verfügt.

Es gibt zwei Möglichkeiten, wie die elektrische Energie verwendet werden kann. Zu einen kann die Energie ins öffentliche Netz eingespeist werden. Zum Anderen kann die Energie zwischengespeichert werden. Die Solaranlage arbeitet in diesem Fall im Inselbetrieb. Beim Inselbetrieb besteht keine Verbindung zum öffentlichen Netz. Überschüssige Energie wird zwischengespeichert. Diese Zwischenspeicherung kann anhand von Akkumulatoren stattfinden. Diese speichern die überschüssige Energie und geben sie bei Bedarf ab.

Für unsere Versuche und Experimente spielt jedoch der Inselbetrieb keine Rolle, da unsere Photovoltaikanlage als zusätzliche Energiequelle dienen soll, um die Verbraucher zu versorgen. Es besteht kein Bedarf die Energie zwischen zu speichern. In unserem Fall wird die Energie in unser Netz eingespeist. Wir können die Energie nicht direkt in unser Netz einspeisen, da es sich um Gleichstrom handelt. Der Gleichstrom muss zuerst in Wechselstrom umgewandelt werden, bevor er ins öffentliche Netz eingespeist werden kann. Hier kommt ein Wechselrichter ins Spiel. Ein Wechselrichter wandelt Gleichstrom in Wechselstrom um.

Eine kurze Einleitung zum Thema Wechselrichter findet sich in dem nachfolgenden Abschnitt.

2.11.2 Wechselrichter

Ein Wechselrichter, auch Inverter genannt, ist ein elektrisches Gerät, welches Gleichstrom in Wechselstrom umwandelt. Solche Geräte werden oft bei Photovoltaik-Anlagen, in Kraftfahrzeugen oder anderen Einrichtungen genutzt. Man unterscheidet zwischen zwei verschiedenen Steuerarten bei Wechselrichtern. Es gibt

- selbstgeführte Wechselrichter
- fremdgeführte Wechselrichter

Selbstgeführte Wechselrichter erzeugen unabhängig vom Stromnetz die Wechselspannung und werden meist im Inselbetrieb genutzt, während fremdgeführte Wechselrichter netzsynchronen Wechselstrom erzeugen. In diesem Fall wird die Spannung und die Frequenz an das öffentliche Netz angepasst. Auch bei netzgekoppelten Photovoltaik-Anlagen wird oft ein fremdgeführter Wechselrichter genutzt. Man unterscheidet drei Varianten von Wechselrichtern bei netzgekoppelten Photovoltaik-Anlagen.

- Modulwechselrichter
- Stringwechselrichter
- Zentralwechselrichter

Modulwechselrichter werden direkt am Solarmodul montiert und können parallel geschaltet werden, um evtl. Verluste aufgrund von unterschiedlichen Beleuchtungsstärken der Solarmodule zu verhindern.

Stringwechselrichter sind mit einem Kabel mit mehreren Solarmodulen, die in Reihe geschaltet sind, verbunden. Sie sind die am weitesten verbreiteten Wechselrichter für Photovoltaik-Anlagen

Zentralwechselrichter sind große Wechselrichter, die meistens in einem eigenen Raum untergebracht sind.

2.11.3 Gleichstromaggregate

Die Projektgruppe nutzt für ihre Experimente folgende Gleichstromaggregate, welche in Abbildung 2.25 zu sehen sind. Die Gleichstromaggregate dienen als Simulation einer Photovoltaik-Anlage, da keine echte Anlage zur Verfügung steht. Die Gleichstromaggregate erzeugen Gleichstrom. Sie werden über eine SPS angesteuert und variieren bei vom Netz vorgegebener Spannung den Ausgangsstrom, so dass der Leistungsollwert erreicht wird, den sie über die SPS erhalten haben.

Die Verbindung zur Netzsimulation wird über Wechselrichter hergestellt. Diese wandeln den erzeugten Gleichstrom in Wechselstrom um und speisen ihn in die Netzsimulation ein.

In unserem Fall hat das Gleichstromaggregat nur die Aufgabe, die zu erzeugende Leistung auszugeben. Die Vorgaben richten sich dabei nach einem zeitabhängigen Leistungsprofil, das weiter unten erläutert wird.

Die Gleichstromaggregate drehen den Strom nach und nach hoch bis sie den maximalen Wert von der SPS erhalten haben und fahren dann die Spannung herunter. Der Wechselrichter, der hinter den Gleichstromaggregaten sitzt, bemerkt, dass die Spannung herunter geht.

Abbildung 2.25: Gleichstromaggregate der PVA

2.11.4 Matlab Simulation

Die Werte für das Gleichstromaggregat wurden mittels einer Matlabsimulation bestimmt. Die Werte wurden künstlich am PC simuliert und anschließend über die SPS an das Gleichstromaggregat und somit an den Wechselrichter weitergeleitet. Die Simulation und die Gleichstromaggregate bilden unsere Photovoltaik-Anlage, wobei lediglich nur ein Gleichstromaggregat genutzt wurde. Die topologische Reihenfolge ist im Kapitel 2.13 auf der Abbbildung 2.29 zu sehen.

Die Simulation lag als Matlab-Programm vor [11]. Mit dieser Simulation war es möglich für jeden eingegebenen Tag eines Jahres ein Leistungsprofil für die Photovoltaikanlage zu simulieren. Die genaue Simulation der Photovoltaikanlage besteht aus einer Matlab-Funktion. Die komplette Matlab-Funktion, die die Leistung für ein Zeitintervall erzeugt, ist in Abbildung 2.26 und Abbildung 2.27 zu sehen. Es wurden lediglich aus Platzgründen und Übersichtlichkeit sämtliche Kommentare im Quellcode entfert.

```
function [ResultMatrix] = PVAmod(startTime,stopTime,stepWidth)

% Berechnung der Anzahl der zu erzeugenden Werte
% Übergebene Zeiten müssen im datenum-Format vorliegen

simulationTime = stopTime - startTime;
numberOfSteps = floor(simulationTime/stepWidth);
tempResult = zeros(numberOfSteps,3);
erdneigung = 23.5 * pi / 180;
zzone = 1;
Wirkungsgrad = 0.14;
Sonnenleistung = 1000;
breite = 52.5;
azimut = 0;
elevation = 30;
Flaeche = 25;
el_max = Flaeche * Sonnenleistung * Wirkungsgrad;
aktueller_tag = 1*31+12;
fh = fopen('Dezember.csv', 'w');
% SimulationsSchleife
tempResult = cell(1);
for i = 1:1:numberOfSteps
presentSimulationTime = (startTime + (i*stepWidth));
jahreszeit = (datevec(presentSimulationTime)-[0 1 1 0 0 0])*[0;730;24;1;0;0];
tageszeit = datevec(presentSimulationTime)*[0;0;0;3600;60;1];
tempPresentSimulationTime = datevec(presentSimulationTime);
monthOfPresentSimTime = tempPresentSimulationTime(2);
dayOfPresentSimTime = tempPresentSimulationTime(3);
if (aktueller_tag~=(monthOfPresentSimTime*31 + dayOfPresentSimTime))
wolken = [0:1:floor(1/stepWidth)];
zeit = [0:1:floor(1/stepWidth)];
for j = 1:1:floor(1/stepWidth)
wolken(j) = 1 - (rand^2);
end; % generieren.
aktueller_tag = (monthOfPresentSimTime*31 + dayOfPresentSimTime);
end;

%neue Außentemperatur bestimmen
help = interp1(zeit,wolken,[0:1/60:24],'pchip');
T_aussen = help(1+datevec(presentSimulationTime)* [0;0;0;60;1;0]);
help = interp1(zeit, wolken,[0:1/30:96],'pchip'); %30sec-Werte erzeugen
daempfung = help(1 + floor(datevec(presentSimulationTime)* [0;0;0;120;2;1/30]));
daempfung = abs(daempfung); % daempfung wurde positiv gesetzt
s = [cos(erdneigung)*cos((jahreszeit-8534)*pi/4380);
sin(erdneigung)*cos((jahreszeit-8534)*pi/4380)
-sin((jahreszeit-8534)*pi/4380)];
if (s(1)==0) & (s(3)>0)
s_winkel = 90;
elseif (s(1)==0) & (s(3)<0)
s_winkel = -90;
else
s_winkel = atan(s(3)/s(1))*180/pi;
end;
if (s(1)<0) & (s(3)>0)
s_winkel = s_winkel+180;
elseif (s(1)<0) & (s(3)<0)
s_winkel = s_winkel-180;
end;
zenit = [cos(breite*pi/180)*cos(((laenge-(15*zzone)-s_winkel)*pi/180)+(tageszeit*pi/43200));
sin(breite*pi/180);
-sin(((laenge-(15*zzone)-s_winkel)*pi/180)+(tageszeit*pi/43200))*cos(breite*pi/180)];
```

Abbildung 2.26: Quellcode der Simulationsfunktion

```
nord = [-cos(((laenge-(15*zzone)-s_winkel)*pi/180)+(tageszeit*pi/43200))*sin(breite*pi/180);
cos(breite*pi/180);
sin(((laenge-(15*zzone)-s_winkel)*pi/180)+(tageszeit*pi/43200))*sin(breite*pi/180)];
h = nord * cos(azimut*pi/180) + cross(zenit,nord) * sin(azimut*pi/180);)
n = zenit * cos(elevation*pi/180) - h * sin(elevation*pi/180);
zenitwinkel = acos(dot(-s,zenit)/(norm(s)*norm(zenit)))*180/pi;
einfallswinkel = acos(dot(-s,n)/(norm(s)*norm(n)))*180/pi;
if zenitwinkel > 90 % in der Nacht
power = 0;
else
power = -dot(n,s)*Flaeche*Sonnenleistung*Wirkungsgrad*daempfung;
end;
if power < 0
power =0;
end;
strahlungs_leistung = -dot(n,s)*Sonnenleistung*daempfung; % [W/m^2]
if strahlungs_leistung < 0
strahlungs_leistung =0;
end;
% Daten in die Ergebnismatrix eintragen
tempResult{i,1} = i;
tempResult{i,2} = datestr(presentSimulationTime,'dd-mm-yyyy HH:MM:SS');
tempResult{i,3} = power;
fprintf(fh, '%d\t%s\t%d\n', tempResult{i,1}, tempResult{i,2}, tempResult{i,3});
%tempResult(i,1) = i;
%tempResult(i,2) = presentSimulationTime;
%tempResult(i,3) = power;
end;
ResultMatrix = tempResult;
%save -double -tabb Month.csv tempResult
fclose(fh);
```

Abbildung 2.27: Quellcode der Simulationsfunktion

Als Eingabe für die Simulation wird die *Starttime* und die *Stoptime* angegeben sowie die *Schrittgrösse*. Die Eingabeform besteht aus dem Befehl **PVA startime, Stoptime, stepwidth**. PVA ist hier der Dateiname der Simulation. In der Simulation ist die Schrittgrösse mit **stepWidth** bezeichnet. Die beiden anderen Eingabewerte *Starttime* und *Stoptime* sind die Tage zwischen denen gemessen werden soll. Als Eingabe wird jeweils das Datum eingegeben. Es ist jedoch wichtig, dass die eingebenen Daten im datenum Format vorliegen. *StepWidth* gibt die Schritte aus, wie oft ein Messwert ausgegeben werden soll. Eine Eingabe von 1 für *stepWidth* würde einen einzelnen Wert für einen Tag auswerfen. So berechnet der Wert $\frac{1}{24}$ jeweils für jede Stunde einen Wert und wird anschliessend ausgegeben. Die Simulation beginnt jeweils um 0.00 Uhr eines Tages und endet dementsprechend um 24.00 Uhr am letzten Tag des Intervalls. Zudem wird eine Dämpfung in der Simulation berechnet, die dazu dient, Wolken während eines Tages zu simulieren. Der eigentliche Wert in der Funktion war fehlerhaft und musste korrigiert werden. Nach der Korrektur liefert die Dämpfung stets absolute Werte. Vorher war die Dämpfung 0 bzw. negativ, was zur Ursache hatte, dass die ausgegebene Leistung immer 0 war. Weiterhin berechnet die PVA-Funktion intern sämtliche Paramter (Winkel zwischen Zenit und dem Sonnenschein, Winkel zwischen PVA und dem Sonnenschein etc.), die zur Berechnung der Leistung nötig sind. Darauf wird nicht konkret eingegangen, da lediglich die Messung und die Ausgangsleistung wichtig sind. Anzumerken ist nur, dass die Koordinaten für Berlin in der Funktion eingetragen waren.

Zu Beginn gab es erhebliche Probleme mit der Ausgabe. Die Daten werden generell ins Matlab-interne Datumformat umgerechnet und es wird nicht die normale Form für ein Datum dd.mm.yyyy verwendet. Dieses Format ist jedoch bei der Auswertung wesentlich einfacher. So wurde die eigentliche Matlab-Funktion leicht modifiziert, so dass nicht die eigentliche Datumsanzeige, wie sie von Matlab verwendet wird, angezeigt wird. Ansonsten müssten die Daten erneut unter Java zurück konvertiert werden. Die Ausgabe der Funktion unter Matlab erfolgt generell als Matrix. Folgende Spalten sind in der Ergebnismatrix:

- Erste Spalte: **Index i**
- Zweite Spalte: **Datum der jeweiligen Zeitangabe**
- Dritte Spalte: **Leistung in Watt**

i ist der fortlaufende Index, der bei jeder neuen Ausgabe einer Zeile um eins erhöht wird. Er entspricht der Schrittgrösse *StepWidth*. Das Datum in der zweiten Spalte wird nach Modifikation in folgender Form ausgegeben:

dd-mm-yyyy HH:MM:SS

Es ist wichtig, dass Stunden, Minuten und auch Sekunden ausgegeben werden, da später ein sehr kurzes Zeitintervall für unsere Messung gewählt wird und die Anzeige für den Tag selber sowie für die Stunden und Minuten nicht zur Unterscheidung der Messungen ausreichen würden. Die letzte Spalte gibt die Leistung in Watt aus. Die Messungen werden direkt in eine csv-Datei geschrieben. Diese csv-Dateien sind die Grundlage für die spätere Java-Implementierung und dienen als Datenquelle für das Gleichstromaggregat.

Der Inhalt solch einer Auswertungsdatei ist auf Abbildung 2.28

```
1323        21-07-2007 05:30:45        9.392792e-001
1324        21-07-2007 05:31:00        2.757160e+000
1325        21-07-2007 05:31:15        4.570440e+000
1326        21-07-2007 05:31:30        6.437413e+000
1327        21-07-2007 05:31:45        8.266465e+000
1328        21-07-2007 05:32:00        1.020882e+001
1329        21-07-2007 05:32:15        1.205895e+001
1330        21-07-2007 05:32:30        1.409989e+001
1331        21-07-2007 05:32:45        1.597596e+001
1332        21-07-2007 05:33:00        1.785234e+001
1333        21-07-2007 05:33:15        2.004207e+001
1334        21-07-2007 05:33:30        2.233750e+001
1335        21-07-2007 05:33:45        2.427832e+001
1336        21-07-2007 05:34:00        2.672347e+001
1337        21-07-2007 05:34:15        2.870220e+001
1338        21-07-2007 05:34:30        3.130796e+001
1339        21-07-2007 05:34:45        3.332773e+001
1340        21-07-2007 05:35:00        3.610186e+001
1341        21-07-2007 05:35:15        3.816533e+001
1342        21-07-2007 05:35:30        4.111255e+001
1343        21-07-2007 05:35:45        4.322195e+001
1344        21-07-2007 05:36:00        4.533166e+001
1345        21-07-2007 05:36:15        4.850096e+001
1346        21-07-2007 05:36:30        5.179602e+001
1347        21-07-2007 05:36:45        5.400221e+001
1348        21-07-2007 05:37:00        5.746584e+001
1349        21-07-2007 05:37:15        5.972200e+001
1350        21-07-2007 05:37:30        6.334650e+001
1351        21-07-2007 05:37:45        6.565309e+001
1352        21-07-2007 05:38:00        6.942766e+001
1353        21-07-2007 05:38:15        7.178470e+001
1354        21-07-2007 05:38:30        7.569542e+001
1355        21-07-2007 05:38:45        7.810248e+001
1356        21-07-2007 05:39:00        8.213234e+001
1357        21-07-2007 05:39:15        8.458854e+001
1358        21-07-2007 05:39:30        8.871741e+001
1359        21-07-2007 05:39:45        9.122144e+001
1360        21-07-2007 05:40:00        9.542609e+001
1361        21-07-2007 05:40:15        9.797618e+001
1362        21-07-2007 05:40:30        1.022303e+002
1363        21-07-2007 05:40:45        1.048242e+002
1364        21-07-2007 05:41:00        1.090982e+002
1365        21-07-2007 05:41:15        1.117334e+002
1366        21-07-2007 05:41:30        1.143688e+002
1367        21-07-2007 05:41:45        1.186680e+002
1368        21-07-2007 05:42:00        1.228811e+002
1369        21-07-2007 05:42:15        1.255889e+002
1370        21-07-2007 05:42:30        1.297149e+002
1371        21-07-2007 05:42:45        1.324532e+002
1372        21-07-2007 05:43:00        1.364501e+002
1373        21-07-2007 05:43:15        1.392146e+002
1374        21-07-2007 05:43:30        1.430374e+002
```

Abbildung 2.28: Auswertung für Juli 2007

2.11.5 Auswertung

Zunächst wurde für jeden Monat eine Messung durchgeführt. So wurde jeweils der 21. Tag eines jeden Monats komplett von 0.00 Uhr bis 24.00 Uhr simuliert und anschliessend in die besagten csv-Dateien abgespeichert. Das Zeitintervall für eine Messung lag bei 15 Sekunden. Der StepWidth-Wert lag somit bei 86400. Am Ende lagen somit 12 csv-Dateien vor, die jeweils mit dem Monatsnamen benannt wurden. Diese Dateien dienen als Grundlage füer spätere Tests sowie für die abschließsenden Experimente. Bei den Experimenten wurde ein Monatsprofil ausgewählt. Das Gleichstromaggregat erzeugt die dementsprechende Leistung und speist diese über den Wechselrichter dann in die Netzsimulation ein.

Einen Einfluss auf die ausgegebene Leistung hat man nur über vorgegebene Parameter wie Jahreszeit und Standort der simulierten Photovoltaikanlage. Die Simulationsfunktion berechnet die Leistung automatisch anhand diese Parameter.

2.12 Photovoltaik-Simulation mit Java

2.12.1 Allgemeine Beschreibung der Photovoltaik-Simulation

Es ist erwünscht, die Leistung einer Photovoltaikanlage zu simulieren. Dazu gibt vorgegebene simulierte Leistungen, die bereits durch ein Matlab-Programm erzeugt wurden. Weiterhin ist ein Gleichstrom-Aggregat erforderlich, das die Rolle einer Photovoltaikanlage spielt und reale Leistungen erzeugt sowie durch die SPS gesteuert wird. Das Zusammenführen der Leistungswerte und die Erzeugung der Leistungen auf Hardware wurde mit einem Java-Programm ausgeführt.

Die Photovoltaik-Simulation des Java-Programms besteht aus drei Schritten: Zuerst werden Leistungswerte aus einem Profil gelesen, das anhand eines MatLab-Programms erstellt wurde. Danach werden die Werte nacheinander über Modbus zu einer SPS geschickt, mit denen die SPS das Gleichstrom-Aggregat steuert. Anschließend liest Java die Leistungswerte wieder aus der SPS aus.

2.12.2 Die Werte der Leistung aus der Dateien lesen

Das Auslesen der Werte aus einer Datei ist eine Vorbereitung für die Ausführung der Photovoltaik-Simulation. Dafür ist die Klasse „LoadValues“ zuständig. Sie liest die kompletten Leistungswerte aus einem Profil (hier „loadname“) aus, das die Leistungen speichert, die eine Photovoltaikanlage an einem bestimmten Tag in einer bestimmten Zeit erbringt. Die Werte werden hier unter einer Vektorvariable „data“ gespeichert. Die Simulationszeiten, die in jedem Zeitpunkt mit einem Leistungswert zusammen angegeben wurden, wurden auch unter einer Vektorvariable „simulationtime“ aufgenommen. Das heißt, man kann sowohl die Leistungswerte als auch die Zeit wissen, wann welcher Leistungswert gerade ist. Folgender Code stellt den ganzen Prozess dar.

```
1 Vector<Float> data;
2 Vector<String> simulationtime;
3
4 public LoadValues(String loadname) throws Exception {
```

```
5
6         FileReader fr = new java.io.FileReader(loadname);
7
8         BufferedReader br = new java.io.BufferedReader(fr);
9
10        data = new Vector<Float>();
11        simulationtime = new Vector<String>();
12
13        int i = 0;
14        while(br.ready()) {
15
16                String line = br.readLine();
17                // {index, Simulationtime, power}
18                String value[] = line.split("\t");
19
20                data.add( new Float( Float.parseFloat(
21                        parsePower(value[2]))) );
22                simulationtime.add(new String(value[1]));
23                i++;
24        }
25        log.info(i + "Dateien wurde gelesen");
26
27        br.close();
28        fr.close();
29 }
```

2.12.3 Photovoltaik simulieren mit Gleichstrom-Aggregat-Controller (GSAController)

Der Gleichstrom-Aggregat-Controller (GSAController) ist der Kernprozess und das Organisationszentrum der Photovoltaik-Simulation. Der Controller ist eine Subklasse des RegisterObservers[7], also kann er Werte lesen, die man wissen möchte. Beim Aufrufen dieser Klasse muss man festlegen, unter welcher Registeradresse einer SPS und wie viele Register man lesen möchte. Man kann das Tempo zwischen zwei Lesevorgängen zudem einstellen. Anderseits enthält der GSAController einen UPnP-Knoten. Er packt die gelesenen Werte in einen UPnP-Knoten. Dieser Knoten wird für die Netzwerksteuerung benötigt, um die Informationen von dem Knoten transportiert werden zu können.

```
1  public GSAController(ModbusController mbCtl, final short referenceNr,
2          final short wordCount, int delay, int spsid)
3  {
4          //regestriere Oberserver
5          super(mbCtl, referenceNr, wordCount, delay);
6
7          running = false;
8          node = new Inverter(spsid, Node.State.PU_NODE);
9          node.setSollWirkleistung(initialSollWirkleistung);
10         device =
11         GenericUPnPDeviceFactory.getInstance().createDeviceFromObject
12         (node);
13
14         node.addObserver(this);
15 }
```

Eine weitere wichtige Funktion dieser Klasse ist es, Werte in die SPS-Register zu schreiben. Diese Funktion wurde in der Methode „valuesToSPS“ implementiert.

In dieser Methode wurde zuerst eine Instanz von „LoadValues“ erzeugt, damit die Werte vorhanden sind, die simuliert werden sollen. Sie sind alle Leistungswerte, die die Leistungen einer Photovoltaikanlage an einem Tag simulieren. Dann werden die Werte nach einer bestimmten Zeitdistanz (hier:

[7] „RegisterObserver“ ist eine Klasse, die Werte der Register einer SPS liest, und die Register beobachtet, ständig nach vorgegebener Zeiteinheit nachfragt, ob eine Wertveränderung vorliegt. Wenn ja, werden gelesene Werte aktualisiert. Hier wurde die Klasse „ActionListener“ implementiert, indem man die Methode „actionPerformed“ implementiert, und dadurch werden die Veränderungen der Register überprüft. Wenn eine Veränderung der beobachteten Register existiert, werden die gelesenen Werte aus den Registern sofort aktualisiert. Also ist „RegisterObserver“ eine Subklasse von „Observable“.

stepWidth) nacheinander in die SPS-Register geschrieben. „referenceNr“ ist die Registeradresse. Ab dieser Adresse schreibt die Klasse die Werte in die nacheinander hängenden Register ein.

In der Simulation ist das Einstellen der Simulationsbeginnszeit möglich. Man kann als Parameter unter „time“ die Startzeit angeben, ab wann die Simulation starten soll.

```
public void valuesToSPS (String filename, int stepWidth,
        final short referenceNr, int faktor,
        int time)throws MessageParsingException
{
        LoadValues lv = null;

        try{
                System.out.println("filename ist : " + filename);
                lv = new LoadValues(filename);
        }catch (Exception e){
                System.out.println("this File" + filename + e);
                return;
        }

        System.out.println("size ist : " + lv.data.size());

        int i = (time * 60 * 60) / 15; // Zeitindex ausrechnen

        if (i >= lv.data.size()) {
                System.out.println("time out of bounds");
        }

        boolean sync = true;

        if (sync) {
                System.out.println("Warte auf synchronisiertes Start Signal...");
                while (!running) /* warte auf asynchrones sync*/;
        } else {
                System.out.println("Starte ohne auf synchronisation zu warten.");
                running = true;
        }

        for(; i < lv.data.size(); i++) {
                List<Short> s = new ArrayList<Short>();
                int value;

                System.out.println(
                lv.toString(i) + " Watt (Tabelle) => " + lv.data.get(i) / faktor + " Watt (Einspeisung total)");

                double watt = lv.data.get(i) / faktor;

                // Beschr?nken der zur sps gesendeten
                //werte auf maximal sollWirkleistung
                if (limit) watt =
                Math.min(watt, node.getSollWirkleistung());

                value = HelperFunctions.reinterpretAsInt((float) watt);
                s.add(HelperFunctions.getShortFromInt(value, false));
                s.add(HelperFunctions.getShortFromInt(value, true));
                this.getMbCtl().writeMultipleRegisters(referenceNr,s);

                writeWait(stepWidth);

        }

        running = false;
}
```

Um die Photovoltaik-Simulation mit zwei anderen Simulationen gleichzeitig starten zu können, wird eine boolesche Kontrollvariable: „sync“ benötigt, die dafür sorgt, dass die Simulation gestartet wird, wenn sie von „false“ auf „true“ geändert wird.

Die Leistungen können von einem Kontrollknoten von Außen eingeschränkt werden. Wenn die boolesche Variable „limit“ „true“ gesetzt ist, wird die von außen eingegebene Einschränkung eingeschaltet und wahrgenommen. Der kleinste Wert vom Einschränkungswert und vom aktuellen Wert aus der Datei wird in das SPS-Register geschrieben.

2.13 SPS-Implementierung Photovoltaik

2.13.1 Nachbildung einer Photovoltaikanlage

Die Aufgabe der SPS besteht darin, die simulierten Daten einer Photovoltaikanlage an das angeschlossene Gleichstromgerät weiterzuleiten. Das Gleichstromaggregat vertritt also eine Photovoltaikanlage. Wie in der Abbildung2.29 dargestellt wurde, speist das Gerät die erzeugten Leistungen über einen Wechselrichter ins Netz ein. Im Labor stehen Gleichstromaggregate DC power supply 62012P-80-60 zur Verfügung, die sowohl durch die Spannung als auch durch Strom steuerbar sind. Die SPS - Implementierung ist für die gleichzeitige Steuerung von mehreren Geräten gedacht.

Abbildung 2.29: *Nachbildung einer Photovoltaikanlage*

Die SPS kommuniziert mit Java über Modbus und verhält sich als Modbusslave. So werden die Daten ständig angefragt.Siehe dazu auch Kapitel 2.5

2.13.2 Steuerung

An der SPS sind Klemmen für die Steuerung von drei Gleichstromgeräte eingesetzt. Das sind zwei Ausgänge und zwei Eingänge pro Gerät. Welche von denen genutzt werden, ist abhängig davon, welche Steuerung vorgezogen wird. Wie oben erwähnt wurde, ist das Gleichstromaggregat durch die Spannung oder durch den Strom steuerbar. In unserem Fall wird es durch den Strom gesteuert. Zu diesem Zweck gibt es eine Ausgangsvariable `Isoll_Netzteil1.`Sie setzt den Sollwert am Eingang der Maschine. Die Spannung wird als Konstante (35 V) eingestellt.

Der Sollwert des Stroms ergibt sich aus den Solldaten, die von Java empfangen werden.

Die simulierten Leistungen einer Photovoltaikanlage werden in einem gemeinsamen Speicherbereich geschrieben, der von der SPS-Seite als Merkeradresse bezeichnet wird.

```
VAR_GLOBAL
Netzteil1_USoll_test AT %MW0 : WORD:=35;

SPSinput_Psoll AT %MD2 : REAL := 0;
END_VAR
```

An der Adresse MD2 (doubleword) wird der Wert von der Sollleistung angelegt. Die Java-seitige Adresse lautet 0x3004. Daraus wird der Wert für `Isoll` berechnet und weitergegeben.

Darstellung der Werte an den Ausgängen.

Analoge Ausgänge der SPS können die Werte von `0V-10V` in `WORD` annehmen. Daher ist es wichtig die Sollwerte richtig darzustellen. Die Umrechnung sieht in dem SPS - Code z.B. wie folgend aus:

```
Isoll_Netzteil1:=REAL_TO_WORD(temp/60*32768);
```

2.14 Consumer

2.14.1 Einleitung

Im folgenden Abschnitt werden die Verbraucher, auch Consumer genannt, näher erklärt. Die genauere Erläuterung der Implementierung findet anschliessend im Kapitel 2.15 statt. Die Consumer sind ein wichtiger Bestandteil in der Laborumgebung.

Wir haben für die Stromerzeugung in unserer Laborumgebung u.a. zwei Synchronmaschinen sowie eine Asychronmaschine. Dazu kommt noch die Photovoltaik-Simulation, die auch zur Stromerzeugung beiträgt. Diese Stromerzeugungsquellen stellen jeweils Kraftwerke, Windanlagen und Solaranlagen dar. Um jedoch eine reale Umgebung zu schaffen und den RealDezent-Algorithmus unter echten Bedingungen zu testen, benötigen wir 'Haushalte', die den erzeugten Strom verbrauchen bzw. einen ständigen Bedarf an Strom benötigen, der tageszeitabhängig ist. Erst durch die Verbraucher kann überhaupt eine vernünftige Bilanzierung stattfinden.

Diese Haushalte werden durch unsere Verbraucher simuliert. Uns stehen im Labor des Fachbereichs Elektrotechnik drei solcher Konsumenten zur Verfügung. Hierbei handelt es sich um drei einzelne elektronische Lasten, die den Stromverbrauch kontinuierlich simulieren können. Eine solche elektronische Last ist auf Abbildung 3.11 zu sehen.

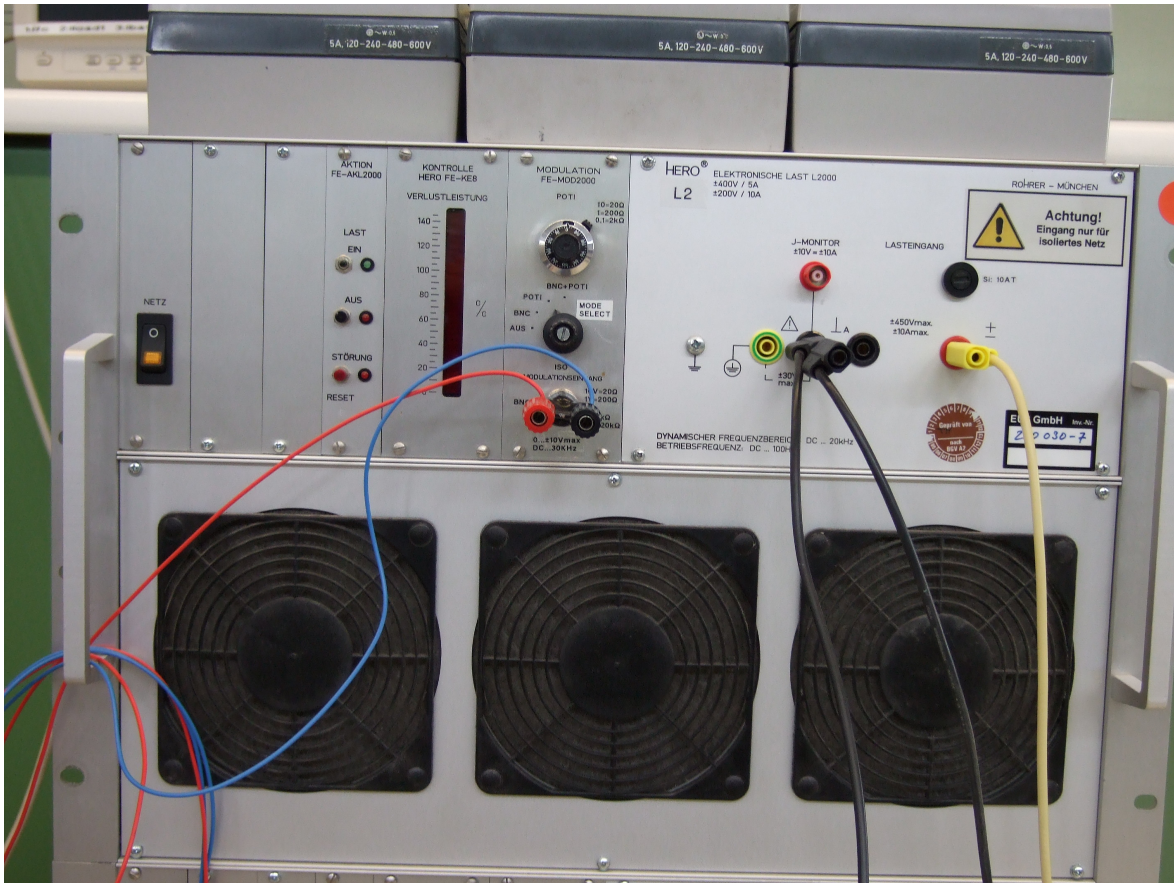


Abbildung 2.30: Consumer Frontansicht

Hier sieht man die Frontansicht des Geräts. Gut zu erkennen ist bereits die Verkabelung des Geräts. Weiterhin sind die grossen Lüfter erkennbar, die für eine ständige Kühlung der Geräts sorgen. Die aufgenomme Leistung wird in Wärme umgewandelt.

2.14.2 Elektronische Lasten

Die Geräte sind vom Typ HERO elektritronische Lasten L2000. Ein Gerät und somit ein Consumer schafft eine maximale Leistung von 1 Kilowatt. So würden uns rein theoretisch ein Maximalverbrauch von 3 Kilowatt zu Verfügung stehen. Wir beschränken uns jedoch in unseren Experimenten und Tests aus Sicherheitsgründen auf maximal 300 - 400 Watt pro Gerät.

Die Funktionsweise einer elektronischen Last ist recht simpel. Die von der eElektronischen Last aufgenommene elektrische Energie wird in diesem Fall in Wärmeenergie umgewandelt, zur Kühlung werden Lüfter oder wassergekühlte Elemente verwendet. Auch eine Rückspeisung in das öffentliche Stromversorgungsnetz ist unter bestimmten Voraussetzungen möglich.

2.14.3 Funktionsweise der Consumer

Die Grundidee für die Steuerung der Consumenten ist recht einfach. Alle drei Consumer können und werden parallel und unabhängig von einander betrieben. Die drei Consumer werden von einer SPS angesteuert. Die genauere Implementierung der SPS und des Java-Teils werden in den unteren Abschnitten ausführlich erläutert.

Erste Versuche befassten sich mit dem generellen Ansteuern der Consumer. Hierbei wurde versucht per Java einen Wert an die zugehörige SPS zu senden. Dieser Wert wurde dann an die jeweiligen Lasten weiter gegeben, die dementsprechend Leistung verbrauchen. Das Setzen des Werts in der SPS erfolgt in regelmässigen Intervallen, so dass eine genaue Beobachtung möglich war.

Um den Verbrauch der einzelnen Consumer zudem manuell zu überwachen, stehen uns Messgeräte des Fachbereichs Elektrotechnik zu Verfügug. Auf der Abbildung 2.31 sind die jeweiligen Messgeräte für jeden Consumer zu sehen.

Im späteren Experimentverlauf wurden die Werte nicht mehr direkt über Java manuell eingegeben, sondern konnten per UPnP gesetzt werden. Zudem wurden auch vorgefertigte Leistungsprofile genutzt. Diese Profile basieren auf Verbraucherverhalten von Haushalten zu verschiedenen Zeiten an einem Tag.

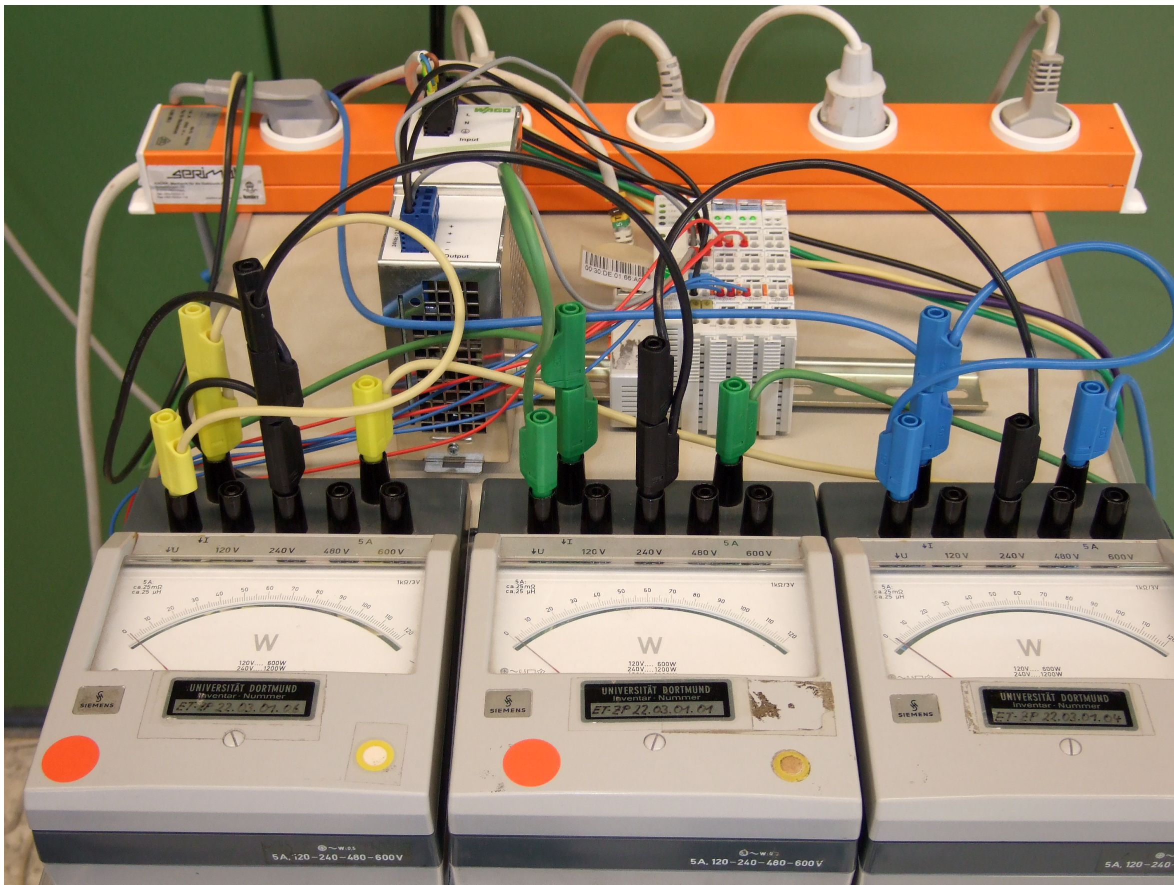


Abbildung 2.31: Messgeräte für Consumer

2.15 Consumer-Simulation mit elektronischen Lasten

2.15.1 Allgemeine Beschreibung der Consumer-Simulation

Die Consumer-Simulation ist die Simulation, die den Stromverbrauch simuliert. Drei Lasten werden hier benötigt, die beispielweise elektronische Hausgeräte darstellen und in einer bestimmten Zeit einen gewissen Strom verbrauchen. Die drei Lasten sollen die vorgegebenen Leistungen (Strom) verbrauchen.

Man hat hier drei verschiedenen Methoden um die Leistungen, die verbraucht werden sollen, zu erzeugen.

1. Man kann die Leistungenswerte zufällig im Programm automatisch generieren lassen.
2. Man kann die Leistungen durch UPnP eingeben.
3. Die Leistungswerte können aus einer Datei ausgelesen werden.

Dann werden die Leistungswerte in die SPS-Register geschrieben und schließlich von den Lasten verbraucht. Zwischen den Leistungen, die von den Lasten tatsächlich verbraucht wurden und den Leistungen, die man vorgeschlagen hat, könnte es eventuell kleinere Abweichungen geben. Es ist dann wichtig die Istwerte zu wissen, die das Java-Programm aus Registern der SPS auslesen kann.

2.15.2 Beschreibung der Java-Klasse: „ConsumerController"

„ConsumerController" ist die Klasse, die das Verbrauchen der drei Lasten simuliert. Jede Last ist wieder ein UPnP-Knoten. Durch UPnP können die Werte und Zustände der Lasten von Außen gesetzt und beobachtet werden. Diese Klasse ist eine Subklasse von „RegisterObserver"[8], weil die Leistungswerte aus der SPS ausgelesen und die Wertveränderungen gemerkt werden müssen.

```
1  public ConsumerController(ModbusController mbCtl, final short referenceNrforRead,
2              final short wordCount, int delay, int spsid) {
3          super(mbCtl, referenceNrforRead, wordCount, delay);
4
5          // Kontrollschnittstelle fuer einen Operator: ConsumerController-Device.
6          deviceThis = GenericUPnPDeviceFactory.getInstance().createDeviceFromObject(this);
7
8          // Nodes und Node-Devices initialisieren.
9          deviceConsumer = new GenericDevice[3];
10         consumers = new Consumer[3];
11         for (int i = 0;  i < 3; i++) {
12                 consumers[i] = new Consumer(1001 + i, Consumer.State.PU_NODE );
13                 consumers[i].setSollWirkleistung(-maxP);
14                 deviceConsumer[i] =
15                 GenericUPnPDeviceFactory.getInstance().createDeviceFromObject(consumers[i]);
16                 consumers[i].addObserver(this);
17         }
18
19         // Senden und Empfangen starten.
20         // Achtung: Erst hier starten, wegen threading Problemen
21         // (Listener wird sonst aufgerufen bevor consumers initialisiert wurden).
22         shutdown = false;
23 }
```

Diese Klasse hat verschiedene Modi für verschiedene Generierungen der Leistungswerte. Die sind: „same", „random" und „readfile".

„same" bedeutet: Alle Lasten sollen gleiche Werte haben, wenn „same" „true"wäre.

Mit „random" kann man die Leistungswerte im Programm generieren lassen, wenn „random" auf „true" ist. Die Generierung erstellt die Methode „randomP", um Leistungswerte zu erzeugen und die Methode „generateUfromP", um die entsprechende Spannungen aus Leistungswerten zu generieren. Wenn „random" auf „false" ist, könnte man die Werte durch UPnP manuell setzen. Es gibt für jede Last eine Set-Methode, die von UPnP aufgerufen werden kann und dadurch die Werte gesetzt werden können.

```
1  private float randomP(float maxP) {
2          return (float)(Math.random()*maxP);
3  }
```

[8]„RegisterObserver" ist eine Klasse, die Werte der Register einer SPS liest, und die Register beobachtet, ständig nach vorgegebener Zeiteinheit nachfragt, ob eine Wertveränderung vorliegt. Wenn ja, werden gelesene Werte aktualisiert. Hier wurde die Klasse „ActionListener" implementiert, indem man die Methode „actionPerformed" implementiert, und dadurch werden die Veränderungen der Registern überprüft. Wenn eine Veränderung der beobachteten Register existiert, werden die gelesenen Werte aus den Registern sofort aktualisiert. Also ist „RegisterObserver" eine Subklasse von „Observable".

```
4
5 private float generateUfromP(float p) {
6         float random = (float)(260 + 5);
7         float u = p / random;
8         return u;
9 }
10
11 public void setP1(double p) {
12         p1 = Math.min((float)p, maxP);
13 }
14
15 public void setP2(double p) {
16         p2 = Math.min((float)p, maxP);;
17 }
18 public void setP3(double p) {
19         p3 = Math.min((float)p, maxP);;
20 }
```

Modus „same" und „random" schließen sich nicht gegenseitig aus. Man kann immer drei verschiedene Zufallswerte erzeugen, je nach, ob „same" auf „true" ist oder nicht, wird der erste Wert für alle drei Lasten genommen oder alle drei Zufallswerte für die jeweilige Last.

„readfile" benutzt man, wenn Werte in Frage kommen, die in einer Datei vorgegeben sind. Wenn „readfile" auf „true" gesetzt ist, muss „random" logischerweise ausgeschaltet werden. Anders herum muss „readfile" ausgeschaltet werden, wenn „random" „true" ist.

```
1 public void setRandom(boolean random) {
2         if(random) readfile = false;
3         this.random = random;
4 }
5
6 public void setReadfile(boolean readfile) {
7         if(readfile) random = false;
8         this.readfile = readfile;
9 }
```

2.15.3 Werte schreiben auf die SPS-Register mit „valuesToSPS"

Sowohl generierte Werte als auch aus einer Datei ausgelesene Werte sollen in die SPS-Register geschrieben werden. „referencNr" ist die Anfangsadresse der Register. In diese Adresse soll man die Werte nacheinander einschreiben. „referenceNrforModus" ist die Adresse, in der ein Modus geschrieben wird. Man kann die maximalen Leistungen, die von den Lasten verbraucht werden sollen, per Hand mit „maxP" angeben. Mit „shutdown" kann der Simulationsablauf abgebrochen werden und gleichzeitig ist es auch das Signal für das Synchronstarten der verschiedenen Simulationen. Wenn „shutdown" auf „false" gesetzt wird, wird die Consumer-Simulation gestartet.

```
1 public void valuesToSPS (final short referenceNr,
2         final short referenceNrforModus, float maxP, String filename)
3 {
4   this.referenceNr = referenceNr;
5   this.maxP = maxP;
6   float u1 = 0;
7   float u2 = 0;
8   float u3 = 0;
9
10  XYSeries series = null;
11  if(filename != null) {
12          CSVParser csvp = new CSVParser(filename);
13          if (csvp != null) {
14            series = csvp.normalizeSeries(csvp.getXYSeries());
15            //if (series != null) csvp.displaySeries(series);
16          }
17  }
18
19  while(!shutdown){
20
```

```
21    // Sende Leistungen/Spgen zur sps
22    try {
23      // U1 und P1 auf *alle* Ui und Pi auf SPS replizieren?
24      this.getMbCtl().writeSingleRegister(referenceNrforModus, same ? SPS_SAMEVALUES : SPS_DIFFRENTVALUES);
25
26      // Ui und Pi schicken.
27      List<Short> s = new ArrayList<Short>();
28      this.addValueToList(s, u1); this.addValueToList(s, p1);
29      this.addValueToList(s, u2); this.addValueToList(s, p2);
30      this.addValueToList(s, u3); this.addValueToList(s, p3);
31      this.getMbCtl().writeMultipleRegisters(referenceNr, s);
32    } catch(MessageParsingException e) {
33      e.printStackTrace();
34    }
35
36    if (this.random) {
37      writeWait(stepWidth);
38      // P1..3 und U1..3 mit Zufallswerten belegen
39      p1 = randomP(maxP);
40      p2 = randomP(maxP);
41      p3 = randomP(maxP);
42
43    } else if(this.readfile) {
44      if (series != null) {
45        if(seridx < series.getItemCount()) {
46          writeWait(stepWidth);
47          XYDataItem data = series.getDataItem(seridx);
48          seridx++;
49          float p = (float)(data.getYValue() * maxP);
50          // Beschr?nken der Consumierten Leistung
51          float sollP1 = (float)Math.abs(consumers[0].getSollWirkleistung());
52          float sollP2 = (float)Math.abs(consumers[1].getSollWirkleistung());
53          float sollP3 = (float)Math.abs(consumers[2].getSollWirkleistung());
54                    if (!limit) sollP1 = sollP2 = sollP3 = Float.MAX_VALUE;
55                    p1 = Math.min(p, sollP1);
56                    p2 = Math.min(p, sollP2);
57                    p3 = Math.min(p, sollP3);
58                    } else {
59                      this.readfile = false;
60                      seridx = 0;
61                    }
62      } else {
63        this.readfile = false;
64        seridx = 0;
65      }
66
67    } // if (this.readfile)
68
69    u1 = generateUfromP(p1);
70    u2 = generateUfromP(p2);
71    u3 = generateUfromP(p3);
72    } // while(!shutdown)
73
74
75    System.out.println("ConsumerController shutdown.");
76
77    try {
78      // Ui und Pi resetten.
79      List<Short> s = new ArrayList<Short>();
80      this.addValueToList(s, 0); this.addValueToList(s, 0);
81      this.addValueToList(s, 0); this.addValueToList(s, 0);
82      this.addValueToList(s, 0); this.addValueToList(s, 0);
83      this.getMbCtl().writeMultipleRegisters(referenceNr, s);
84    } catch(MessageParsingException e) {
85      // Wanted to stop anyway...
86      e.printStackTrace();
87    }
88    this.stop();
89  }
```

Diese Klasse beobachtet ständig die Veränderungen der Knoten. Falls ein oder mehrere Knoten die Limitierungen der Lastleistungen eingesetzt haben, soll man dies sofort bemerken und die Werte, die zu der SPS geschickt werden sollen, sollen sofort limitiert werden.

```
1  public void update(Observable changed, Object arg) {
2          // arg ist die ver?nderte Node instanz (hier Consumer).
3          Consumer c = (Consumer) arg;
4
5          // Synchronisiertes starten des Tageszyklus-Streamings
6          if (c.getStartSimulation()) this.setReadfile(true);
7
8          // Doch nicht beschr?nken?
9          if (!limit) return;
10
11         // Beschraenken der Consumierten Leistung
```

```
12          // Ueberschreitet eine istWirkleistung ihre sollWirkleistung?
13          boolean overTheTop = false;
14          float p[] = new float[3];
15          float u[] = new float[3];
16          for (int i = 0; i < 3; i++) {
17          //if (Math.abs(consumers[i].getIstWirkleistung()) >
18         Math.abs(consumers[i].getSollWirkleistung()) )
19                  if (consumers[i].getIstWirkleistung() < consumers[i].getSollWirkleistung() ) {
20                          overTheTop = true;
21                          p[i] = (float) consumers[i].getSollWirkleistung();
22                  } else {
23                          p[i] = (float) consumers[i].getIstWirkleistung();
24                  }
25                  u[i] = generateUfromP(p[i]);
26          }
27          if (overTheTop) {
28                  try {
29                          // Ui und Pi schicken.
30                          List<Short> s = new ArrayList<Short>();
31                          this.addValueToList(s, u[1]); this.addValueToList(s, p[1]);
32                          this.addValueToList(s, u[2]); this.addValueToList(s, p[2]);
33                          this.addValueToList(s, u[3]); this.addValueToList(s, p[3]);
34                          this.getMbCtl().writeMultipleRegisters(referenceNr, s);
35                  } catch(MessageParsingException e) {
36                          e.printStackTrace();
37                  }
38          }
39
40 }
```

2.16 SPS-Implementierung Consumer

Die SPS-Implementierung wurde für drei Verbraucher durchgeführt. Daher hat die SPS drei Ausgänge zum Senden der Werte an die Verbraucher und drei Eingänge zum Lesen der Istwerte. In diesem Fall werden Verbraucher durch die Spannung gesteuert.

- **Von der SPS zum Verbraucher:**

  Die SPS bekommt diese Spannungswerte von Java. Die Werte werden in einem Register abgelegt, wo beide Seiten (Java und SPS) drauf zugreifen können. Dafür hat die SPS eine globale Variable namens `inputValues` unter *Ressourcen* hinterlegt, die ein Array darstellt. Die Zahlen sind in `FLOAT.FLOAT` sind in CoDeSys als `REAL` bezeichnet.

  ```
  inputValues AT %MD0 : ARRAY [0..5] OF REAL;
  ```

  Weiter muss die SPS wissen, welche Werte zu erwarten sind. Denn es soll im Rahmen unserer Experimente möglich sein, dass alle Verbraucher einen gleichen Wert oder unterschiedliche Werte erhalten. Aus diesem Grund wurde eine Variable `modi` definiert. `modi` kann den Wert entweder 1 oder 0 annehmen, der von der Java-Seite bestimmt wird und daher eine Merkeradresse hat.

  ```
  modi AT %MW30: WORD;
  ```

  - 1 steht für die Weiterleitung unterschiedlicher Werte an die Verbraucher
  - 0 bedeutet; gleicher Wert für alle drei Verbraucher.

  Anschliessend werden alle Ausgänge der SPS mit dem entsprechenden Wert belegt, je nachdem wie der `modi` umgeschaltet wird.

  Die Implementierung sieht folgendermassen aus:

  ```
  IF (modi  = 1) THEN
  ```

```
IF (inputValues[0] < 3.0) AND (inputValues[1] < 800) THEN

Consumer1_Usoll := REAL_TO_WORD(inputValues[0]/10*32768);
Consumer1_PSoll:= inputValues[1];
END_IF

IF (inputValues[2] < 3.0) AND (inputValues[3] < 800) THEN

Consumer2_Usoll :=REAL_TO_WORD(inputValues[2]/10*32768);
Consumer2_PSoll:=inputValues[3];
END_IF

IF (inputValues[4] < 3.0) AND (inputValues[5] < 800) THEN

Consumer3_Usoll := REAL_TO_WORD(inputValues[4]/10*32768);
Consumer3_PSoll := inputValues[5];
END_IF
END_IF

IF (modi  = 0) THEN

tempU := inputValues[0];
tempP := inputValues[1];

Consumer1_Usoll := REAL_TO_WORD (inputValues[0]/10*32768);
Consumer2_Usoll := REAL_TO_WORD (inputValues[0]/10*32768);
Consumer3_Usoll := REAL_TO_WORD (inputValues[0]/10*32768);

END_IF
```

- **Weitere Möglichkeiten**
  - **Vom Verbraucher zur SPS**

    Unter *Ressourcen* gibt es eine globale Variable `sendValues`, die dazu dient, die Istwerte von den Verbrauchern an Java zu schicken.

    ```
    sendValues AT %MD6 : ARRAY [0..5] OF REAL;
    ```

    Im Gegensatz zu den Sollwerten sind die Istwerte hier nicht die Spannungen, sondern Stromwerte. Daraus können die Istleistungen (die tatsächlichen Leistungsbezüge) berechnet und in `sendValues` mit dem Iststromwert an Java geschickt werden.

    Der Code dafür ist schon implementiert. Wenn kombiniert zu einem späteren Zeitpunkt ein Messgerät (Schnittstelle RS 232) angeschlossen wird, kann die Java-Seite die Istwerte direkt sehen.
  - **Regelung**

    Während unserer Experimente haben die Geräte genau funktioniert. Daher war es nicht nötig einen Regler einzubauen. Falls Unregelmässigkeiten auftauchen würden, steht ein

Baustein namens PID-Regler in der Bibliothek `util.lib` zur Verfügung. Dieser Regler ist leicht zu einem PI-Regler umwandelbar und die Ausgänge von der SPS werden dann als Stellglieder genutzt.

Kapitel 3

Hardware-Simulation

Die Hardware-Simulations-Gruppe hat sich damit Beschäftigt, Strukturen, Programme und Verfahren zu Entwickeln und zu Implementieren, die notwendig sind, um ein Energienetz zu Automatisieren und zu Steuern.

Die Herausforderung liegt darin, dass zu jeder Zeit ein stabiler Netz-Zustand gehalten werden muss und dies über ein Wide-Area-Network mit dezentral verwalteten elektrischen Netz-Komponenten.

Hierzu war es notwendig einzelne Hardware-Komponenten, wie Generatoren, Leitungen und Verbraucher einerseits in ihren physikalischen Eigenschaften nachzubilden und andererseits ihr Verhalten zu Simulieren, um die entwickelten Programme zu verifizieren.

Die Simulation ermöglichte es außerdem gefahrlos größere Netze zu simulieren, als sie real zur Verfügung standen.

Die nachfolgenden Abschnitte widmen sich nun der Modellierung eines verteilten rechnergestützten Netzmodells. Die Abschnitte sind zunächst zweigeteilt in einen kleinen einführenden Theorie- und einen Praxis-Teil, der den Rest umfasst.

Zuerst werden im Abschnitt *Theoretische Grundlagen* (3.1) die Grundlagen dargestellt, Strukturen geschaffen und Verfahren behandelt, die der praktischen Implementierung zu Grunde liegen. Hierzu gehören die Bestandteile eines elektrischen Netzes (Produktions-, Übertragungs- und Verbrauchskomponenten), die Betriebsmodi (Inselnetz- und Verbundbetrieb), sowie die Verfahren zur Netzberechnung (Lastflussberechnung bzw. State Estimation).

Anschließend wird im Abschnitt *Netzdarstellung und Implementierung* (3.2) auf die konkreten Java-Strukturen eingegangen, die entwickelt wurden, um ein elektrisches Netz abzubilden. Es werden die einzelnen Bestandteile des Netzes (Node und Subklassen SyncNode, AsyncNode, Inverter, Consumer), des gesamten Netzes (PowerNet) und der Berechnungsverfahren (AbstractBalancer, LFBalancer, WLBalancer) vorgestellt.

Im Abschnitt *Bilanzierungsaufgaben* (3.3) wird dann auf die Umsetzung der Netzberechnungen in

Java eingegangen. Es werden verschiedene Formen der Bilanzierung behandelt (Wirkleistungsbilanzierung, Blindleistungsbilanzierung, virtueller Generator und Lastflussestimation).

Der *UPnP* Abschnitt (3.4) behandelt das Netzwerk-Protokoll, das es ermöglicht die einzelnen Netzbestandteile über ein Rechnernetz verteilt auszuführen und dynamisch einzubinden. Es wird erklärt, wie die so genannten UPnP-Devices (verteilte Geräte) und UPnP -ControllPoints (Kontroll-Punkte/-Geräte) eingerichtet werden können, um ein elektrisches Netz verteilt über ein Rechnernetz zu repräsentieren. Beendet wird dieser Abschnitt durch die Darstellung der Ergebnisse eines UPnP -Belastungstests, in denen auf Testfälle eingegangen wird, mit denen die Qualität der Datenübertragung im verteiltem Netzmodell überprüft werden soll.

Schließlich wird im Abschnitt über die *Hardwaresimulation* (3.9) auf die Simulation eines verteilten Energienetzes eingegangen. Kernstück bildet die ContainerGUI, die zur Erzeugung und Überwachung einer Netzmodellinstanz eine grafische Oberfläche zur Verfügung stellt.

3.1 Theoretische Grundlagen

3.1.1 Lastflussberechnung

Im Grunde geht es um die Lösung eines Gleichungssystems. Im ersten Abschnitt wird auf das Newton-Raphson-Verfahren eingegangen und mit einem einfachem Beispiel erklärt. Das Newton-Raphson Verfahren wird zu Berechnung des Lastflusses benutzt. Daraufhin werden lediglich die einzelnen Komponenten und Systemparameter erklärt, die für eine Berechnung mit dem Newton-Raphson Verfahren nötig sind, und zu einem Gleichungssystem zusammengesetzt.

Newton-Raphson-Verfahren

Das Newton-Raphson Verfahren oder die Newton-Raphson Methode sind Standardverfahren für die Lösung nichtlinearer Gleichungen und Gleichungssysteme. Das Verfahren ist benannt nach Sir Isaac Newton und Joseph Raphson. Im Allgemeinen soll die Gleichung $f(x) = 0$ für eine stetig differenzierbare Funktion f gelöst werden. Im Folgenden wird das Verfahren im eindimensionalen Raum beispielhaft erklärt. Das lösen von nichtlinearen Gleichungssystemen dient später der Berechnung des Betriebszustandes eines elektrischen Energieübertragungssystems indem für die einzelnen Knoten Gleichungen aufgestellt und mittels des Newton-Raphson Verfahrens gelöst werden.

Als Beispiel wird die Berechnung der Wurzelfunktion gewählt. Diese leitet sich aus der Gleichung $f(x) = 0$ mit $f(x) = \frac{1-a}{x^2}$ her, wobei hier $\sqrt{a}$ iterativ bestimmen werden soll. In Abbildung 3.1.1 wird die Funktion graphisch dargestellt.

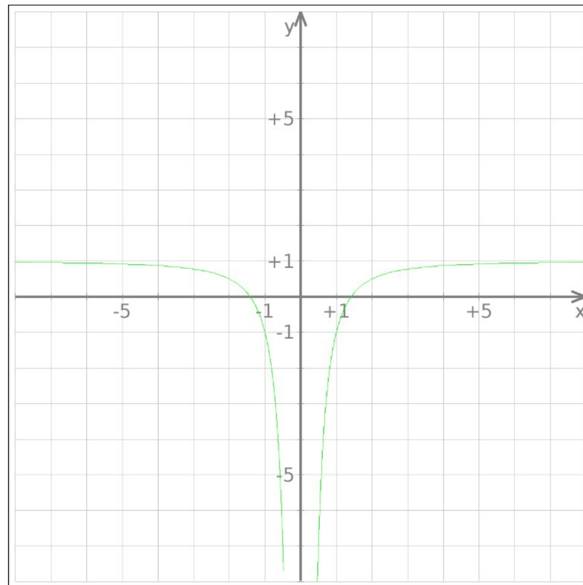


Abbildung 3.1: Beispiel für die Anwendung eines Newton-Raphson Verfahren

Die Funktion in Abbildung 3.1.1 schneidet für $x > 0$ die X-Achse bei $x = \sqrt{a}$. Mittels des Newton-Raphson Verfahren wird der Schnittpunkt iterativ errechnet. Dazu sei x_0 ein beliebiger Startwert (in diesem Beispiel sollte er größer Null gewählt werden, da die Wurzelfunktion nur auf positive reelle Zahlen abgebildet wird). Als erstes wird die Tangente an dem Punkt $(x_0, f(x_0))$ bestimmt. Diese setzt sich aus der Steigung $f'(x_0)$ und der Abweichung an der Y-Achse $f(x_0)$ zusammen. Somit ergibt sich als Tangentenfunktion :

$$t(x) = f'(x_0) * (x - x_0) + f(x_0)$$

Der nächste Schritt ist die Berechnung des Tangentenschnittpunktes mit der X-Achse x_1. An diesem Punkt wird die nächste Tangente an die Funktion f gelegt um mit dem Verfahren fortzufahren. Über n Iterationen ergibt sich dann folgender Ablauf

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

In dem oben angegebenen Beispiel mit der Funktion $f(x) = 1 - \frac{a}{x^2}$ mit $f'(x) = \frac{2a}{x^3}$ ergibt sich

$$x_{n+1} = x_n - \frac{1 - \frac{a}{x_n^2}}{\frac{2a}{x_n^3}} = x_n - \frac{x_n^3}{2a} + \frac{x_n}{2} = \frac{x_n}{2}\left(3 - \frac{x_n^2}{a}\right)$$

Der Verlauf der Berechnung ist in Tabelle 3.1 für verschiedene Werte zusammengefasst.

Bereits nach wenigen Iterationen erreicht das Verfahren annäherungsweise Werte der Wurzelfunktion. Abbildung 3.1.1 skizziert das Verfahren noch einmal graphisch.

n	x_n mit $a = 2$	x_n mit $a = 3$	x_n mit $a = 5$
0	$1,5$	2	3
1	$1,4$	$1,6$	$1,8$
2	$1,4141$	$1,72$	$2,1$
3	$1,41421355$	$1,73203$	$2,22$
4	$1,4142135623731$	$1,1320508074$	$2,23601$

Tabelle 3.1: Beispielverlauf des Newton-Raphson Verfahren

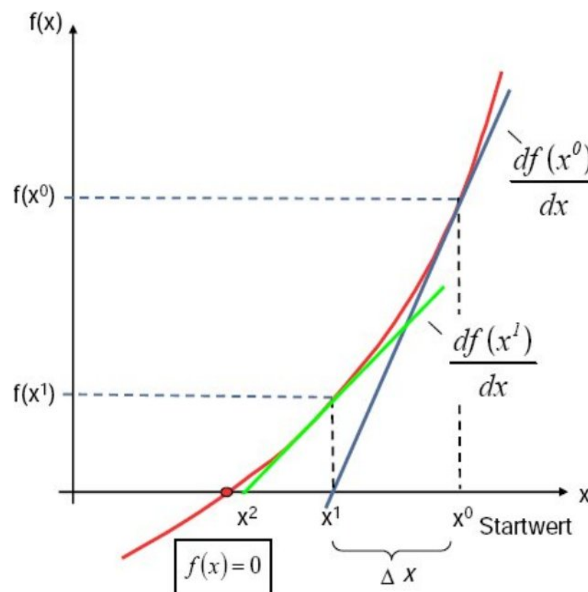


Abbildung 3.2: Graphischer Ablauf des Newton-Raphson Verfahren

Konvergenzverhalten Eine wichtige Eigenschaft bei der Anwendung des Newton-Raphson Verfahren ist das Konvergenzverhalten. Dabei existieren Szenarien in dem das Verfahren um den Schnittpunkt divergiert und somit kein konkretes Ergebnis liefert.

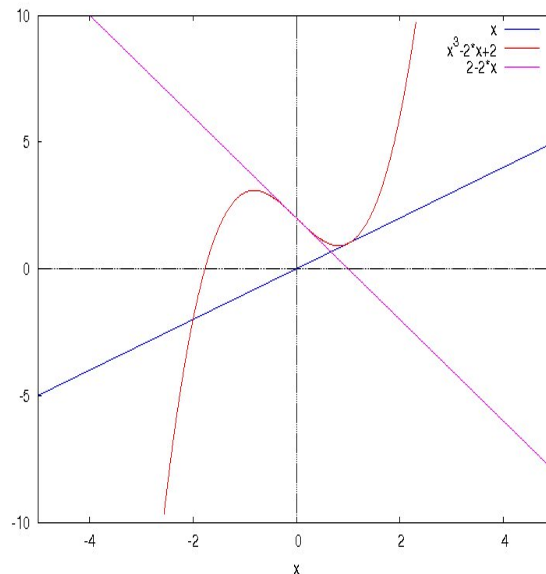


Abbildung 3.3: Kein Konvergenzverhalten beim Newton-Raphson Verfahren

Ein Szenario in dem das Newton-Raphson verfahren nicht konvergiert ist die Funktion

$$f(x) = x^3 - 2x + 2$$

mit

$$f'(x) = 3x^2 - 2.$$

An dem Punkt $x = 1$ liegt der neue Schnittpunkt der Tangente an f. Diese Tangente schneidet die X-Achse bei $x = 0$. Die Tangente an $f(0)$ schneidet die X-Achse allerdings wieder bei $x = 1$ womit das Verfahren nicht konvergieren würde und immer zwischen $x = 1$ und $x = 0$ wechselt. Abbildung 3.1.1 zeigt die Funktion f mit den beiden Tangenten an $f(1)$ und $f(0)$. Die Schnittpunkte der Tangenten mit der X-Achse liegen auf den jeweiligen X-Werten der Schnittpunkte mit der Funktion f.

Anwendung auf elektrische Energieübertragungsnetzwerke

Das mathematische Netz wird in Knoten und Kanten eingeteilt. Knoten können, je nach steuerbarem Einfluss sogenannte PQ- oder PU-Knoten sein. Ein Referenzknoten existiert nur im mathematischen Netz und gibt Bilanzierungsgrößen wie Wirkleistungsverlust bzw. Wirkleistungsgewinn an. Die Kanten, die die einzelnen Knoten miteinander verbinden, werden in Form einer Matrix, der sogenannten Admitanzmatrix angegeben. Die Admitanzmatrix stellt die jeweiligen Widerstände in komplexer Form dar, wobei die Zeilen den Quellknoten und die Spalten den Zielknoten entsprechen. Die erfassten Werte für die Knoten und Kanten ergeben das zu lösende Gleichungssystem.

Alle Knoten in einem Netz haben die gleichen Parameter, jedoch unterscheiden sie sich nach gegebenen (bzw. regelbaren) und unbekannten Parametergrößen. Die betrachteten Parameter lauten wie folgt:

- Eingespeiste/Entnommene Wirkleistung im Knoten i:
  $P_i = \sum_{j=1}^{n} [e_i * (e_j * g_{ij} - f_j * b_{ij}) + f_i * (f_i * g_{ij} + e_j * b_{ij})]$
- Eingespeiste/Entnommene Blindleistung im Knoten i:
  $Q_i = \sum_{j=1}^{n} [f_i * (e_j * g_{ij} - f_j * b_{ij}) + e_i * (f_j * g_{ij} + e_j * b_{ij})]$
- Spannungsbetrag im Knoten i:
  $U_i^2 = e_i^2 + j * f_i^2$
- Spannungswinkel im Knoten i:
  $\delta_i = \arctan \frac{e_i}{f_i}$

Hierbei repräsentieren e_i und f_i den Real- bzw. Imaginärteil des Knotens, sowie g_{ij} und b_{ij} selbiges für die Admitanzen (siehe Seite 100). Wie sich diese Formeln ergeben ist an dieser Stelle nicht wichtig. Klar ist allerdings, dass die dargestellten Variablen für P_i, Q_i, U_i und δ_i auch als Matrix geschrieben werden können. Dazu im späteren Verlauf aber mehr.

PQ-Knoten PQ-Knoten repräsentieren alle Knoten ohne Einspeisung und ohne vorgegebene Spannung. An einem PQ-Knoten sind Wirk- und Blindleistungseinspeisung bzw. Wirk- und Blindleistungsaufnahme vorgegeben. PQ-Knoten simulieren in der Regel Verbraucher im elektrischen Netz.

PU-Knoten PU-Knoten haben einen vorgegebenen Leistungs- und Spannungsbetrag. Vorgegebene Werte sind die Spannung und die Wirkleistung. Meistens handelt es sich bei PU Knoten um Kraftwerkseinspeisungen. Zu beachten ist für die Berechnung nach dem Newton-Raphson-Verfahren, das die Blindleistungsgrenzen angegeben werden müssen. Werden diese Grenzen von der Berechnung überschritten kann ein PU-Knoten in einen PQ-Knoten umgewandelt werden.

Referenzknoten In einem mathematischem Netz muss mindestens ein Referenzknoten enthalten sein. Der Referenz- oder auch Bilanzknoten gibt die Differenz zwischen der eingespeisten Wirk- sowie Blindleistung wieder. Mit berücksichtigt werden hierbei die auf dem Netz anfallenden Leitungsverluste.

Knotenart	gegeben	gesucht
Referenzknoten	U, δ_U	P, Q
Einspeiseknoten (PV-Knoten)	U, P	δ_U, Q
Lastknoten (PQ-Knoten)	P, Q	U, δ_U

Tabelle 3.2: Gegebene und Gesuchte Werte der einzelnen Knotenarten

Admitanzmatrix Die Admitanzmatrix gibt die Verbindungen innerhalb des Netzes wieder. Die Kanten werden dabei durch den Kehrwert des komplexen Widerstandes in einer Matrix gespeichert. Dabei gibt die Zeile den Quell- und die Spalte den Zielknoten an.

$$\overline{Y} = \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{pmatrix}, y_{ij} = g_{ij} + j * b_{ij} \tag{3.1}$$

Um die zu lösende Gleichung aufzustellen muss noch die Matrix J der partiellen Ableitungen in Abhängigkeit von e_i und f_i eingeführt werden. Die Ableitungen bilden sich aus den Knotengleichungen für die PU-, PQ- und Referenzknoten. Wie die Gleichungen dort aufgestellt werden und wie die Ableitungen gebildet werden, wird später im Kapitel zur nicht-linearen Estimation noch genauer behandelt. ΔF beinhaltet die unbekannten Funktionale e_i und f_i und ΔX beschreibt die gegebenen Messwerte für P_i, Q_i und U_i.

$$J * \Delta F = \Delta X \tag{3.2}$$

$$\Delta F = J^{-1} * \Delta X \tag{3.3}$$

Die Matrix J kann in sechs Teilmatrizen $J_1, \ldots, J_6$ unterteilt werden. J_1 und J_2 beschreiben die Ableitungen für die Wirkleistung, J_3 und J_4 die Blindleistung und J_5 und J_6 entsprechen den Ableitungen für die Spannungsgeregelten Knoten. Die Matrix J hat also folgenden Aufbau :

$$J = \begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \\ J_5 & J_6 \end{bmatrix} \tag{3.4}$$

Die Teilmatrizen $J_1, \ldots, J_6$ ergeben sich aus den Beziehungen innerhalb des elektrischen Netzwerks. Für die Berechnung wird ein Startwert für den Vektor ΔF gewählt und eine erste Iteration gestartet. Die errechneten Werte werden eingesetzt und die für P, Q und U erhaltenen Werte weichen noch weit von einem akzeptabelen Ergebnis ab. Nach aber maximal vier bis fünf Iterationen ist die Abschätzung in der Regel so genau, dass keine weiteren Berechnungen mehr nötig sind.

Ein einfaches Beispiel Zum besseren Verständnis wird nur ein einfaches Netz mit drei Knoten analysiert und mittels des Newton-Raphson Verfahren gelöst. Das Beispielnetz besteht, wie schon erwähnt, aus drei Knoten, einem Referenzknoten, einer Einspeisung (also einem PU-Knoten) und einem Konsumenten (PQ-Knoten). Dabei zeigt Tabelle 3.1.1 welche Werte für die einzelnen Knoten anliegen.

Knoten	Einspeisung		Last		Spannung	
	P_G	Q_G	P_L	Q_L	U_i	δ_i
1 (Referenzknoten)	?	?	$1,0$	$0,5$	$1,0$	$0,0$
2 (PU-Knoten)	$1,5$	?	$0,0$	$0,0$	$1,0$	?
3 (PQ-Knoten)	$0,0$	0	$1,0$	?	?	?

Tabelle 3.3: Beispielwerte für eine Lastflussberechnung

Der Index G steht für die Generator- also für die erzeugte Leistung des Knotens. Der Index stellt die Last, also die verbrauchte Leistung dar. Für die Netztopologie wird die oben beschriebene Admitanzmatrix eingesetzt. Diese hat folgende Form:

$$\overline{Y}_k = j \begin{pmatrix} -20 & 10 & 10 \\ 10 & -20 & 10 \\ 10 & 10 & -20 \end{pmatrix}$$

Die positiven Werte geben an, das eine Verbindung von Knoten i (Spalte) zu Knoten j (Zeile) besteht. Die Verbindungen von Knoten i zu Knoten j mit $i = j$ sind die Summen der eingehenden Admitanzen der Leitungen, multipliziert mit -1. Somit wird ersichtlich, das alle Knoten untereinander verbunden sind und das zwischen allen Knoten die gleiche Admitanz herrscht.

Das weitere Vorgehen besteht aus dem aufstellen der Gleichungen. Dabei sind ein paar Umformungen nötig, die ein genaueres Verständnis der physikalischen Vorgänge im elektrischen Leistungsaustausch vorraussetzen. Diese Umformungen werden übersprungen, und direkt mit der aufstellung der Jacobischen Matrix J fortgefahren. Die Gleichungen für die Wirkleistung P_1, P_2 und P_3 sehen demnach folgendermaßen aus:

$$P_1 = -10f_2 - 10f_3$$
$$1,5 = -10e_2f_3 + 10f_2 + 10e_3f_2$$
$$-1,0 = -10e_3f_2 + 10f_3 + 10e_2f_3$$

Die Gleichungen für die Blindleistung Q_1, Q_2 und Q_3:

$$Q_1 = 20 - 10e_2 - 10e_3$$
$$Q_2 = -10e_2 + 20 - 10e_2e_3 - 10f_2f_3$$
$$-1,0 = 20e_3^2 - 10e_2e_3 - 10f_2f_3 + 20f_3^2 - 10e_3$$

Und als letztes die Gleichung für die Spannung an Knoten 2 ($(U_2)^2$) :

$$1,0 = e_2^2 + f_2^2$$

In Tabelle 3.1.1 ist bereits zu sehen, welche Werte schon vorhanden, und welche noch gesucht werden. P_2 und P_3 sind demnach bekannt, genauso Q_3 und U_1, wobei U_2 und U_3 mittels des Real- bzw. Imaginärteils e_2, f_2, e_3 und f_3 errechnet werden können. Es wird also die Wirkleistung für Knoten eins (P_1), die Blindleistung für Knoten eins und zwei (Q_1, Q_2) sowie die Komponenten der komplexen Spannnung e_2, e_3, f_2 und f_3 gesucht.

Es muss nun im folgenden die Jacobi Matrix J aufgestellt werden. Diese wurde in $J_1, \ldots, J_6$ unterteilt, wobei J_1 und J_2 die Gleichungen für die Wirkleistung enthält. Da hier aber die Ableitung der Gleichungen von J gesucht wird, damit das Newton-Raphson Verfahren angewendet werden kann, werden die partiellen Ableitungen nach f_2, f_3, e_2 und e_3 gebildet. Dadurch ergibt sich für J_1 und J_2 folgendes Bild:

$$J_1 = \begin{bmatrix} 10 + 10e_3 & -10e_3 \\ -10e_3 & 10e_2 + 10 \end{bmatrix}$$
$$J_2 = \begin{bmatrix} -10f_3 & 10f_2 \\ 10f_3 & -10f_2 \end{bmatrix}$$

Die Matrizen für die Blindleistung und Referenzspannung folgen dem gleichen Prinzip. Die vollständige Gleichung auf die das Newton-Raphson Verfahren angewendet wird, sieht folgendermaßen aus:

$$\begin{bmatrix} 10+10e_3 & -10e_3 & -10f_3 & 10f_2 \\ -10e_3 & 10e_2+10 & 10f_3 & -10f_2 \\ -10f_3 & 40f_3-10f_2 & -10e_3 & 40e_3-10e_2-10 \\ 2f_2 & 0 & 2e_2 & 0 \end{bmatrix} \begin{bmatrix} \Delta f_2 \\ \Delta f_3 \\ \Delta e_3 \\ \Delta e_2 \end{bmatrix} = \begin{bmatrix} \Delta P_2 \\ \Delta P_3 \\ \Delta Q_3 \\ \Delta U_2 \end{bmatrix}$$

Die Jacobi-Matrix beinhaltet also die partiellen Ableitungen, der Vektor ΔF die Differenz zwischen dem letzten Iterationswert und dem neuen Achsenschnittpunkt. Gestartet wird das Verfahren nur mit $e_i = 1,0$ und $f_i = 0,0$ mit $i \in \{1,2,3\}$. Die erste Iteration verspricht bekanntlich nicht sehr viel. Mittels Umformung werden jetzt die nächsten Wert für f_2, f_3, e_2 und e_3 ermittelt:

$$f_2 = 0,066$$

$$f_3 = -0,016$$

$$e_2 = 0$$

$$e_4 = -0,05$$

Die Werte für die neue Iterationen errechnen sich nun aus den Werten der letzten Iteration und den gerade errechneten:

$$f_2 = 0 + 0,066 = 0,066$$

$$f_3 = 0 - 0,016 = -0,016$$

$$e_2 = 1 + 0 = 1,0$$

$$e_3 = 1 - 0,05 = 0,95$$

Die weiteren Iterationen werden jetzt nicht mehr betrachtet, da sich das Verfahren von nun an wiederholt. Jedoch ist anzumerken, das bereits nach drei Iterationen, bei geeigneter Wahl der Startwerte, akzeptabele Werte für f_2, f_3, e_2 und e_3 zu erwarten sind.

$$f_2 = 0,0681$$

$$f_3 = -0,0181$$

$$e_2 = 0,9889$$

$$e_3 = 0,9403$$

Mit diesen Werten lassen sich die Knotenleistungen errechnen

$$P_2 = 1,49957$$

$$P_3 = -0,99959$$

$$Q_3 = -0,99211$$

sowie der Spannungswert für Knoten zwei

Knoten	Einspeisung		Last		Spannung	
	P_G	Q_G	P_L	Q_L	U_i	δ_i
1 (Referenzknoten)	$0,50$	$1,07$	$1,0$	$0,5$	$1,0$	$0,0$
2 (PU-Knoten)	$1,5$	$0,61$	$0,0$	$0,0$	$1,0$	$3,9$
3 (PQ-Knoten)	$0,0$	0	$1,0$	$1,0$	$0,945$	$-1,1$

Tabelle 3.4: Errechnete Werte für eine Lastflussberechnung

$$(U_2)^2 = 0,9808$$

In Tabelle 3.1.1 sind alle Netzparameter eingetragen. Das Verfahren konnte bereits nach der dritten Iteration ein annehmbares Ergebnis vorweisen. Dieses Beispiel ist natürlich nur ein sehr grober Überriss für dieses komplexe Verfahren. Im Zusammenhang mit dem RealDezent Projekt wurde die Lastflussberechnung für Bilanzierungsaufgaben eingesetzt. Für diese Bilanzierungsaufgaben war es nötig, die Wirkleistungs- und Blindleistungsdifferenz inklusive der, auf dem Netz anfallenden Leitungsverluste, zu errechnen.

3.1.2 State Estimation

Eine Zustandsbestimmung, auch Ausgleichsrechnung genannt oder englisch State Estimation [5], wird benötigt, um aus einem gemessenen inkonsistenten Netzzustand einen konsistenten zu schätzen. Das heißt, es werden aus einer Menge widersprüchlicher (fehlerhafter/verrauschter) Messwerte eine Menge von geschätzten Messwerten abgeleitet, die nicht widersprüchlich sind, sondern deren (netzspezifischen) funktionalen Zusammenhänge gültig sind. Schätzen bedeutet hier, eine Wertemenge abzuleiten, deren Wahrscheinlichkeit, dass sie dem wahren Netzzustand entspricht möglichst hoch ist [6].

Im Rahmen der Erprobung wurden Netze mit unterschiedlicher Topologie modelliert (leiterförmig, stern, etc.). Ein Teil der Parameter waren dabei gegeben und wurden zum Teil mit Hilfe der Lastflussberechnung berechnet. Diese berechneten Werte der Lastflussberechnung wurden verrauscht um Messwert-Eingaben realer Netze zu simulieren. Anschließend wurde der Estimations-Algorithmus angewendet und die geschätzten Werte mit den unverrauschten werden verglichen,
um einen überblick über die genauigkeit der Schätzung zu erhalten.

Im RealDezent-System übernimmt die Estimation die Bereinigung der von den einzelnen Verbrauchern und Produzenten gemessenen Parameter und stellt diese, zu Regelungszwecken, zur Verfügung.

Lineare State-Estimation

Bei der linearen State-Estimation werden Zustandswerte zu Messwerten in eine lineare funktionale Abhängigkeit gebracht, um aus den Messwerten Rückschlüsse über den Systemzustand zu ziehen. Es wird also ein lineares Modell benutzt:

$$\tilde{z} = H\vec{x} + \vec{v}$$
$\tilde{z} \hat{=}$ Fehlerbehafteter Messvektor
$H \hat{=}$ Messmatrix
$\vec{x} \hat{=}$ Netzzustand
$\vec{v} \hat{=}$ Gaussscher Zufallsvektor mit Erwartungswert Null

Zum Aufruf der implementierten Methode wird die Messmatrix H, der Messvektor $\tilde{z}$, sowie ein korrespondierender Vektor $\vec{\sigma}$ mit Messvarianzen übergeben. Um den Systemzustand $\vec{x}$ zu berechnen wird zunächst eine Matrix R^{-1}, mit den Varianzen σ_i auf der Hauptdiagonalen erstellt. Anschließend kann die wahrscheinlichste Lösung gefunden werden durch:

$$\hat{\vec{x}} = (H^T R^{-1} H)^{-1} H^T R^{-1} \tilde{z}$$

Die lineare State-Estimation wurde zunächst implementiert und anhand kleinerer Beispiele und verrauschten Messwerten mit Erfolg erprobt. Es ergeben sich allerdings zweierlei Probleme mit diesem Modell:

1. Die Zusammenhänge in Leitungsnetzen sind meist quadratisch oder gar trigonometrisch.
2. Eine automatische erzeugung einer Messmatrix aus und für beliebige Netz-Topologien kann schwierig sein.

Nicht-Lineare State-Estimation

Die nicht-lineare State-Estimation zeichnet sich im Gegensatz zur linearen State-Estimation durch ein Modell aus, das einen nicht-linearen Zusammenhang zwischen Zustand und Messwerten beschreibt.

Zustands-Estimation mittels Newton-Raphson-Verfahren

Es wird versucht den Systemzustand zu Schätzen, indem zunächst mit einer initialen Annahme über den Systemzustand gestartet wird und es wird dann versucht diesen stetig zu verbessern.

$\hat{\vec{x}} = (\hat{e}_1, \hat{f}_1, ..., \hat{e}_n, \hat{f}_n)^T$ ist der (geschätzte) Zustands-Vektor, wobei e_i und f_j für Real- und Imaginärteil der komplexen Spannung am Knoten i stehen. $\vec{z}$ steht für den Messvektor, hier z.B. $\vec{z} = (U_1^2, P_1, Q_1, ..., U_n^2, P_n, Q_n)^T$, mit U_i^2, P_i, Q_i, die für die entsprechenden Messgrößen am Knoten i stehen.

$\vec{z}$ ist gegeben durch die am System durchgeführten Messungen oder durch simulierte Werte. $\hat{\vec{x}} = (1, 0, ..., 1, 0)^T$ ist der initiale (geschätzte) Zustandsvektor, hierbei handelt es sich um einen möglichst guten, aber suboptimalen Startwert, der im Folgendem iterativ verbessert werden soll.

$\vec{F}$ ist eine Funktionsschar, so dass $z_i = F_i(\vec{x})$, oder kurz $\vec{z} = \vec{F}(\vec{x})$. Für die Funktionen gilt im allgemeinem $dim(Urbild) > dim(Bild)$ und sind somit nicht invertierbar. Es muss gelten $dim(\vec{z}) > dim(\vec{x})$, d.h. es gibt mehr Messwerte als zu bestimmende Zustandsgrößen. Außerdem berechnet genau eine Funktion einen Messwert (F_i berechnet z_i). Die Funktion i stellt also genau den Zusammenhang zwischen dem Zustandsvektor und dem Messwert i dar.

Der Algorithmus stellt sich folgendermaßen dar:

1. Abstand zu den Messwerten $\vec{z}$ bei Zustand $\hat{\vec{x}}$ berechnen:

   I. $\Delta\vec{z} := \vec{z} - F(\hat{\vec{x}})$

2. Stelle die Jakobi-Matrix f von F auf, mit Elementen f_{ij}:

   II. $f_{ij} := \frac{\partial F_j}{\partial x_i}(\hat{\vec{x}})$

   Mit der Jakobi-Matrix f gilt nun:

   III. $\Delta\hat{\vec{z}} = f * \Delta\hat{\vec{x}}$

3. Mittels III. lässt sich $\Delta\hat{\vec{x}}$ berechnen. Da f keine Quadratische Matrize ist, lässt sich keine normale Inverse (nach Gauss-Jordan) bilden. Es kann die Pseudoinverse mittels Singular-Value-Decomposition (siehe Seite 130) berechnet werden.

   IV. $\Delta\hat{\vec{x}} := \Delta\hat{\vec{z}} * f^-1$

4. Die Abschätzung für $\hat{\vec{x}}$ kann nun mit Hilfe von $\Delta\hat{\vec{x}}$ verbessert werden:

   V. $\hat{\vec{x}} := \hat{\vec{x}} + \Delta\hat{\vec{x}}$

5. Solange wir uns noch verbessern können, starten wir eine erneute Iteration ab 1. mit dem verbessertem Schätzwert $\hat{\vec{x}}$:

   VI. $Verbessert \iff \exists i : |\Delta\hat{x}_i| > \epsilon$

Am Ende erhält man also einen geschätzten Zustandsvektor $\hat{\vec{x}}$, der die gemessenen Messwerte mit möglichst kleinem Fehler reproduzieren kann. Weshalb angenommen werden kann, dass dieser geschätzte Zustandsvektor auch dem echtem Zustandsvektor nahe kommt.

Funktionale für die Nicht-Lineare State-Estimation in Leitungsnetzen

Nettoknotenwirkleistung P_i

$$P_i = e_i * \sum_{j=1}^{n} (g_{ij} * e_j - b_{ij} * f_j) + f_i * \sum_{j=1}^{n} (b_{ij} * e_j + g_{ij} * f_j)$$

Partielle Ableitungen von P_i:

$$\begin{aligned}
i \neq j : \frac{\partial P_i}{\partial e_j} &= e_i * g_{ij} + f_i * b_{ij} \\
\frac{\partial P_i}{\partial e_i} &= (2 * e_i * g_{ii} - b_{ii} * f_i) + \sum_{j=1, j\neq i}^{n} (g_{ij} * e_j - b_{ij} * f_j) + (b_{ii} * f_i) \\
&= 2 * e_i * g_{ii} + \sum_{j=1, j\neq i}^{n} (g_{ij} * e_j - b_{ij} * f_j) \\
&= 2 * e_i * g_{ii} - (g_{ii} * e_j - b_{ii} * f_j) + \sum_{j=1}^{n} (g_{ij} * e_j - b_{ij} * f_j) \\
&= e_i * g_{ii} + b_{ii} * f_j + \sum_{j=1}^{n} (g_{ij} * e_j - b_{ij} * f_j) \\
i \neq j : \frac{\partial P_i}{\partial f_j} &= -e_i * b_{ij} + f_i * g_{ij} \\
\frac{\partial P_i}{\partial f_i} &= (-b_{ii} * e_i) + (b_{ii} * e_i + 2 * f_i * g_{ii}) + \sum_{j=1, j\neq i}^{n} (b_{ij} * e_j + g_{ij} * f_j) \\
&= 2 * f_i * g_{ij} + \sum_{j=1, j\neq i}^{n} (b_{ij} * e_j + g_{ij} * f_j) \\
&= 2 * f_i * g_{ii} - (b_{ii} * e_i + g_{ii} * f_i) + \sum_{j=1}^{n} (b_{ij} * e_j + g_{ij} * f_j) \\
&= f_i * g_{ii} - b_{ii} * e_i + \sum_{j=1}^{n} (b_{ij} * e_j + g_{ij} * f_j)
\end{aligned}$$

Nettoknotenblindleistung Q_i

$$Q_i = f_i * \sum_{j=1}^{n} (g_{ij} * e_j - b_{ij} * f_j) - e_i * \sum_{j=1}^{n} (b_{ij} * e_j + g_{ij} * f_j)$$

Partielle Ableitungen von Q_i:

$$
\begin{aligned}
i \neq j : \frac{\partial Q_i}{\partial e_j} &= f_i * g_{ij} - e_i * b_{ij} \\
\frac{\partial Q_i}{\partial e_i} &= (g_{ii} * f_i) - (2 * e_i * b_{ii} + g_{ii} * f_i) - \sum_{j=1, j \neq i}^{n} (b_{ij} * e_j + g_{ij} * f_j) \\
&= -2 * e_i * b_{ii} - \sum_{j=1, j \neq i}^{n} (b_{ij} * e_j + g_{ij} * f_j) \\
&= -2 * e_i * b_{ii} + (b_{ii} * e_j + g_{ii} * f_i) - \sum_{j=1}^{n} (b_{ij} * e_j + g_{ij} * f_j) \\
&= -e_i * b_{ii} - g_{ii} * f_j - \sum_{j=1}^{n} (b_{ij} * e_j + g_{ij} * f_j) \\
i \neq j : \frac{\partial Q_i}{\partial f_j} &= -f_i * b_{ij} - e_i * g_{ij} \\
\frac{\partial Q_i}{\partial f_i} &= (g_{ii} * e_i - 2 * f_i * b_{ii}) + \sum_{j=1, j \neq i}^{n} (g_{ij} * e_j - b_{ij} * f_j) - (g_{ii} * e_i) \\
&= -2 * f_i * b_{ii} + \sum_{j=1, j \neq i}^{n} (g_{ij} * e_j - b_{ij} * f_j) \\
&= -2 * f_i * b_{ii} - (g_{ii} * e_i - b_{ii} * f_i) + \sum_{j=1}^{n} (g_{ij} * e_j - b_{ij} * f_j) \\
&= -f_i * b_{ii} - g_{ii} * e_i + \sum_{j=1}^{n} (g_{ij} * e_j - b_{ij} * f_j)
\end{aligned}
$$

(Quadratischer) Knotenspannungsbetrag U_i^2 $\boxed{U_i^2 = e_i^2 + f_i^2}$

Partielle Ableitungen von U_i^2:

$$
\begin{aligned}
i \neq j : \frac{\partial U_i^2}{\partial e_j} &= 0 \\
\frac{\partial U_i^2}{\partial e_i} &= 2 * e_i \\
i \neq j : \frac{\partial U_i^2}{\partial f_j} &= 0 \\
\frac{\partial U_i^2}{\partial f_i} &= 2 * f_i
\end{aligned}
$$

(Quadratischer) Knotenstrombetrag I_i^2

$$I_i^2 = \sum_{j=1}^{n} \left(g_{ij}^2 * (e_j^2 + f_j^2) + b_{ij}^2 * (e_j^2 + f_j^2) \right)$$

Partielle Ableitungen von I_i^2:

$$\frac{\partial I_i^2}{\partial e_j} = 2 * e_j * (g_{ij}^2 + b_{ij}^2)$$
$$\frac{\partial I_i^2}{\partial f_j} = 2 * f_j * (g_{ij}^2 + b_{ij}^2)$$

Leitungswirkleistungsfluss P_{ij}

$$P_{ij} = g_{ij} * (e_i^2 - e_i * e_j - f_i * f_j + f_i^2) + b_{ij} * (-e_i * f_j + f_i * e_j)$$

Partielle Ableitungen von P_{ij}:

$$\frac{\partial Pij}{\partial e_j} = g_{ij} * e_i + b_{ij} * f_i$$
$$\frac{\partial Pij}{\partial e_i} = g_{ij} * (2 * e_i + e_j) - b_{ij} * f_j$$
$$\frac{\partial Pij}{\partial f_j} = g_{ij} * f_i - b_{ij} * e_i$$
$$\frac{\partial Pij}{\partial f_i} = g_{ij} * (f_j + 2 * f_i) + b_{ij} * e_j$$

Leitungsblindleistungsfluss Q_{ij}

$$Q_{ij} = -b_{ij} * (e_i^2 - e_i * e_j - f_i * f_j + f_i^2) + g_{ij} * (-e_i * f_j + f_i * e_j)$$

Partielle Ableitungen von Q_{ij}:

$$\frac{\partial Qij}{\partial e_j} = -b_{ij} * e_i + g_{ij} * f_i$$
$$\frac{\partial Qij}{\partial e_i} = -b_{ij} * (2 * e_i + e_j) - g_{ij} * f_j$$
$$\frac{\partial Qij}{\partial f_j} = -b_{ij} * f_i - g_{ij} * e_i$$
$$\frac{\partial Qij}{\partial f_i} = -b_{ij} * (f_j + 2 * f_i) + g_{ij} * e_j$$

(Quadratischer) Leitungsstrom I_{ij}^2 $\boxed{I_{ij}^2 = (g_{ij}^2 + b_{ij}^2) * (e_i^2 - 2 * e_i * e_j + e_j^2 + f_i^2 - 2 * f_i * f_j + f_j^2)}$

$$\frac{\partial Iij^2}{\partial e_j} = (g_{ij}^2 + b_{ij}^2) * (-2 * e_i + 2 * e_j)$$
$$\frac{\partial Iij^2}{\partial e_i} = (g_{ij}^2 + b_{ij}^2) * (2 * e_i - 2 * e_j)$$
$$\frac{\partial Iij^2}{\partial f_j} = (g_{ij}^2 + b_{ij}^2) * (-2 * f_i + 2 * f_j)$$
$$\frac{\partial Iij^2}{\partial f_i} = (g_{ij}^2 + b_{ij}^2) * (2 * f_i - 2 * f_j)$$

3.1.3 Elektrisches Netz

Ein elektrisches Netz besteht aus verschiedenen Komponenten :

1. Produktionskomponenten, z.B :
   - Turbinen
   - Elektrische Maschinen
   - Regelung
2. Übertragungskomponenten, z.B :
   - Transformatoren
   - Übertragungsleitung
3. Verbrauchskomponenten, z.B :
   - z.B : Glühlampen, Kochherde, etc..

Produktionskomponenten

Die Hauptbestandteile der Produktionskomponenten sind Turbinen, elektrische Maschinen und Regelungen.

Turbinen Turbinen ermöglichen die Umwandlung potentieller Energie wie Wasser oder Wasserdampf in kinetische Energie (Rotationsenergie). Wenn Turbinen mit Generatoren gekoppelt sind, wird die Rotationsenergie in elektrische Energie umgewandelt.

Elektrische Maschinen Elektrische Maschinen ermöglichen eine Wandlung von elektrischer und mechanischer Leistung. Als Motor entzieht die Maschine elektrische Energie vom Netz und erzeugt mechanische Energie. Andernfalls, wenn die mechanische Leistung in die Maschine fließt und elektrische Leistung erzeugt wird, nennt man diese Maschine einen Generator. Jede rotierende elektrische Maschine besteht aus einem feststehenden Teil (Stator) und aus einem umlaufenden Teil (Rotor). Das

Prinzip der rotierenden elektrischen Maschine beruht auf dem Induktionsgesetz und der Kraftwirkung zwischen zwei Magnetfeldern. Es gibt eine Vielzahl unterschiedlicher Varianten elektrischer Maschinen z.B. Asynchronmaschine, Synchronmaschine und Gleichstrommaschine. In dieser Ausarbeitung werden nur die ersten beiden Maschinentypen behandelt.

1. Synchronmaschine
   Die Synchronmaschine ist die wichtigste elektrische Maschine für die Erzeugung elektrischer Energie. Sie wird in unterschiedlichen Leistungsbereichen wie zum Beispiel als Drehstrom-Generator in Kraftwerken eingesetzt. Die Drehzahl einer Maschine ist im Motorbetrieb fest durch die Netzfrequenz vorgegeben. Im Generatorbetrieb ist die Frequenz der abgegebenen Leistung durch die Drehzahl der Maschine bestimmt. Die Frequenz einer Synchronmaschine ist folgendermaßen definiert: f = p n [Hz] dabei ist p die Polpaarzahl und n die Umdrehung pro Minute.

   - Aufbau der Synchronmaschine
     Eine Synchronmaschine siehe Abb. 3.4 besteht hauptsächlich aus einem Stator und einem Rotor. Der Stator ist prinzipiell aus Blechen geschichteter Hohlzylinder mit Nuten aufgebaut, in denen eine Drehstromwicklung eingelegt ist. Der Rotor ist aus massivem Eisen mit Nuten, in denen eine Gleichstromwicklung eingelegt ist.

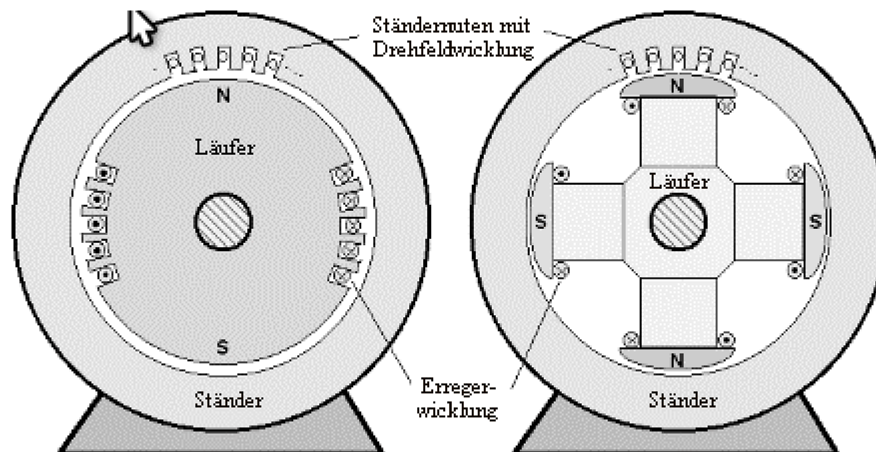


Abbildung 3.4: die Synchronmaschine

   - Betriebsart einer Synchronmaschine
     Im Motorbetrieb erzeugt der Erregerstrom im Motor ein rotierendes Magnetfeld. Bei der Einspeisung in die Statorwicklung aus dem Drehspannungssystem entsteht ein Ständerdrehfeld. Durch die magnetische Bindung beider Felder wird der Rotor synchron zum Ständerdrehfeld mitgedreht. Im Generatorbetrieb erzeugt der Erregerstrom ein rotierendes Magnetfeld, das in der Ständerwicklung ein Drehfeldsystem induziert.

2. Asynchronmaschine
   Die Asynchronmaschine wird als Motor oder als Generator eingesetzt. Sie wurde ca. im Jahre 1885 durch den Italiener Galileo Ferraris und den Kroaten Nicola Tesla erfunden. Die Asynchronmaschine hat einen einfacheren und robusteren konstruktiven Aufbau.

- Aufbau der Asynchronmaschine
  Die Asynchronmaschine siehe Abb. 3.5 besteht aus einem Rotor, der keine Gleichstromerregerwicklung zur Erzeugung eines Erregerfeldes trägt, sondern eine kurzgeschlossene Drehstromwicklung und einen Stator mit Drehfeldwicklung.

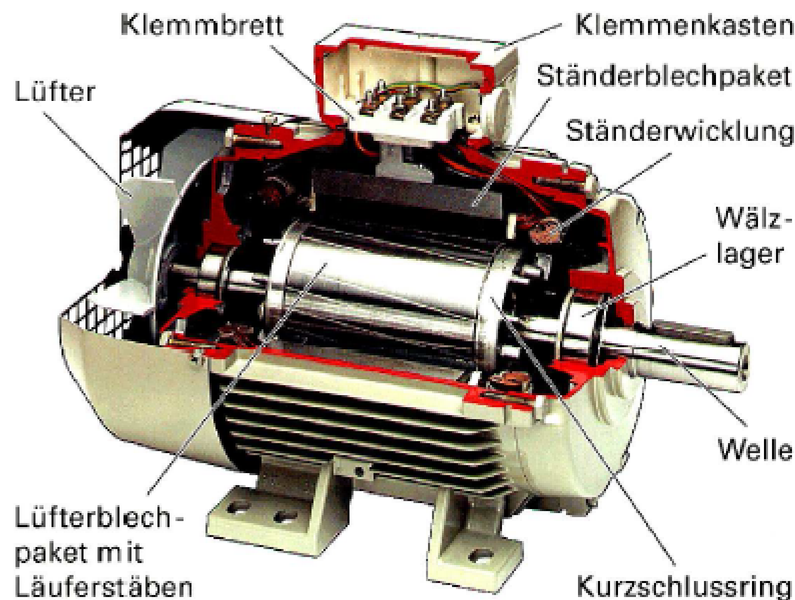


Abbildung 3.5: die Asynchronmaschine

- Betriebsart einer Asynchronmaschine
  Im Stator einer Asynchronmaschine wird bei der Speisung mit einem Drehspannungssystem ein Drehfeld erzeugt. Bewegt sich das Drehfeld über die Rotorwicklung hinweg, werden Spannungen induziert. Diese Spannungen erzeugen Ströme in der kurzgeschlossenen Rotorwicklung. Dies erzeugt ein magnetisches Drehfeld. Das Drehfeld des Stators und des Rotors Wechselwirken und somit entsteht ein Drehmoment. Um das Betriebsverhalten einer Asynchronmaschine zu beschreiben, definiert man den Schlupf $s = \frac{n_s - s}{n_s}$ wobei n_s die Synchrondrehzahl und n die Rotordrehzahl ist. Im Motorbetrieb ist die Rotordrehzahl geringer als die Synchrondrehzahl d.h. $n < n_s$ und $s > 0$ ist. Im Generatorbetrieb rotiert der Rotor oberhalb der Synchrondrehzahl d.h. $n > n_s$ und $s < 0$ ist. Im Bremsbetrieb sind die Richtungen von Rotordrehung und Ständerdrehfeld entgegengesetzt. Die Rotordrehzahl ist kleiner Null und die Maschine nimmt sowohl mechanische Leistung als auch elektrische Leistung auf.

Regelungskomponenten Bei den Regelungskomponenten wird prinzipiell zwischen der Frequenz- und Spannungsregelung unterschieden.

1. Frequenzregelung
   Bei der Frequenzregelung wird versucht das Gleichgewicht zwischen erzeugter und verbrauch-

ter Wirkleistung zu halten.

2. Spannungsregelung
   Um Verbrauchern eine konstante Spannung zu gewährleisten ist die Regelung der Netzspannung erforderlich. Dies geschieht durch die Erzeugung von Blindleistung in Kraftwerken sowie durch Übersetzungsänderungen bei Transformatoren, die an den Übergängen von einer Spannungsebene zu einer anderen im Netz eingesetzt werden.

Übertragungskomponente

Für die Übertragung der elektrischen Energie sind Transformatoren und Netzleitungen zuständig.

1. Transformatoren
   Mit Hilfe von Transformatoren lassen sich im Netz unterschiedliche Spannungsebenen miteinander verbinden oder zwei Netze voneinander galvanisch trennen. Transformatoren bestehen in der Regel aus zwei elektrischen Kreisen Ober- und Unterspannungswicklung, die über einen Eisenkern magnetisch miteinander gekoppelt sind.

2. Elektrische Leitungen
   Elektrische Leitungen haben die Aufgabe die Verbindung zwischen den elektrischen Bauteilen herzustellen. Sie bestehen aus elektrischen Leitern und isolierenden Umhüllungen. Sie werden benutzt bei

   - elektrischen Leistungsübertragungen zum Beispiel als Drehstromleitung, Starkstrom-Seekabel, etc..
   - Signalübertragungen zum Beispiel als Netzwerkkabel, Luftkabel, etc..
   - Hochfrequenz zum Beispiel in Form von Hybridleitungen und Koaxialleitungen.

Verbrauchskomponente

Beim Energieverbrauch unterscheidet man zwischen dem ohmschen Verbraucher und dem induktiven Verbraucher. Die ohmschen Verbraucher sind unter anderem Glühlampen, Kochherde und Heizungen. Beispiele für induktive Verbraucher sind elektrische Motoren und Transformatoren.

Inselnetzbetrieb

Das Inselnetz versorgt ein definiertes Gebiet und hat keine Verbindung zum Verbundnetz. Zum Beispiel wurde das Westberliner Netz am 4. März nach der Teilung Deutschlands für 40 Jahre zur Strominsel. Ein Inselnetz kann mit einem Dieselgenerator, Photovoltaikanlagen, Windenergieanlagen und Brennstoffzellen betrieben werden. Das Inselnetz hat aber Nachteile zum Beispiel unterliegt es erhöhten Frequenz- und Spannungsschwankungen. Außerdem entstehen hohe Kosten für das Bereithalten von Stromreserven.

Verbundbetrieb

Im Gegensatz zum Inselnetz sind im Verbundnetz die Teilnetze bzw. die Inselnetze miteinander zu einem größerem Netz verbunden. Vorteile sind unter Anderem die Stabilität des Energiesystems, die Lastschwankungsminimierung und die Steigerung der Betriebszuverlässigkeit des Netzes. So sind zum Beispiel in Deutschland die Netze der vier Netzbetreiber EnBW Transportnetze AG, E.ON Netz Gmbh, RWE Transportnetz Strom Gmbh und Vattenfall Europe Transmission Gmbh zusammengeschlossen.

3.2 Netzdarstellung und Implementierung

In diesem Teil wird das elektrische Netz, das bereits oben beschrieben worden ist, modelliert.

3.2.1 Edge

Die Klasse Edge siehe Abb. 3.6 modelliert die Übertragungskomponenten. Sie stellt eine Verbindung zwischen Referenz-, Einspeise- und Lastknoten dar. Nachfolgend werden die einzelnen Parameter des Konstruktors beschrieben:

- id : Index der Kante
- info : Information über die Kante
- source : Index des Quellknotens
- target : Index des Zielknotens
- induktiv: Imaginärteil der Admitanz der Leitung
- resistiv: Realteil der Admitanz der Leitung
- kapazitiv: Kapazität der Leitung
- querleitwert: Querleitwert der Leitung
- länge: Länge der Leitung
- max Strom: Maximaler Strom, der durch die Leitung fließen kann
- current Strom: Aktueller Strom durch die Leitung

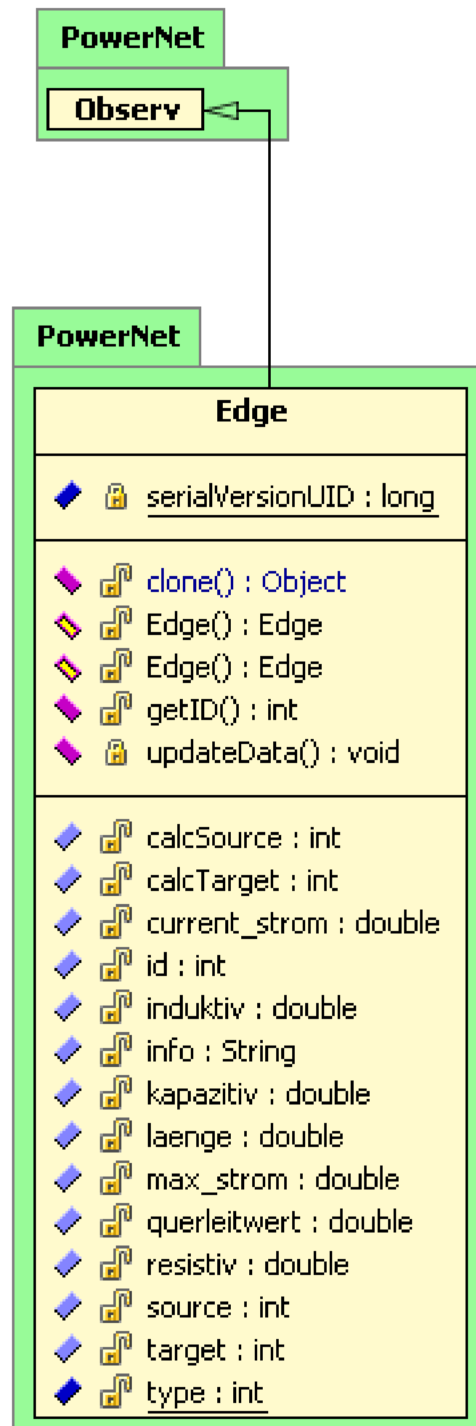


Abbildung 3.6: die Klasse Edge

3.2.2 Node

Die Klasse Node siehe Abb. 3.7 modelliert die Produktions- und Verbrauchskomponenten. Es gibt drei Typen von Knoten:

- Referenzknoten
  Stellt das Gleichgewicht zwischen der eingespeisten und entnommenen Leistung her.

- Einspeiseknoten (PU Knoten)
  Können Synchrongeneratoren sein, wenn sie für die Spannungsregelung eingesetzt werden.

- Lastknoten (PQ Knoten)
  Können Verbraucher, Synchron- oder Asynchrongeneratoren sein, falls sie zur Regelung der Wirk-/Blindleistung dienen.

Nun werden hier die einzelnen Parameter des Node-Konstruktors beschrieben:

- id : Index des Knotens

- state : Ein Node kann als SLACK_NODE, PU_NODE, aber auch als PQ_NODE betrachtet und behandelt werden.

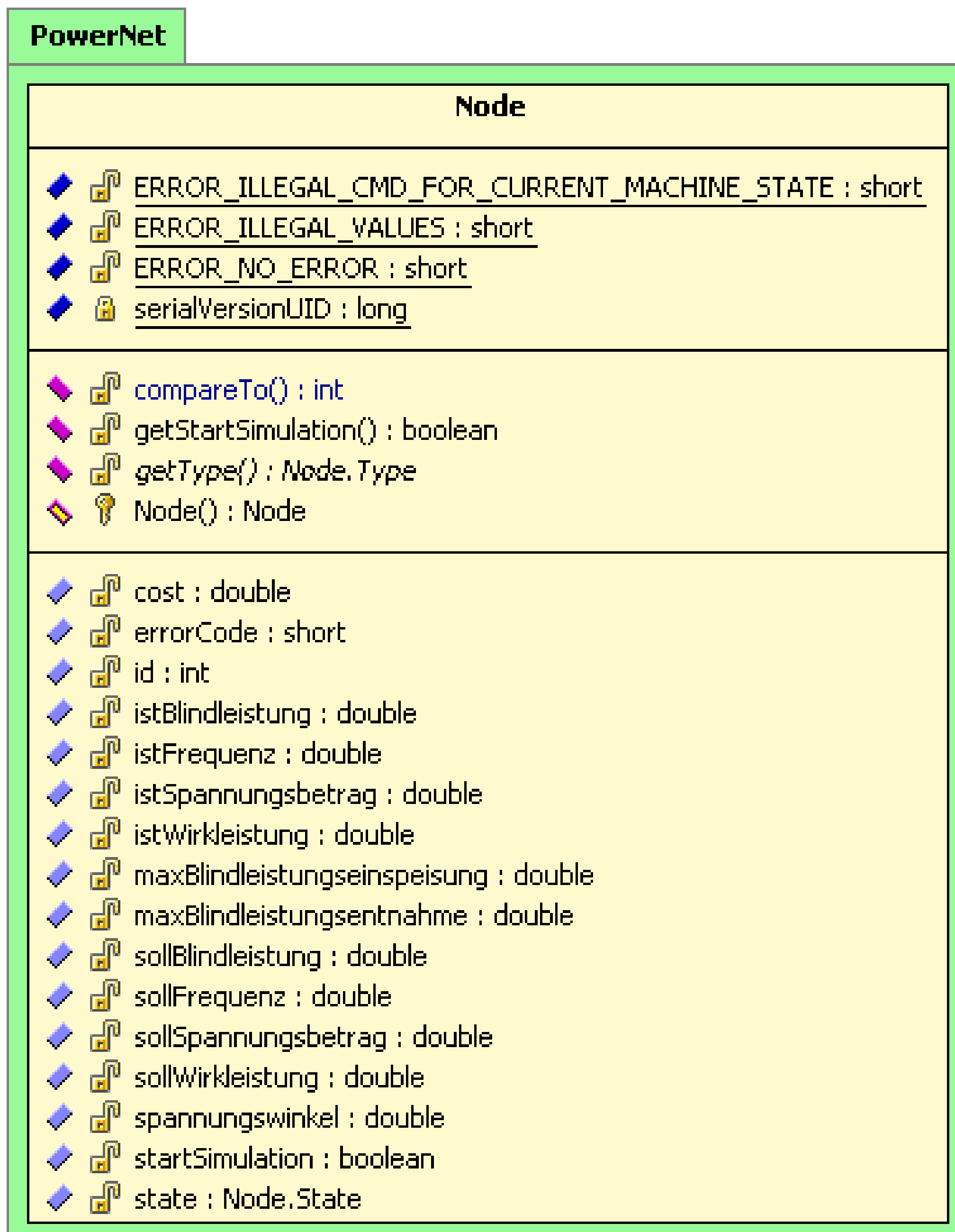


Abbildung 3.7: die Klasse Node

3.2.3 SynchroGenerator

Die Klasse SynchroGenerator siehe Abb. 3.8 ist von der Klasse Node abgeleitet. Sie simuliert einen Synchrongenerator. Jeder Synchrongenerator hat eine Anlaufzeit, eine Nennwirkleistung und ein Trägheitsmoment, die einzeln eingegeben werden können. Für die Wirkleistungsbilanzierung im Inselnetz Betrieb, sind zwei Methoden definiert. Die Methode adjustPower() berechnet aus der alten und der ak-

tuellen Frequenz die Wirkleistung für den Frequenzausgleich. Die Methode adjustFrequenz() ist die Umkehrfunktion zu adjustPower(). Sie berechnet aus der Sollwirkleistung die aktuelle Frequenz im Inselnetz. Die einzelnen Parameter des Konstruktors sind folgendermaßen definiert:

- id : Index des Knotens
- state : Ein Synchrongenerator kann ein PU_NODE oder auch PQ_NODE sein

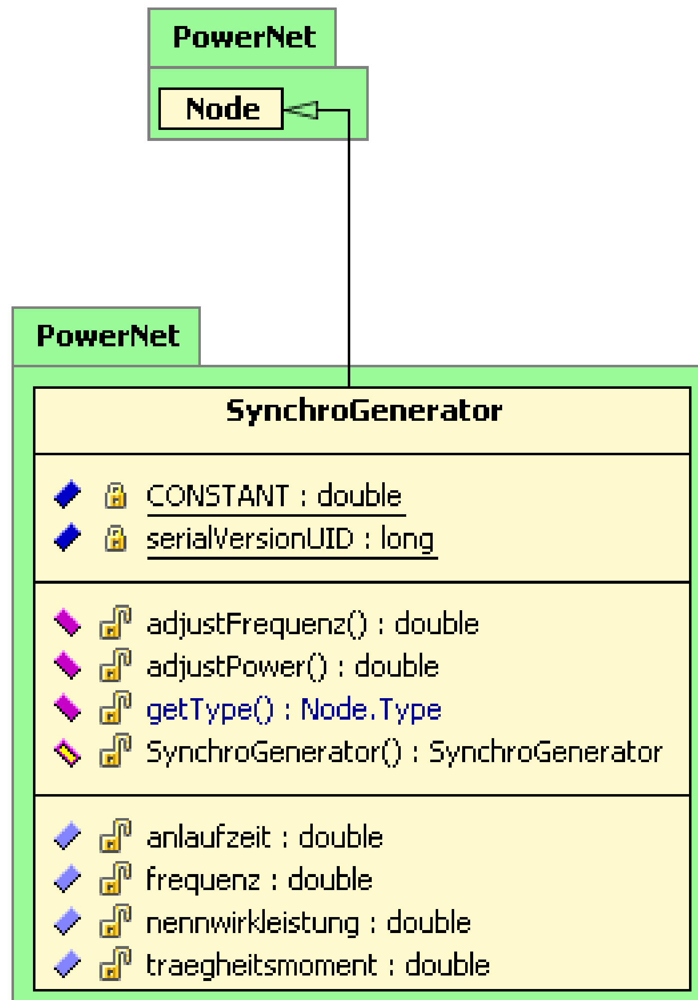


Abbildung 3.8: die Klasse SynchroGenerator

3.2.4 AsynchroGenerator

Die Klasse AsynchroGenerator siehe Abb. 3.9 simuliert einen Asynchrongenerator. Sie erbt von der Klasse Node. Die einzelnen Parameter des Konstruktors sind folgendermaßen definiert:

- id : Index des Knotens
- state : ein Asynchrongenerator ist ein PQ_NODE

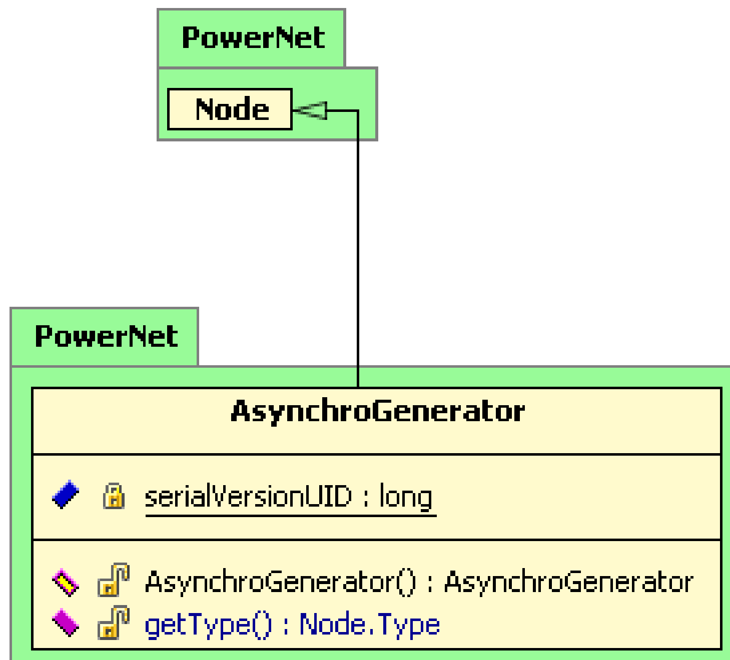


Abbildung 3.9: die Klasse AsynchroGenerator

3.2.5 Inverter

Ein Inverter (Wechselrichter) siehe Abb. 3.10 ermöglicht bei der Kopplung mit einer Windanlage die Erzeugung eines netzsynchronen Wechselstromes und besitzt unter Anderem die gleiche Spannung und Frequenz. Die Klasse Inverter ist von der Klasse Node abgeleitet und hat folgende Parameter:

- id : Index des Knotens
- state : ein Inverter ist ein PQ_NODE

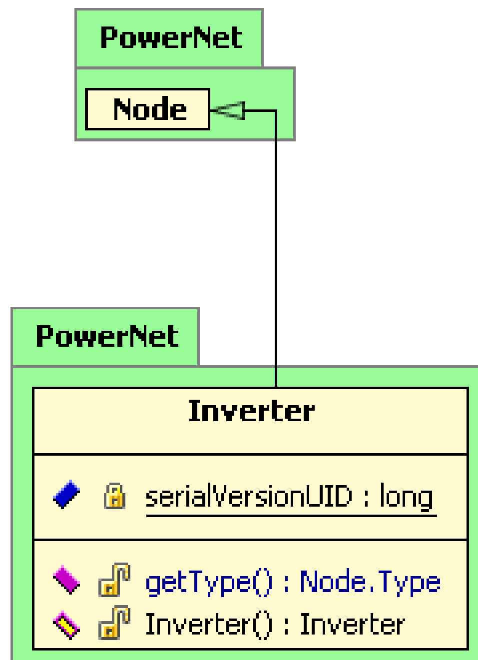


Abbildung 3.10: die Klasse Inverter

3.2.6 Consumer

Die Klasse Consumer(Verbraucher) siehe Abb. 3.11 simuliert die Verbraucherkomponenten. Sie ist von der Klasse Node abgeleitet und hat die folgenden Parameter:

- id : Index des Knotens
- state : ein Consumer ist ein PQ_NODE

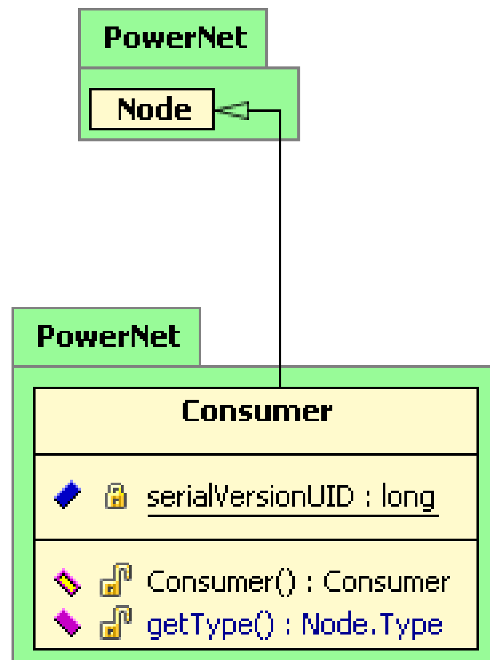


Abbildung 3.11: die Klasse Consumer

3.2.7 PowerNetDatastructure

Die Klasse PowerNetDatastructure siehe Abb. 3.12 enthält die Datenstruktur unseres Netzmodels. Die Nodes und Edges sind jeweils in Mengen des Typs HashSet enthalten.

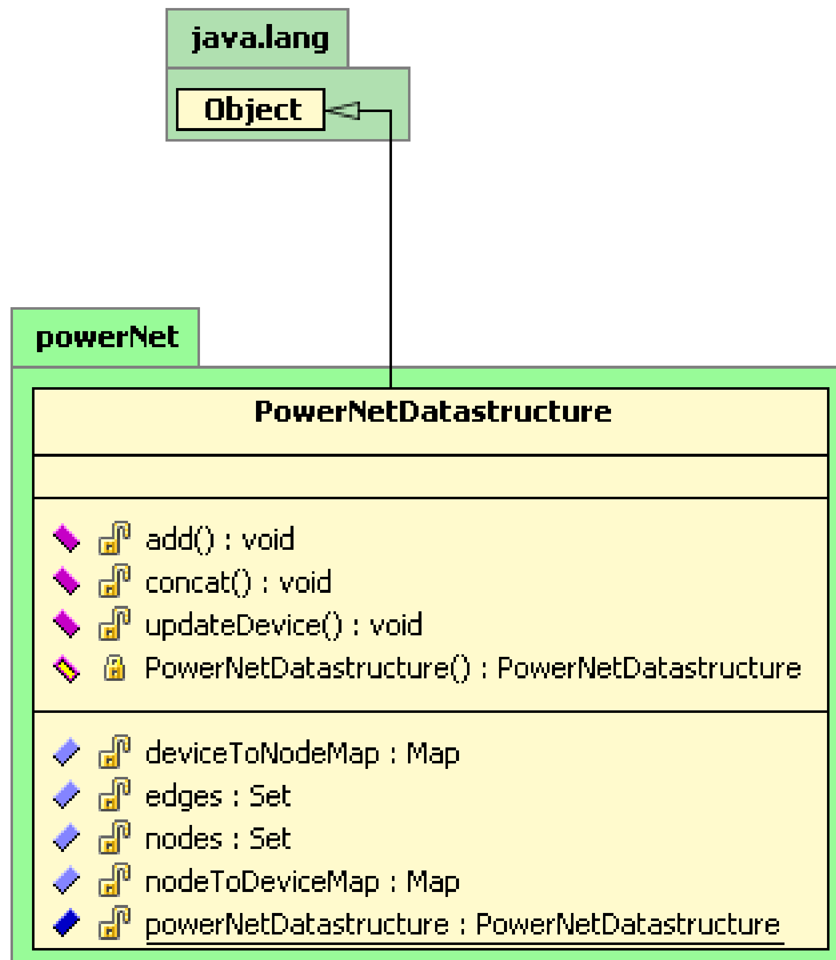


Abbildung 3.12: die Klasse PowerNetDatastructure

3.2.8 AbstractBalancer

Die Klasse AbstractBalancer siehe Abb. 3.13 ist die abstrakte Basis-Klasse für das Balancing, von ihr werden zwei Klassen abgeleitet, die die konkreten Balancing-Verfahren implementieren.

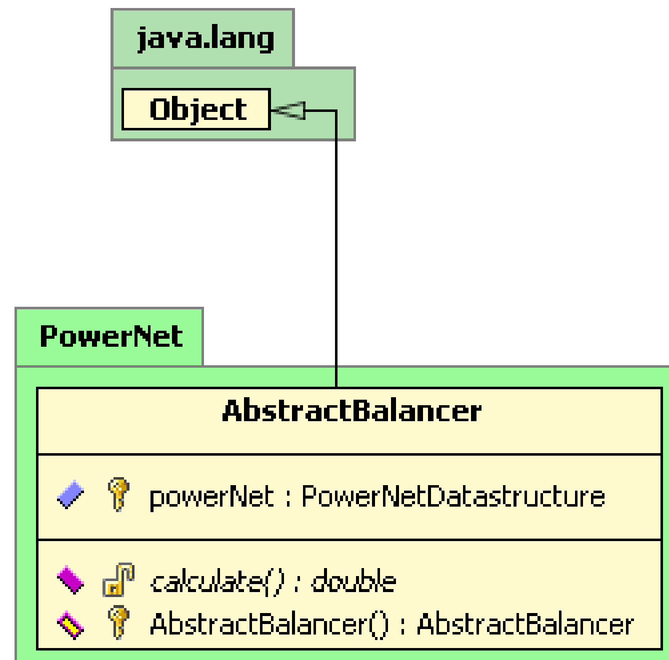


Abbildung 3.13: die Klasse AbstractBalancer

LFBalancer

Die Klasse LFBalancer siehe Abb. 3.14 ist von der Klasse AbstractBalancer abgeleitet. Die Lastflussberechnung, die bereits in den vorherigen Abschnitten erläutert wurde, erfolgt hier durch aufrufen der Methode calculate().

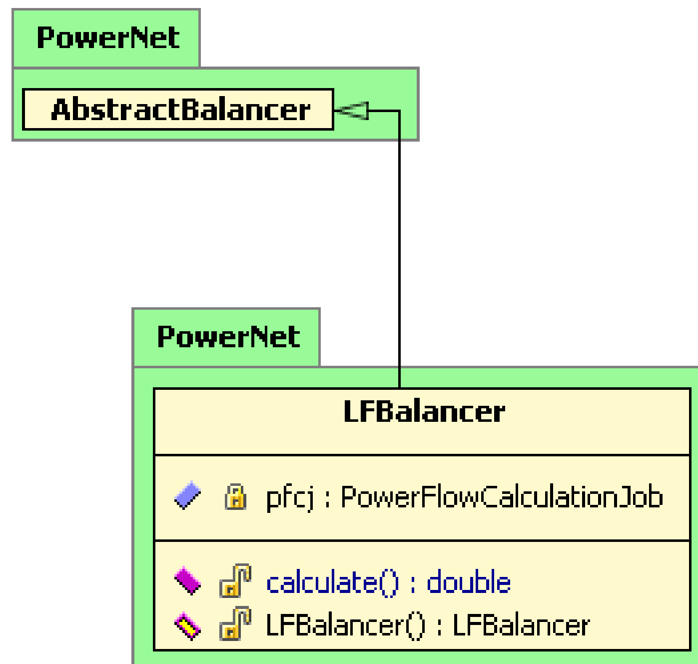


Abbildung 3.14: die Klasse LFBalancer

WLBalancer

Die WLBalancer-Klasse siehe Abb. 3.15 ist ebenfalls von der Klasse AbstractBalancer abgeleitet. Die Methode calculate() führt die Wirkleistungsbilanzierung durch. In den nächsten Abschnitten werden die einzelnen Funktionalitäten näher betrachtet.

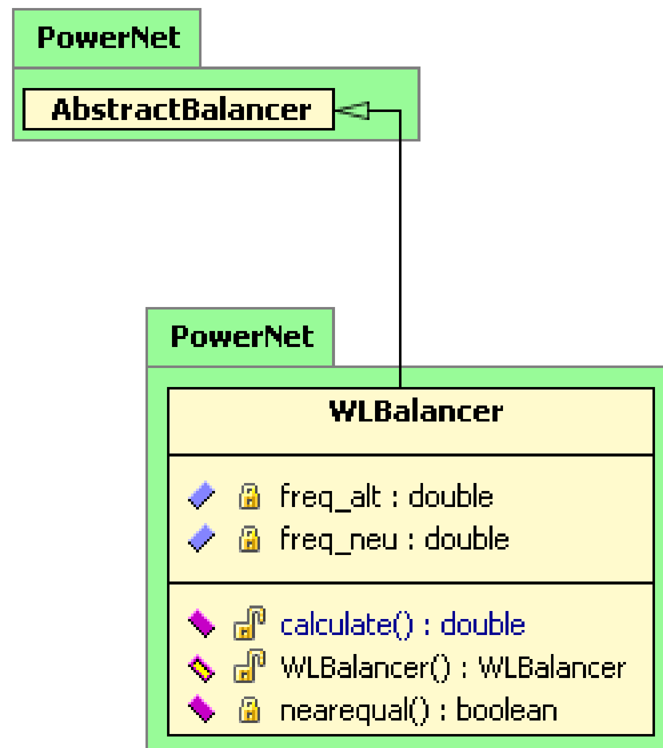


Abbildung 3.15: die Klasse WLBalancer

3.3 Bilanzierungsaufgaben

3.3.1 Bilanzierungen in der Hardwarsimulation

Die Bilanzierung dient im besonderen Maße dem Leistungsausgleich in vermaschten Energieübertragungsnetzwerken. Dabei dient die Wirkleistungsbilanzierung der korrekten einspeisung bzw. entnahme von Wirkleistung und der Erfassung der Netzfrequenz. Die Blindleistungsbilanzierung übernimmt die Regelung der zulässigen Spannung um das Netz stabil zu halten, bzw. die Blindleistungseinspeisungen der einzelnen Netzknoten zu regulieren.

Wirkleistungsbilanzierung

Die Wirkleistungsbilanzierung berechnet die aktuelle Netzfrequenz anhand der Wirkleistungsbilanz. Ebenfalls stellt sie an den Synchrongeneratoren die nötige Leistung ein, um die Wirkleistungsbilanz auszugleichen. Diese beiden Berechnungen passieren allerdings abhängig voneinander. Im einfachen Fall könnte die fehlende oder überschüssige Wirkleistung am Referenzknoten einfach durch das hochfahren bzw. abfahren eines Generators im Netz erledigt werden. Bei mehreren Generatoren wären hier mehrere mögliche Kombinationen von abgegebener oder eingespeister Leistung an verschiedenen Generatoren akzeptabele Lösungen. Allerdings wird im Inselnetzbetrieb die eingespeiste Leistung der

Generatoren durch ein ansteigen, bzw. abfallen der Netzfrequenz beeinflusst. Genau diese Netzfrequenz wird in diesem Bilanzierungsschritt gesucht um einen Wirkleistungsausgleich herzustellen.

Die aktuelle Netzfrequenz bewegt sich im Bereich von 50Hz. Die neu errechneten Abweichungen sind von der Differenz zwischen eingespeister und abgenommener Wirkleistung abhängig. Herscht auf dem Netz ein Wirkleistungsmangel [1] so müssen die Generatoren mehr Wirkleistung ins Netz einspeisen. Das führt dazu, dass die Schwungmasse in den Generatoren abgebremmst wird, somit verringert sich die Frequenz auf dem Netz. Andersherum funktioniert dies analog; liegt ein Wirkleistungsüberschuss [2]vor, beschleunigt die Schwungmasse in den Generatoren und die Frequenz auf dem Netz steigt.

Die Wirkleistungsbilanzierung simuliert nun dieses gegebene Verhalten. Dabei wird, in Abhängigkeit von der Wirkleistungsbilanz, für eine neue Frequenz, überprüft, wieviele Leistung die Synchongeneratoren auf dem Netz einspeisen würden. Deckt die neu eingespeiste bzw. entnommene Wirkleistung die Bilanz ab, so wurde die richtige Frequenz gefunden. Ist die Bilanz immer noch unausgeglichen [3], so wird die Frequenz nach der Methode der binären Suche halbiert, bzw. um den Faktor 0,25 erhöht.

Wichtig ist, dass bei der neu berechneten Wirkleistung nur Synchrongeneratoren berücksichtig werden dürfen. Diese werden über die Frequenz geregelt und so auf die aktuelle Netzsituation angepasst. Die Methode `adjustPower` in der Klasse `SynchroGenerator` gibt die neue Wirkleistungkonfiguration für die, als Parameter angegebene, Frequenzdifferenz an.

$$\Delta P = \frac{(0,5 * j * (2 * \pi * f_{alt})^2) - (0,5 * j * (2 * \pi * f_{neu})^2)}{\Delta t} \tag{3.5}$$

ΔP ist die abgegebene Wirkleistung für die Frequenzdifferenz f_{alt} und f_{neu}. Dabei ist das Trägheitsmoment j noch ein wichtiger Aspekt für den Anlauf des Generators. Die Leistungsdifferenz wird auf dem Zeitintervall Δt betrachtet.

Ein Beispiel verdeutlicht den Ablauf der Wirkleistungsbilanzierung. Im Beispielnetz befinden sich zwei Synchrongeneratoren und ein Konsument. Der Konsument verbraucht 0,3 Watt. Mehrere Konsumenten können auch zu einem einzigen Konsument zusammengefasst werden. Die zwei Synchrongeneratoren speisen 0,3 und 0,1 Watt in das Netz ein. In der Summe 0,4 Watt, also müssen nur noch die, durch die Lastflussberechnung ermittelten Leitungsverluste korrigiert werden. Die Leitungsverluste liegen hier bei -0,098 Watt.

Das Beispiel soll nun klar machen, wo die binäre Suche ansetzt und wie die Frequenz gefunden wird, die einen stabilen Netzbetrieb gewährleistet.

Abbildung 3.16 zeigt die Zusammenhänge zwischen der gesuchten Frequenz und der Wirkleistungsbilanz. Die drei angegebenen Beispiele folgen der binären Suche, dabei wird jeweils mit $50Hz \pm 1$ begonnen. Um den Nullpunkt der Y-Achse sammeln sich die Messwerte. Hier nähert sich der Algorithmus dem gewünschten Ausgleichswert. Je weiter der Messwert vom Nullpunkt entfernt ist, desto größer ist die Streuung.

[1]Ein Wirkleistungsmangel liegt dann vor, wenn mehr Wirkleisung auf dem Netz verbaucht wird, als von den Generatoren eingespeist wird.

[2]Es wird mehr Wirkleistung eingespeist als von den Konsumenten verbraucht.

[3]Ausgeglichen heißt in diesem Sinne, bis zu einer gewissen Genauigkeit.

Iteration	F_{alt}	F_{neu}	Summe der Generatoren	Differenz
1	$50,0$	$51,0$	$-1,615$	$1,5179$
2	$51,0$	$50,5$	$-0,8039$	$0,7059$
3	$50,5$	$50,25$	$-0,4010$	$0,3029$
4	$50,25$	$50,125$	$-0,2002$	$0,1022$
5	$50,125$	$50,0625$	$-0,1000$	$0,0020$

Tabelle 3.5: Beispiel Wirkleistungsbilanzierung

Den Verlauf von Tabelle 3.5 sieht man in Abbildung 3.16 anhand der mittleren Geraden. Dabei startet der Algorithmus bei 51 Hz [4] und erhält als Leistungsdifferenz für die neue Frequenz 1,5179 Watt auf dem betrachteten Netz. Bei dem nächsten Versuch muss sich die Frequenz wieder senken, da die zuvor angegebenen 1,5179 Watt nicht den angestrebten 0 Watt entsprechen. Bei 50,5 Hz erzielt der Algorithmus 0,7059 Watt, daraufhin wird die Hertzzahl wieder verringert auf 50,25 Hz. In Abbildung 3.16 zeigen die Kreuze auf den Linien den Verlauf für die einzelnen Frequenzen. Die nicht besprochenen Verläufe zeigen andere Verhalten, anderer Szenarioen. So wird in der rechten Gerade eine Frequenz von circa 50,9 Hz benötigt, damit das Netz einen Ausgleich schafft. Im linken Verlauf liegt der Wert nahe 50 Hz. Die Streuung auf den Gerade zeigt, das die binäre Suche in sehr kleinen Schritten rechnet. Dabei ballen sich diese Punkte alle um den Nullpunkt der Y-Achse.

Bei der Anwendung mit Generatoren treten allerdings noch Eigenschaften zu Tage, die im simulierten Betrieb keine Erwähnung gefunden haben. Durch die schwankenden Messwerte an den Generatoren wird die Wirkleistungsbilanzierung nicht an jedem Generator mit den gleichen Werten gestartet. Da sich die Werte nur um einen relativen Wert erhöhen driften diese schnell auseinander. Diese Relation bezieht sich auf die unterschiedlichen Trägheitsmomente, Nennwirkleistung und Anlaufzeiten der Maschinen. Das Verhalten zeigt sich in den durchgeführten Experimenten. Dort wurde, um die Leistung der einzelnen Netzkomponenten auszugleichen, die Wirkleistungsbilanzierung als verwaltende Instanz eingesetzt.

Blindleistungsbilanzierung

Die Blindleistungsbilanzierung folgt dem gleichem Prinzip wie die Wirkleistungsbilanzierung. Mit der Ausnahme, dass die Iterationsschritte dem Newton-Raphson-Verfahren und nicht der binären Suche nachgeht. Der Algorithmus sucht eine Spannung im Referenzknoten. Durch diese Spannung wird die Blindleistung einzelner Knoten beeinflusst und der Blindleistungsausgleich hergestellt. Dabei nähert sich das Verfahren stetig an die passende Spannung an.

$$U_{01}^{'} = \frac{U_{01}}{Q_{alt} - Q_{akt}} * Q_{akt} \tag{3.6}$$

U_{01} stellt die Spannungsdifferenz, zu Beginn -0.05, und errechnet anhand der alten Blindleistung Q_{alt} und der neuen Blindleistung Q_{akt} eine neue Differenz der Spannung am Referenzknoten. Die neu errechnete Spannungsdifferenz wird auf die aktuelle Spannung addiert und der Algorithmus startet von neuem, bis die Blindleistungsbilanz, errechnet durch die Lastflussberechnung, einen annehmbaren

[4] Das Verfahren ändert nicht wirklich die Frequenz auf dem physikalischem Netz, die angegebene Hertzzahl wird lediglich für die Berechnung benutzt.

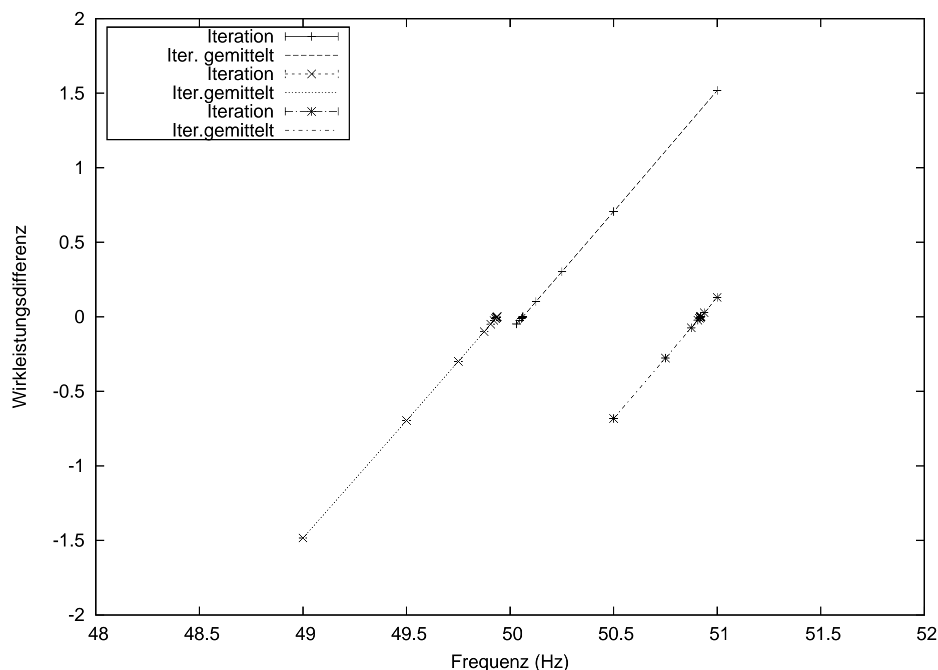


Abbildung 3.16: Beispiel Wirkleistungsbilanzierung für drei verschiedene Systemzustände

Wert [5] erreicht.
Ein Beispiel verdeutlicht dieses Vorgehen.

I	U	U_{01}	$U - U_{01}$	Q_{alt}	Q_{akt}	U'_{01}
1	1	$-0,05$	$0,95$	$0,141413$	$-0,384$	$0,036$
2	$0,95$	$0,036$	$0,986$	$-0,384$	$0,018$	$-0,001$
3	$0,986$	$-0,001$	$0,984$	$0,018$	$-5,018 * 10-4$	$4,434 * 10^{-4}$
4	$0,984$	$4,434 * 10^{-4}$	$0,984$	$-5,018 * 10^{-4}$	$-6,190 * 10-7$	$5,476 * 10^{-8}$
5	$0,984$	$5,476 * 10^{-8}$	$0,984$	$-6,190 * 10-7$	$2,049 * 10^{-11}$	$-1,813 * 10^{-12}$

Tabelle 3.6: Beispiel Blindleistungsbilanzierung

Das Beispielnetz hat neun Knoten, inklusive Referenzknoten. Fünf Knoten treten als Produzenten auf, allesamt Synchrongeneratorknoten, drei Knoten bilden die Gruppe der Konsumenten. Der Spannungsbetrag am Referenzknoten startet mit $U = 1$ und durch die Lastflussberechnung ist eine Blindleistungsbilanz von $Q_G = 0,141413$ gegeben, wie in der ersten Zeile der Tabelle zu sehen ist. Als Startwert für die Spannungsdifferenz der Blindleistungsbilanzierung wird $U_{01} = -0,05$ angenommen. Der, wieder durch die Lastflussberechnung, ermittelte Blindleistungswert dieser neuen Spannung ist $Q_{akt} = -0,384074$. Die Blindleistunsbilanzierung ermittelt durch die oben angebene Formel

[5] Annehmbarer Wert liegt nahe bei 0

$$Q'_{01} = \frac{-0,05}{0,141413 - (-0,384074)} * -0,384074$$

die neue Spannungsdifferenz $U'_{01} = 0,036544$. Die nächste Iteration berechnet Q_{akt} neu mittels der Lastflussberechnung und eine neue Spannungsdifferenz mittels der Formel auf Seite 127. Bereits nach der fünften Iteration wird ein akzeptabeler Wert für die Spannungsdifferenz errechnet, der als gültige Spannungsdifferenz auf die aktuelle Spannung addiert wird.

In Tabelle 3.3.1 ist der Ablauf einer Spannungssuche ausführlich aufgelistet. Nach fünf Iterationen ist bereits ein brauchbares Ergebnis einer gültigen Spannung im Referenzknoten errechnet worden. Das Beispiel findet sich auch in Abbildung 3.3.1 als mittlerer Verlauf wieder. Zu beachten ist hier wieder, dass nach jeder Iteration eine neue Lastflussberechnung durchgeführt wird, die sich in einer neuen Blindleistung Q_{akt} ausdrückt. Abbildung 3.3.1 zeigt die Suche nach der richtigen Spannung im Refe-

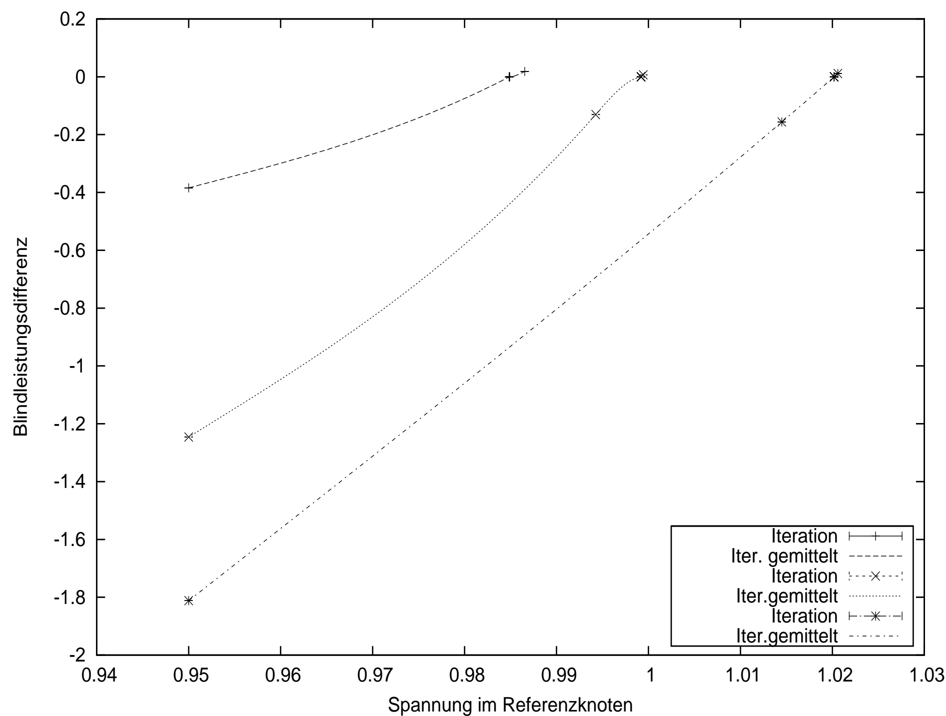


Abbildung 3.17: Beispiel Blindleistungsbilanzierung für drei verschiedene Systemzustände

renzknoten, um die Blindleistungsbilanz auf dem Netz auszugleichen. Hier ist, an den drei Beispielen, gut zu sehen, wie schnell sich die Iterationswerte[6] dem Nullpunkt auf der Y-Achse annähern.

[6]Dargestellt durch die markierten Punkte auf den Linien

Bilanzieren mittels virtuellem Generator

Für die Ermittlung der Wirkleistung, die zur Aufrechterhaltung des elektrischen Energieübertragungssystems nötig sind, wird ein virtueller Generator eingesetzt. Der virtuelle Generator mittelt die Eigenschaften aller Synchrongeneratoren, die in einem Netz auftreten. Somit kann, wie schon bei der Wirkleistungsbilanzierung beschrieben, die aktuelle Frequenz und anschließend, die zur Netzerhaltung nötige Leistung ermittelt werden.

Im ersten Schritt wird der virtuelle Generator erstellt. Dabei werden die Daten (Nennwirkleistung, Trägheitsmoment und Anlaufzeit) aller im Netz befindlichen Synchrongeneratoren erfasst und über die Anzahl der Generatoren gemittelt. Daraufhin wird die momentane Netzfrequenz anhand des virtuellen Generators ermittelt.

$$f_{neu} = \sqrt{f_{alt}^2 - \frac{\Delta p * \Delta t}{2 * j * \pi^2}} \tag{3.7}$$

Hierbei steht Δp für die Leistungsdifferenz und Δt für die schon vorher erwähnte Länge des Simulationsintervalls. Ist einmal die momentane Netzfrequenz ermittelt, muß für $f_{alt} - f_{neu}$ die entsprechende Leistung gefunden werden. Da das Verfahren schon bei der Wirkleistungsbilanzierung erörtert wurde, wird dieser Vorgang hier nicht nocheinmal wiederholt. Allerdings ist noch anzumerken, dass die Leistungsdifferenz für die Frequenzabweichung ebenfalls über den virtuellen Generator errechnet wird.

Zusammenfassend ist über dieses Verfahren zu sagen, das auf einfachem Wege eine Ausgleichsleistung für ein Netz ermittelt werden kann. Durch die Summierung der anfallenden Leistungen an den einzelnen Netzkomponenten, also der Wirkleistungsdifferenz werden allerdings keine zusätzlichen Informationen wie Leitungsverluste berücksichtigt. Falls auf diese Wert Rücksicht genommen werden soll, so muss, anstelle einer einfachen Bilanzrechnung die Lastflußberechnung auf das elektrische Netz angewendet werden. Die momentane Wirkleistungsbilanz kann dann aus dem Referenzknoten ermittelt werden. Jedoch steigert dieser Umweg den Aufwand bei der Berechnung. Desweiteren muss im Laufe der Simulation die Frequenz aufgezeichnet werden, so dass von Iteration zu Iteration auf die letzte Frequenz f_{alt} zurückgegriffen werden kann.

3.3.2 Implementierung der Lastflussestimation in Java

Berechnung der Pseudo-Inversen aus der Singulärwert-Zerlegung

Die Singulärwert-Zerlegung (Singular Value Decomposition) zerlegt eine Matrize M in drei Teile U, V und W, mit $M = U * W * V^T$. Diese Zerlegung lässt sich nutzen, um auch aus Singulären Matrizen M eine (pseudo) Inverse zu bilden. Ist die Zerlegung in U, V und W erst einmal berechnet, so lässt sich die Peudo-Inverse $M^{-1} = V * W^{-1} * U^T$ leicht berechnen, da W nur Werte ungleich Null auf der Hauptdiagonalen besitzt.

Durch transponieren von W (W ist nicht Quadratisch) und durch invertieren der Elemente auf der Hauptdiagonalen lässt sich W^{-1} berechnen. Bei der Invertierung von W gibt es lediglich zu beachten, dass wenn die Diagonale Werte nahe Null ($wert < \epsilon$) enthält, diese Werte zu Nullen invertiert werden

(aus nummerischen Gründen).

Darstellung der Funktionale und berechnung der partiellen Ableitungen

Ein Augenmerk bei der Implementierung war die effiziente und dennoch verständliche und flexible Repräsentation der Funktionale, deren partielle Ableitungen, sowie des gesamten Modelles. Die einzelnen Funktionale sind zusammen mit ihrer jeweiligen Ableitung jeweils in einer Klasse zusammengefasst, welche eine Subklasse der Klasse `Function` ist.

Die Klasse `Function` deklariert die Methoden `double eval()` und `double evalAbl(int varindex)`, die von den Subklassen überschrieben werden. Die Methode `eval` hat keine Parameter und die Methode `evalAbl` bestitzt nur den Integer Parameter `varindex`, der angibt, nach welcher Variable $x_{varindex}$ abgeleitet werden soll. Die weiteren Funktionsparameter $\vec{x}$ und Funktionskonstanten b_{ij} g_{ij} werden global als statische Arrays gehalten. Ihre Referenzen müssen nur einmal vor dem Newton-Raphson-Verfahren gesetzt werden.

Die Klasse `Function` bietet außerdem Hilfsmethoden an, die das Schreiben und Auswerten der Funktionale und Ableitungen vereinfacht: `boolean is_e(int varindex)` beantwortet die Frage danach, ob die Variable $x_{varindex}$ eine e-Variable ist, eine analoge Methode existiert für f. Sie werden benötigt, um zu entscheiden, wie die konkrete Ableitung nach Variable $x_{varindex}$ auszusehen hat. Außerdem gibt es noch die Methoden `double e(int index)` und `double f(int index)`, die den entsprechenden Wert x_i für e_{index} (bzw. für f_{index}) liefert.

Üblicherweise werden bei der Instanziierung der Subklassen notwendige Indizes übergeben, hierbei handelt es sich um den oder die Netzknoten, auf den oder die sich dieses instanziierte Funktional bezieht. Oder anders gesagt, welcher Messwert an welchem Knoten (oder auch Kante) durch diese Funktion berechnet werden soll. Zum Beispiel angenommen `Function func = new P(12)`, dann würde `func.eval()` nun die Nettoknotenwirkleistung am Knoten 12 bestimmen (bei gegebenen $\vec{x}$, b und g).

Listing 3.1: Klasse P

```
1  public class P extends Function{
2          int i;
3
4          public P(int i){
5                  this.i=i;
6          }
7
8          public double eval(){
9                  double sum1=0, sum2=0;
10                 int n = x.length / 2;
11                 for(int j=0;j<n;j++){
12                         sum1 += g[i][j] * e(j) - b[i][j] * f(j);
13                         sum2 += b[i][j] * e(j) + g[i][j] * f(j);
14                 }
15                 return e(i)*sum1 + f(i)*sum2;
16         }
17
18         public double evalAbl(int varindex){
19
20                 int j = (varindex>>1);
21
22                 if (is_e(varindex)) {
23                         if(j != i) {
```

```
24                      return e(i) * g[i][j] + f(i) * b[i][j];
25                  } else { //if j == i
26                      int n = x.length / 2;
27                      double sum = e(i) * g[i][i] + b[i][i]*f(i);
28                      for (int k = 0; k < n; k++) {
29                          sum += g[i][k]*e(k) - b[i][k]*f(k);
30                      }
31                      return sum;
32                  }
33              } else { // if is_f(varindex)
34                  if(j != i) {
35                      return f(i) * g[i][j] - e(i) * b[i][j];
36                  } else { //if j == i
37                      int n = x.length / 2;
38                      double sum = f(i) * g[i][i] - b[i][i]*e(i);
39                      for (int k = 0; k < n; k++) {
40                          sum += b[i][k]*e(k) + g[i][k]*f(k);
41                      }
42                      return sum;
43                  }
44              }
45          }
46
47  }
```

Das Modell, welches nun an das implementierte Newton-Raphson-Verfahren übergeben wird, besteht aus dem Messwert-`double[]`-Array und einer Funktionenschar in Form eines korrespondierenden `Function[]`-Arrays.

Wir übergeben zum Beispiel ein Modell der Form:

Listing 3.2: Beispielmodell

```
1   double z[] = { u1,p1,q1, u2,p2,q2 };
2   Function func[] = { new U2(1),new P(1),new Q(1), new U2(2),new P(2),new Q(2) };
```

Ein Modell mit knotenbezogenen Messwerten U_i^2, P_i sowie Q_i, hat sich als ausreichend redundant für die Bestimmung des Systemzustand gezeigt. Mit diesem Modell konnten Systemzustände bestimmt werden, deren über die Funktionale bestimmten, geschätzten Messwerte nahe den original Messwerten lagen. Es wurden Fehlerquadratsummen von bis zu 10^{-14} erreicht.

Das Einbeziehen einer leitungsbezogenen Messgröße (I_{ij}) für jede Kante führte nicht zu besseren Ergebnissen, die Fehlerquadratsummen wurden um potenzen schlechter (10^-8), waren aber immer noch im akzeptabelem Bereich. Vermutlich bedingt durch numerische Ungenauigkeiten in der Estimation oder bereits in den eingegangenen Messgrößen (die hier durch eine bereits vorliegende Lastflussberechnungs-Implementierung simuliert wurden).

Simuliert und berechnet wurden verschieden vermaschte Netze mit bis zu 50 Knoten und etwa doppelt so vielen Kanten. Die bestimmten komplexen Spannungen, deren Winkel und Beträge lagen innerhalb akzeptabler Tolleranzen.

3.4 UPnP

UPnP [8] wurde entwickelt, um das einfache Hinzufügen verschiedenster neuer Geräte in ein bestehendes Netzwerk zu ermöglichen. Dabei soll gewährleistet werden, dass weder Treiber installiert noch neue Geräte an den verschiedenen gebrauchten Stellen im Netzwerk sichtbar gemacht werden

müssen. UPnP soll also für unterschiedlichste Device von verschiedensten Anbietern die automatische Erkennung durch Zero-Configuration und invisible Networking möglich machen.

UPnP ermöglicht das Finden und Kontrollieren neuer Devices, wie z.B. Netzwerkdrucker, Internet Gateways, Video- und Audiogeräte sowie die gesamte Palette des Consumer Electronics Equipment.

In unserem Real-Dezent System wird das Ziel gesetzt, unterschiedliche Geräte (verschiedene Stromgeneratoren- und Konsumenten-Knoten) zu erkennen und anzusteuern. Genau wegen diesen speziellen Fähigkeiten, wird UPnP im Real-Dezent System angewendet, um zu ermöglichen, dass die neu zum Stromnetz hinzugefügten Geräte automatisch zwecks Ansteuerung erkannt werden.

3.4.1 Komponenten

Ein UPnP Netzwerk enthält zwei wichtige Komponenten, nämlich Devices und Control Points. Jedes Netzwerk setzt sich aus beliebig vielen solcher Komponenten zusammen.

Device

Jedes Ger"at, welches im Netzwerk angesprochen werden soll, muss als ein Device implementiert werden. Ein UPnP-Device oder so genanntes “Root Device“ ist ein Container für Services und eingebettete Devices. Ein Root Device ist einer der Grundsteine im UPnP-Netzwerk. Jedes Root Device kann keine oder mehrere eingebettete Devices bzw. einen oder mehrere Services beinhalten. Jedes *Root Device* besitzt eine Description in Form eines XML-Dokuments, in dem einige wesentliche Angaben über das Device zusammen gefasst werden, z.B. *device Type, friendlyName, manufacturer, manufacturer URL, modelName, modelDescription, UDN*, eine Liste von eingebetteten Devices und eine Liste von Services, u.s.w.

Services

Ein Service ist die kleinste logische Funktions- bzw. Steuerungseinheit im UPnP-Netzwerk. Es ähnelt Methoden in Java, besteht aus Actions und modelliert anhand State Variablen den Zustand von Root Devices.

State Variablen sind einzelne Aspekte vom Zustand eines Root Device und existieren nur in den Services, daher kann jedes Service keine oder mehrere Actions beinhalten, aber es muss mindestens eine State Variable besitzen.

Ähnlich wie die Description eines Devices, spezifiziert das UPnP-Forum auch das XML-Template für Service Descriptions, in der Actions und deren Argumente bzw. einige wesentliche Angaben von State Variablen, z.B. deren Datentyp, Bereich und Werte etc., definiert werden.

Im Real-Dezent System sollen Aktionen wie get/set für Informationen von Devices und An-/Ausschalten der Devices, sowie Steigern/Senken der Leistungseinspeisung von Konten (Devices) implementiert werden. So sind die Zustandsvariablen (wie beispielweise die Parameter) von Devices auch über Services zugreifbar. Alle zur Verfügung stehenden Aktionen und Zustandsvariablen eines Device werden in einer Description erfasst, die in einem XML-Format beschrieben und abgespeichert wird.

Control-Point

Control-Points können im UPnP Netzwerk alle Services, die von verschiedenen Devices angeboten werden, entdecken und steuern. Sobald ein Control-Point einen neuen Device gefunden hat, kann er sich die Description des Device holen und die Actions des Device aufrufen, um Services zu steuern. Auch ist der Control-Point in der Lage, die State Variablen von Services zu überwachen und bei Wertänderungen von Variablen die *Event Message* zu empfangen.

Action

Jede Action kann keine oder mehrere Argumente besitzen und jedes Argument kann in der Service Description als Eingabe- oder Ausgabeparameter definiert werden. Jedes Argument sollte einer Variablen entsprechen. Durch den Aufruf einer Action kann das Device seinen Status mittels Statevariablen verändern.

3.4.2 Protokolle

UPnP verwendet hauptsächlich Open Standard Protokolle wie TCP/IP, HTTP und XML. Dabei bildet der TCP/IP Protocol Stack die Basis eines UPnP Netzwerkes. Zu beachten sind die folgenden Protokolle, die bei der Kommunikation zwischen Control-Point und Device, sowie beim Steuern der Devices eine grosse Rolle spielen:

- HTTP,HTTPU,HTTPMU HTTP bzw. HTTPU und HTTPMU, welche Variationen von HTTP sind, werden zur Kommunikation genutzt da diese Protokolle, im Gegensatz zu HTTP, das Versenden von Nachrichten ermöglichen.
- SSDP (Simple Service Discovery Protokol) definiert, wie Devices im UPnP Netzwerk erkannt werden können. Dafür liefert SSDP Methoden, die es einem Control-Point ermöglichen, Ressourcen bzw. Geräte im Netzwerk zu finden. SSDP-Anfragen der Control-Points werden über HTTPMU versendet.
- SOAP (Single Object Access Protocol) definiert den Gebrauch von XML und HTTP, um remote procedure calls (RPC) auszuführen. Das ist dadurch zu erklären, dass SOAP die existierende Internetinfrastruktur verwendet, und daher sehr gut mit Firewalls und Proxys umgehen kann.

Eventing

Eventing basiert auf GENA(*General Event Notification Architecture*). Dadurch können Control-Points die State Variablen abonnieren, damit sie automatisch über Wertänderung von registrierten State Variablen informiert werden.

Nach der Steuerung des Devices erhält der Control-Point schon genügend Angaben über Devices und deren Services. Zuerst sendet der Control-Point zu einem Service ein „*Subscription Request*“, um sich für eine oder mehrere State Variablen des Service zu registrieren. Sobald dieser „*Subscription Request*“ vom Service angenommen wird, schickt der Service („*publisher*“) dem Control-Point sofort

eine „*Subscription*“ mit deren Dauer zurück. So erfolgt eine Registrierung von State Variablen für den Control-Point.

Vor dem Ablauf einer „*Subscription*“ schickt der Control-Point einen „*Renewal Request*“ aus, falls er die State Variable weiterhin abonnieren möchte.

Wenn der Control-Point die State Variable nicht mehr benötigt, wird eine „*Cancellation*“ Mitteilung zum Service gesendet.

Während der Dauer einer *Subscription* einer State Variable informiert der Service per „*Event Message*“ alle registrierten Control-Points über die neuesten Variablenwerte sobald Wertänderungen registrierter State Variablen vorliegen.

Jede *Event Message* umfasst Namen von einer oder mehreren geänderten State Variablen und deren aktuellen Werten. Allerdings wird nicht mitgeteilt, warum sich der Wert einer state Variablen geändert hat.

Eine *Event Message* kommt ebenso in XML zum Ausdruck und ist nach GENA formatiert. Solche *Messages* sollten so schnell wie möglich ausgeschickt werden, so dass alle registrierten Control-Points rechtzeitig über den aktuellen Zustand eines Services informiert sind.

Wenn sich mehr als ein Variablenwert gleichzeitig ändert, sollten alle Wertänderungen in einer einzelnen Mitteilung zusammengefasst werden, um Datenverarbeitung und Netzverkehr zu verringern.

Eine besondere, initiale *Event Message* wird vom Service zum Control-Point gesendet, wenn er sich zum ersten Mal registriert. Sie beinhaltet eine Liste der Namen und Werte aller State Variablen des Service. Dadurch kann der Control-Point den Zustand des Service feststellen.

3.5 UPnP Ablauf

Im folgenden wird der Ablauf der Device Anmeldung an einem Controlpoint näher erläutert. Die einzelnen Schritte einzeln präsentiert.

3.6 Eventing in UPnP-Umgebung

Im letzten Kapitel ist die UPnP-Technik bereits erklärt. Hier wird nur der Ablauf von UPnP, besondere Eventing in UPnP, vorgetellt.

1. **Adressierung (*Addressing*)**

   Da UPnP auf einem IP-Netzwerk basiert, muss ein Device bzw. Control-Point zuerst eine gültige IP-Adresse besitzen. Dies kann nach dem UPnP-Standard entweder via DHCP (*Dynamic Host Configuration Protocol*) oder via *zeroconf* erfolgen.

   Ein Device kann ebenso über seinen *Friendly Name*[7] verfügbar sein. Dies ist in höheren Protokoll-Schicht zu implementieren. In diesen Fällen wird es notwendig, einen *Friendly Name* eines Device auf eine gültigen IP-Adresse abzubilden. Dazu dient DNS(*Domain Name Services*).

[7] Der ist eine kurze Beschreibung des Device für Endbenutzer.

2. **Lokalisierung (*Discovery*)**

   Ein UPnP-Device muss seine Existenz an die Control-Points melden, sobald das Device eine IP-Adresse besitzt. Eine „ssdp::alive“ *discovery message* wird vom Device unter vernetzten aktivierten Control-Points verbreitet. Dies erfolgt via UDP auf der Basis des SSDP[8]. Beim *CyberLink for JAVA* [9], welches ein OpenSource Entwicklungspaket für UPnP darstellt, dient eine Methode ControlPoint::deviceNotifyReceive(SSDPPacket paket) dazu, *discovery messages* von Devices abzuhören.

   Ebenso kann ein Control-Point nach verfügbaren Devices im UPnP-Netzwerk suchen, sobald das Device im Netzwerk auftaucht. In diesem Fall verbreitet der Control-Point unter allen verfügbaren Devices eine *discovery message*, so dass der Control-Point über die Funktionen bzw. Services von allen Devices informiert ist. Sobald ein Device solch eine *discovery message* von einem Control-Point empfängt, antwortet das Device sofort mit einer *response message*, die ein paar wichtige Angaben des Device enthält. Das Empfangen von *response messages* beim Control-Point erfolgt über die Methode ControlPoint::deviceSearchResponseReceived (SSDPPacket packet).

   In beiden Fällen enthält die *discovery message* nur die wichtigsten Angaben eines Device, z.B. Device Name, Device Type, die URL zur genauen Beschreibung des Device, seine Services bzw. eingebettete Devices u.s.w.

   Das Verlassen bzw. die Änderung der IP-Adresse eines Device wird über die „ssdp::byebye“ *discovery message* allen anderen Devices bzw. Control-Points mitgeteilt.

3. **Beschreibung (*Description*)**

   Wenn ein Control-Point per *discovery message* ein Device gefunden hat, welches er benötigt und noch ausführlichere Informationen über das Device anfordern möchte, holt er sich per HTTP die Description des Device von der URL-Adresse, welche ihm bei der Lokalisierung mitgeteilt wurde.

   Eine UPnP-Description, die in Form einer XML-Datei vorliegt, wird in zwei logische Teile eingeteilt - Device-Description und Service-Description.

   Eine Device-Description umfasst z.B. den Modell-Namen, die Modell-Nummer und Seriennummer etc. Für jeden Service, den ein Device enthält, listet die Device-Beschreibung jeweils den Typ, die Namen, die URL-Adresse für die Kontrolle und die URL-Adresse für Ereignismeldung (*Eventing*) aller Services auf. Eine Device-Description enthält auch alle Descriptions seiner eingebetteten Devices, falls das Root-Device eingebettete Devices enthält.

   Device-Descriptions kann man nach einem standardisierten UPnP Device Template erstellen, das vom UPnP-Forum [7] spezifiziert wurde.

   Ähnlich wie bei Device-Descriptions fasst eine Service-Description alle wesentlichen Angaben des Services zusammen, z.B. eine Liste von Actions, auf die das Service antwortet und Parameter oder Argumente für jede Action. Die Service-Description enthält eine Liste von Variablen, die den eventuellen Zustand des Service repräsentieren können. Auch die Datentypen und der Wertebereich von Variablen werden in Descriptions beschrieben.

[8]Das *Simple Service Discovery Protocol(SSDP)* ist ein Netzwerkprotokoll, welches zur Suche nach UPnP-Devices im Netzwerk dient.

4. **Steuerung (*Control*)**

   Sobald die Device-Descriptions vorhanden sind, weiss der Control-Point das Wesentliche über die Devices. Aber er weiss nicht viel über die Services. Um mehr über die Services zu lernen, muss ein Control-Point eine ausführliche UPnP-Description für jeden Service bekommen.

   Um ein Device zu steuern, sendet ein Control-Point an einen Service des Device ein *action request*, welche eine XML-basierte SOAP[9]-Mitteilung (*control message*) ist.

   Als Antwort auf die *control message* gibt der Service actionspezifische Werte oder Fehlermeldungen zurück.

5. **Ereignismeldung(*Eventing*)**

   Eventing basiert auf GENA(*General Event Notification Architecture*). Dadurch können Control-Points die State Variablen abonnieren, damit sie automatisch über Wertänderung von registrierten State Variablen informiert werden.

   Nach der Steuerung des Devices erhält der Control-Point schon genügende Angaben über Devices und deren Services. Zuerst sendet der Control-Point zu einem Service ein „*Subscription Request*“, um sich für eine oder mehrere State Variablen des Service zu registrieren. Sobald dieser „*Subscription Request*“ vom Service angenommen wird, schickt der Service („*publisher*“) dem Control-Point sofort eine „*Subscription*“ mit deren Lebensdauer zurück. So erfolgt eine Registrierung von State Variablen für den Control-Point.

   Vor dem Ablauf einer „*Subscription*“ schickt der Control-Point einen „*Renewal Request*“ aus, falls er die State Variable weiterhin abonnieren möchte.

   Wenn der Control-Point die State Variable nicht mehr benötigt, wird eine „*Cancellation*“ Mitteilung zum Service gesendet.

   Während der Dauer einer *Subscription* einer State Variable informiert der Service per „*Event Message*“ alle registrierten Control-Points über die neuesten Variablenwerte sobald Wertänderungen registrierter State Variablen vorliegen.

   Jede *Event Message* umfasst Namen von einer oder mehreren geänderten State Variablen und deren aktuellen Werte. Allerdings wird nicht mitgeteilt, warum sich der Wert einer State Variablen geändert hat.

   Solche *Messages* sollten so schnell wie möglich ausgeschickt werden, so dass alle registrierten Control-Points rechtzeitig über den aktuellen Zustand eines Services informiert sind.

   Wenn sich mehr als ein Variablenwert gleichzeitig ändert, sollten alle Wertänderungen in einer einzelne Mitteilung zusammengefasst werden, um Datenverarbeitung und Netzverkehr zu verringern.

   Eine besondere, initiale *Event Message* wird vom Service zum Control-Point gesendet, wenn er sich zum ersten Mal registriert. Sie beinhaltet eine Liste der Namen und Werte aller State Variablen des Service. Dadurch kann der Control-Point den Zustand des Service feststellen.

6. **Präsentation (*Presentation*)**

[9] Das *Simple Object Access Protocol(SOAP)* ist ein Netzwerkprotokoll, mit dessen Hilfe Daten zwischen Systemen ausgetauscht und *Prozedur-Fernaufruf* durchgeführt werden können.

Wenn ein Device eine URL-Adresse für die Präsentation hat, dann kann der Control-Point eine Webseite von dieser URL-Adresse zurückholen und in einen Browser laden. Dadurch wird eine HTML-basiete Benutzeroberfläche zur Steuerung und Übersicht vom Zustand des Device erzeugt.

3.7 UPnP Implementierungen

In diesem Kapitel wird näher auf die UPnP Implementierungen eingegangen. Es werden exemplarisch die Implementierung einzelner Komponenten im Allgemeinen und im Rahmen des REAL-DEZENT Projektes dargestellt. Im ersten Teil ist die Implementierung der Devices mit den nötigen Konfigurations-Dateien im Vordergrund, anschliessend die Implementierung eines Controlpoints, welcher auch innerhalb der Projektgruppe benutzt wurde.

3.7.1 Implementierung Device, Controlpoint

Devices sind Container von Services und Actions, die ausgeführt werden. Sie enthalten Statusvariablen, Actions und Services. Die Implementierung eines Devices in JAVA sieht folgendermassen

Listing 3.3: Cyberlink LightDevice Java Implementierung

```
1  public class LightDevice extends Device implements ActionListener, QueryListener
2  {
3   private final static String DESCRIPTION_FILE_NAME =
4     "src/org/cybergarage/samples/upnpSampleLight/description/description.xml";
5
6   private StateVariable powerVar;
7
8   public LightDevice() throws InvalidDescriptionException
9   {
10   super(new File(DESCRIPTION_FILE_NAME));
11
12   Action getPowerAction = getAction("GetPower");
13   getPowerAction.setActionListener(this);
14
15   Action setPowerAction = getAction("SetPower");
16   setPowerAction.setActionListener(this);
17
18   ServiceList serviceList = getServiceList();
19   Service service = serviceList.getService(0);
20   service.setQueryListener(this);
21
22   powerVar = getStateVariable("Power");
23
24   Argument powerArg = getPowerAction.getArgument("Power");
25   StateVariable powerState = powerArg.getRelatedStateVariable();
26   AllowedValueList allowList = powerState.getAllowedValueList();
27   for (int n=0; n<allowList.size(); n++)
28    System.out.println("[" + n + "]␣=␣" + allowList.getAllowedValue(n));
29
30   AllowedValueRange allowRange = powerState.getAllowedValueRange();
31   System.out.println("maximum␣=␣" + allowRange.getMaximum());
32   System.out.println("minimum␣=␣" + allowRange.getMinimum());
33   System.out.println("step␣=␣" + allowRange.getStep());
34  }
35  ...
36  ...
37  ...
38  ////////////////////////////////////////////////
39  // ActionListener
40  ////////////////////////////////////////////////
41
42  public boolean actionControlReceived(Action action)
43  {
44   String actionName = action.getName();
45
46   boolean ret = false;
47
48   if (actionName.equals("GetPower") == true) {
49    String state = getPowerState();
50    Argument powerArg = action.getArgument("Power");
```

```
51     powerArg.setValue(state);
52     ret = true;
53    }
54    if (actionName.equals("SetPower") == true) {
55     Argument powerArg = action.getArgument("Power");
56     String state = powerArg.getValue();
57     setPowerState(state);
58     state = getPowerState();
59     Argument resultArg = action.getArgument("Result");
60     resultArg.setValue(state);
61     ret = true;
62    }
63
64    comp.repaint();
65
66    return ret;
67   }
68   ...
69   ...
70   ...
71
72 }
```

Als erstes muss eine Klasse für das gewünschte Deviceimplementiert werden. Dazu erbt die zu implementierende Klasse von der Klasse DEVICE der Cyberlink UPnP-API [9]. Um auf Events reagieren zu können, wird der ActionListener, welche auch aus der Cybergarage stammt, implementiert. Der Konstruktor dieser Klasse ruft den Konstruktor der Oberklasse auf und übergibt diesem ein FILE Object mit dem Pfad, an der sich die Konfigurationsdatei, also die description.xml befindet. Anschliessend werden nun Actions, Statusvariablen definiert und die ServiceListe gemäss der Konfigurationsdatei erstellt. In den ActionListener Methoden muss konfiguriert werden, wie sich das Device verhalten soll, falls bestimmte Actions ausgelöst werden. Das obige Beispiel zeigt eine Lampe mit 2 möglichen Actions GetPower und SetPower mit einem Service Power. Der ActionListener prüft, welche Action ausgeführt wird und führt dann die entsprechende Methode aus. Beim Starten der Devices muss brücksichtigt werden, dass die benötigten XML Dateien zur Gerätinstallation und zur Bereitstellung von Services benötigt werden. Der folgende Auszug einer Device XML Datei dient zur Geräteinstallation mit den dazugehörigen Services.

Listing 3.4: CyberLink for Java description.xml für das Device

```
1   ...
2   ...
3   ...
4   <deviceType>urn:schemas-upnp-org:device:light:1</deviceType>
5   <friendlyName>CyberGarage Light Device</friendlyName>
6   <manufacturer>CyberGarage</manufacturer>
7   <manufacturerURL>http://www.cybergarage.org</manufacturerURL>
8   <modelDescription>CyberUPnP Light Device</modelDescription>
9   <modelName>Light</modelName>
10  <modelNumber>1.0</modelNumber>
11  <modelURL>http://www.cybergarage.org</modelURL>
12  <serialNumber>1234567890</serialNumber>
13  <UDN>uuid:cybergarageLightDevice</UDN>
14  <UPC>123456789012</UPC>
15  <iconList>
16   <!--icon>
17     <mimetype>image/gif</mimetype>
18     <width>48</width>
19     <height>32</height>
20     <depth>8</depth>
21     <url>icon.gif</url>
22   </icon-->
23  </iconList>
24  <serviceList>
25   <service>
26     <serviceType>urn:schemas-upnp-org:service:power:1</serviceType>
27     <serviceId>urn:schemas-upnp-org:serviceId:power:1</serviceId>
28     <SCPDURL>/service/power/description.xml</SCPDURL>
29     <controlURL>/service/power/control</controlURL>
30     <eventSubURL>/service/power/eventSub</eventSubURL>
31   </service>
32  </serviceList>
33  <presentationURL>http://www.cybergarage.org</presentationURL>
34 </device>
35  ...
36  ...
37  ...
```

Dabei dienen die ersten Zeilen zu Informationszwecken, die aufgrund der Tags selbsterklärend sein sollten und laut Spezifikation auch benötigt werden . Wichtig in diesem Auszug sind die Tags modelDesription, serialNumber, UDN, UPC und ServiceList. Dabei gibt modelDescription den Namen eines Devices wieder, dieser Tag kann gesetzt und erhalten werden mit den Methoden setFriendlyName() und getFriendlyName() in der Klasse Device. Der Tag serialNumber muss eindeutig sein und die UDN wird an ControlPoints übergeben, über die die Geräte an ControlPoints identifiziert werden. Die serviceList gibt die Services an, die ein Device hat. Die Services wiederum enthalten die Action, welche ausgeführt werden. Ferner ist zu beachten, dass die Device Klasse ihre Services aufgrund der description.xml des Devices erkennt. Falls man einen Service anbieten möchte, der Actions enthält, müssen auch die Parameter für die Actions in der Konfigurationsdatei für jeden Service stehen. Der folgende Auszug zeigt eine die description.xml für den Service Power.

Listing 3.5: Cybergarage LightDevice description.xml für Power Service

```
1  <actionList>
2   <action>
3    <name>SetPower</name>
4    <argumentList>
5     <argument>
6      <name>Power</name>
7      <relatedStateVariable>Power</relatedStateVariable>
8      <direction>in</direction>
9     </argument>
10    <argument>
11     <name>Result</name>
12     <relatedStateVariable>Result</relatedStateVariable>
13     <direction>out</direction>
14    </argument>
15   </argumentList>
16  </action>
17  <action>
18   <name>GetPower</name>
19   <argumentList>
20    <argument>
21     <name>Power</name>
22     <relatedStateVariable>Power</relatedStateVariable>
23     <direction>out</direction>
24    </argument>
25   </argumentList>
26  </action>
27 </actionList>
28 <serviceStateTable>
29  <stateVariable sendEvents="yes">
30   <name>Power</name>
31   <dataType>boolean</dataType>
32   <allowedValueRange>
33    <maximum>1</maximum>
34    <minimum>0</minimum>
35    <step>1</step>
36   </allowedValueRange>
37  </stateVariable>
38  <stateVariable sendEvents="no">
39   <name>Result</name>
40   <dataType>boolean</dataType>
41  </stateVariable>
42 </serviceStateTable>
```

Der obige Auszug, der aus der description.xml für den Service Power stammt, definiert Variablen für eine Action, welche in einer Service enthalten ist. Dabei wird für die Variablen definiert, wie sie eingesetzt werden sollen. Die Tagnamen für die State Variablen müssen mit denen in der Klasse definierten Actions und Statevariablen übereinstimmen. Für die Variablen wird hier auch der Wertebereich und der Typ angegeben, z.B. ob sie boolesche Werte erhalten oder integer, oder ob ein ControlPoint Events von diesen Variablen berücksichtigt oder nicht.
Mit Hilfe dieser drei Dateien ist es dann möglich ein Device zu erstellen. Damit man die Devices auch benutzen kann, benötigt man Controlpoints. Ein Device kann sich an einem Controlpoint anmelden um seine Services mit den dazugehörigen Actions anzubieten. Die unten gezeigt Implementierung eines Controlpoints zeigt einen im Rahmen der PG benutzten Controlpoint. Dieser Controlpoint dient zum Loggen von Wertänderungen in unserer Simulationsumgebung. Damit ein ControlPoint auf Er-

eignisse reagieren kann, müssen Listener Interfaces zur Klassenstruktur hinzugefügt und die entsprechenden Methoden implementiert werden. Bei dem folgenden Beispiel werden die Events ausgegeben, die bestimmte Device Typ ähneln. Es sollte aber beachtet werden, dass hier jedes Event an diesem ControlPoint ankommt, jedoch aufgrund der Aufgabenstellung nicht weiter beachtet wird. Weiterhin werden auch Listener Methode aufgerufen, falls sich Geräte anmelden bzw. abmelden. Zudem wurden aus dem obigen Beispiel übersichtshalber die GUI Elemente entfernt.

Listing 3.6: Cybergarage LightDevice description.xml für Power Service

```
1  package edu.udo.cs.ls3.DEZENT.upnp.ctrlPoints;
2
3  import java.io.IOException;
4  import java.util.Date;
5
6  import org.apache.log4j.FileAppender;
7  import org.apache.log4j.Level;
8  import org.apache.log4j.Logger;
9  import org.apache.log4j.SimpleLayout;
10 import org.cybergarage.upnp.ControlPoint;
11 import org.cybergarage.upnp.Device;
12 import org.cybergarage.upnp.device.DeviceChangeListener;
13 import org.cybergarage.upnp.event.EventListener;
14
15 public class CPLogger extends ControlPoint implements EventListener, DeviceChangeListener{
16     private Logger logger = Logger.getLogger(CPLogger.class);
17     private SimpleLayout layout = new SimpleLayout();
18     private FileAppender appender;
19
20     ControlPoint cp = this;
21
22     String filename;
23
24     public CPLogger(){
25
26
27         this.addDeviceChangeListener(this);
28         this.addEventListener(this);
29         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30         logger.setLevel(Level.INFO);
31         logger.addAppender(appender);
32             @Override
33     /*
34      * ==================== UPnP Listener implentation ==============================
35      *
36      *  */
37 //  Logging the events from the devices INVERTER, CONSUMER, ASYNCHR_GENERATOR and SYNCHR_GENERATOR
38     public void eventNotifyReceived(String uuid, long seq, String varName,
39             String value) {
40         if(logState){
41             if(getSubscriberService(uuid).getDevice().getStateVariable("Type").getValue().equals("INVERTER")||
42                     getSubscriberService(uuid).getDevice().getStateVariable("Type").getValue().equals("CONSUMER")||
43                     getSubscriberService(uuid).getDevice().getStateVariable("Type").getValue().equals("SYNCHR_GENERATOR")
                        ||
44                     getSubscriberService(uuid).getDevice().getStateVariable("Type").getValue().equals(\"{a}
                            SYNCHR_GENERATOR"))
45             {
46                 logger.info(new Date().getTime() + ";" + uuid + ";" +varName + ";" + value);
47             }
48             else {
49                 System.out.println("Values␣of␣Device:␣"+getSubscriberService(uuid).getDevice().getFriendlyName()+"ignored␣
                        ␣!!!");
50             }
51         }
52
53
54     }
55     public void registerShutdownHook()
56     {
57         // Vor dem Beenden der Java VM auf jeden Fall vom UPnP Stack abmelden und Subscriptions kuendigen!!!
58         Runtime.getRuntime().addShutdownHook(new Thread()
59         {
60             public void run()
61             {
62                 cp.stop();
63             }
64         });
65     }
66     @Override
67     public void deviceAdded(Device dev) {
68         String devName = dev.getFriendlyName();
69         System.out.println( \"{a}dding␣Device:␣" + devName +"␣...");
70         for(int i = 0; i<dev.getServiceList().size(); i++){
71             cp.subscribe(dev.getServiceList().getService(i));
72         }
73         drawDevPanel();
```

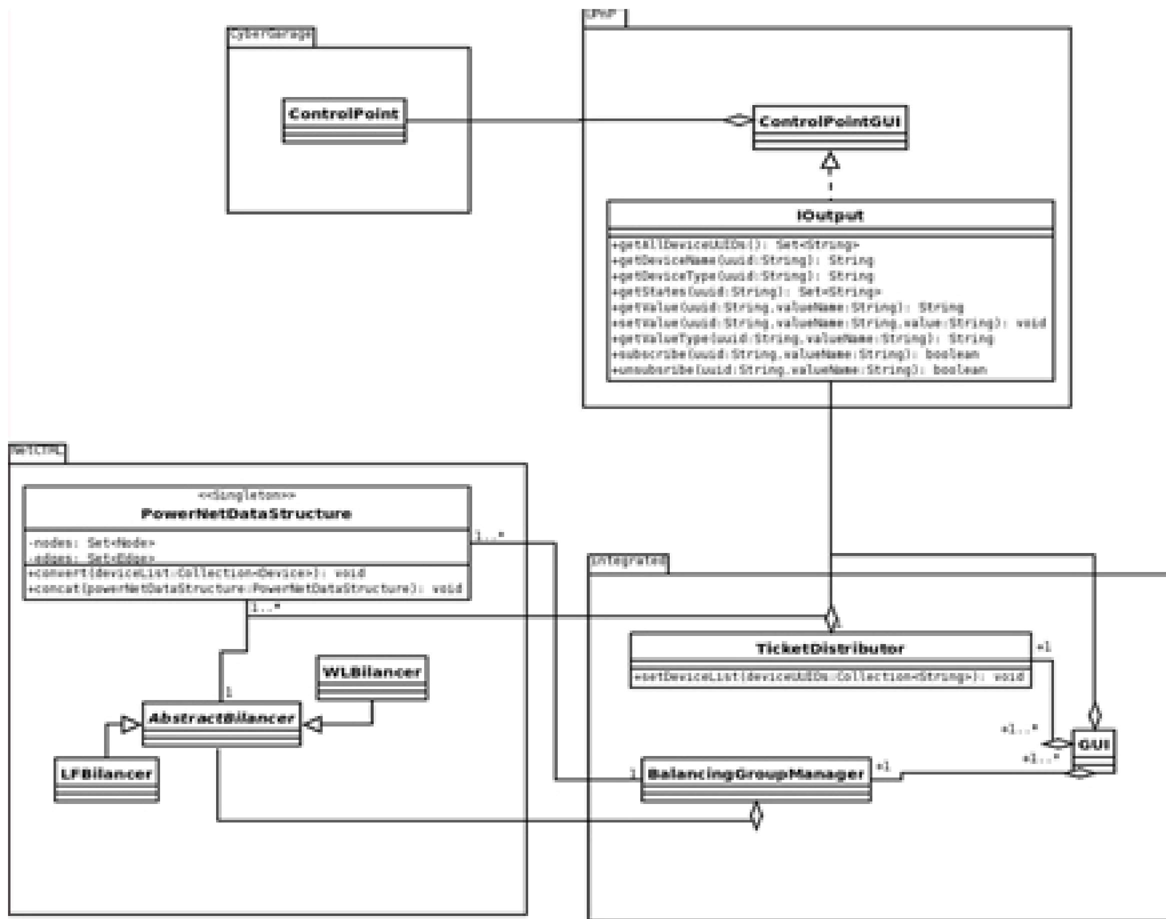


Abbildung 3.18: Klassendiagramm REAL-DEZENT

```
74            System.out.println("Device: " + devName +" added!");
75        }
76        @Override
77        public void deviceRemoved(Device dev) {
78            drawDevPanel();
79            String devName = dev.getFriendlyName();
80            System.out.println("Removed Device : " + devName+ " !!!");
81        }
82
83        public static void main (String args[]){
84            CPLogger cp =new CPLogger();
85            cp.start();
86        }
87
88
89
90
91  }
```

3.7.2 UPnP im Einsatz im Rahmen des REAL-DEZENT Projektes

Hier wird die Implementierung des REAL-DEZENT Projektes auf Basis von UPnP beschrieben. Die Implementierung von REAL-DEZENT soll das Klassendiagramm der Abbildung 3.18 verdeutlichen.

Die Klasse PowernetDataStructure enthält die Knoten und Kanten des Netzes, bildet somit die Grund-

lage zur Modellierung des Netzes und ist von zentraler Bedeutung für die LF und WL Bilanzierung. Der Aufbau der Klasse entspricht dem Entwurfsmuster des Singleton. Damit ist geährleistet, dass jeder Teil des Programms die gleiche Instanz benutzt. Die Klasse benutzt verschiedene Hashmaps. Zwei dieser Hashmaps dienen dazu aus Knoten Devices zu erzeugen und aus Devices Knoten. Mit Hilfe der Methoden getDeviceToNode() und getNodeToDevice() erhält man die dazugehörigen Hashmaps. Die anderen beiden Maps dienen zur Verwaltung der Knoten(Nodes) und Kanten (Edges). Die add(String, Node) fügt den Hashmaps Node, deviceToNode, nodeToDevice die Knoten hinzu, die für die Konvertierung von und zu UPnP und für die LF und WL Bilanzierung wichtig sind. Die Aktualisierung der Knotenmenge wird durch die Klasse TD realisiert. Dies wird durch den Aufruf der Methode Convert-DeviceToNode() verwirklicht. In dieser Methode wird die add Methode der PNDS nach Bestimmung des Node Typs aufgerufen. Die Devices erhält man durch die Methode getDeviceList().getDevice(int) in der Klasse ControlPoint der Cyberlink API. Zudem wird die PowernetDataStructure von den Balancing Group Managern benutzt um die optimalen Kosten bei einer Verhandlung zu bestimmen. Der Ticket Distributo sowie die Balancing Group Manager werden von der GUI benutzt um:

1. Devices dem Ticketdistributor hinzuzufügen
2. die Instanzen der Ticket Distributoren zu verwalten und einem Balancing Group Manager zuzuordnen.

Hierdurch ist es möglich, in der GUI dem Ticket Distributor Devices zuzuordnen und Ticket Distributoren dem Balacing Group Managern. Die Klasse CPGUI implementiert das Interface IOutput. IOutput ist die Schnittstelle über die, die Devices Information enthalten. Der Controlpoint wurde nicht implementiert sondern instanziiert (ein Controlpoint von Cyberlink). Damit können Events von Devices, die sich bei dem Controlpoint subscribed haben empfangen werden.

3.7.3 Device Generierung

Die oben beschriebene Art ein UPnP fähiges Device zu erzeugen, hat den wesentlichen Nachteil, dass für jedes Device, welches benötigt wird, eine eigene Device Klasse mit den dazugehörigen Descriptions, zusätzlich implementiert werden muss. Im Rahmen der PG wurde ein Tool implementiert, welches die Devices generisch erzeugt. Dabei kann aus jedem beliebigen Objekt ein Device erzeugt werden. Hierbei fungieren die Getter und Setter Methoden eines Objektes als Actions in der Service Description und die Parameter der Methoden als Statevariablen.

Implementierung

Die Generischen Devices werden anhand von Objekten erzeugt, dabei werden, wie bereits erwähnt, Get und Set Methoden des Obejekts als Actions des Devices abgebildet. Um die Service Description zu erstellen, werden mehrere Klassen benutzt. Zu den benutzten Klassen zählen unter anderem die GetterSetterDescriptionFactory, die mit Hilfe der Reflection API die Getter und Setter Methoden aus dem Objekt filtert und in der zugeörigen Service Description bereitstellt, die AnnotationDescription-Factory, welche Actions und Statevariablen in eine Service Description einfügen kann. Zudem gibt es die Klasse StaticServiceDescription, die, wie der Name schon vermuten lässt, statische Daten einer Service Description enthält. Diese drei Klassen sorgen für die Erzeugung der Service Description. Die Device Description wird durch die Klasse DeviceDescription erzeugt. Die Klasse DescriptionFactory erzeugt für ein Objekt anhand der vorher erwähnten Klassen die Service und Device Descriptions.

Diese wird dann der Klasse GenericUPnPDeviceFactory übergeben. Somit ist es möglich, für jedes Objekt ein Device zu erzeugen. Das Listing soll dies anhand eines kleinen Beispiels veranschaulichen:

Listing 3.7: Codebespiel zur Erzeugung eines GenericDevices

```
1  public class Klasse1 {
2  ...
3      private int i;
4      Klasse1() {
5          i = 1;
6       }
7      public void setInt(int a){
8          i = a;
9      }
10     public int getInt{
11         return i;
12     }
13 ...
14 }
15 ...
16 public class Test{
17 public static void main(String args[]){
18 GenericUPnPDeviceFactory.getInstance().createDeviceFromObject(new Klasse1()):
19 ...
20 }
21 }
```

Das obige Beispiel ist ein sehr einfaches Beispiel. Dabei wird eine Klasse Test mit nur einem Attribut und einer Set und Get Methode erstellt, die dann in der Test Klasse instantiiert und als Parameter zur Device Erzeugung genutzt wird. Dieses Device hat als Action getInt und SetInt, als Statevariable hat es nur Int. Anhand des kleinen Beispiels soll exemplarisch dargestellt werden, wie man generische Devices erzeugen kann, so wie wir sie in unserer Projektgruppe benutzten.

3.7.4 Die Annotation Description Factory und GetterSetterDescriptionFactory

Die ADF und die GSDF sorgen f ür die dynamische Erzeugung von Devices. Die GetterSetterDescriptionFactory benutzt dabei die Getter und Setter Methoden um die Services w ährend der Laufzeit zu erzeugen. Dabei werden aus den Getter Methoden State Variablen und aus den Setter Methoden werden Actions 3.4.1. Die Methoden werden dazu in verschiedenen Listen gespeichert und gesondert behandelt. Den Name der State Variable erh ält man durch den Aufruf:

```
1 String stateVarName = methodName.substring(3);
```

Dabei wird der String “get“ der Methoden abgetrennt. Also aus getIstWirkleistung wird die State Variable IstWirkleistung generiert. Bei den SetMethoden entspricht der Name der Actions den Namen der Methoden. Einen etwas anderen Namen geht die AnnotationDescriptionFactory. Hierbei werden die Actions und die State Variablen anhand von Annotationen generiert. Bei dieser Methode k önnen nicht nur Get und Set Methoden, sondern auch Methoden die annotiert sind in die Description on-fly eingef ügt werden.

3.8 UPnP Leistungstest

Da Real-Dezent eine zeitkritische Anwendung ist, muss sichergestellt werden, dass die Regelung der Energie innerhalb einer garantierten oberen Schranke erfolgt. Die gegebene obere Schranke beträgt im Real-Dezent Projekt 500 ms. Innerhalb dieser 500 ms müssen die Berechnungen, Kommunikationen und Verhandlungen abgeschlossen sein. Daher haben wir das Kommunikationsverhalten von UPnP getestet um eventuell eine obere Schranke an Devices zu finden, in der die Kommunikation ein bestimmtes Zeitfenster nicht überschreitet.

3.8.1 Testumgebung

Für die Leistungstests standen uns drei Rechner zur Verfügung. Diese waren durch einen Switch mit einer Übertragungsrate von 100 MBit/s miteinander verbunden. An zwei Rechnern wurden jeweils Devices erzeugt und gestartet. Den dritten Rechner benutzten wir als Controlpoint, an dem sich die Devices anmelden konnten. Um die eingehenden Daten zu visualisieren, wurde die JFreeChart API [20] benutzt. Die JFreeChart API ist ein Framework um Daten in einem Diagramm anzuzeigen. Die Tests beliefen sich auf ca. 30 min., da die Visualisierung mit der Datenmenge nach dieser Zeit nicht mehr zu recht kam. Das Resultat war, dass die anzuzeigende Kurve nicht mehr so dargestellt wurde, wie sie sein sollte. Die Abbildung 3.19 zeigt ca. 50 Sinuskurven, bei denen dieser Effekt auftrat.

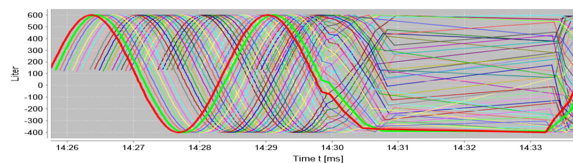


Abbildung 3.19: Sinuskurven für 50 Devices

Die erhaltenen Daten wurden exportiert, um spätere Untersuchungen durchführen zu können. Da die Untersuchung von 50 Devices sehr unübersichtlich ist, wurde ein Tool entwickelt, der die Graphen einzeln darstellt.

3.8.2 Tests

Die erzeugten Devices waren sogenannte Sinusgeneratoren. Die Devices lösten Events aus, welche Sinuswerte darstellten. Die Events wurden in einem Intervall von 250 ms gesendet. Dieser Zeitrahmen wurde gewählt, da Real-Dezent in einem Zyklus von 500 Sekunden die Versorgung sichern soll. Wir starteten 50 Devices von den unterschiedlichen Rechnern und stellten in späteren Untersuchungen fest, dass es vereinzelt einige Verzögerung von ca. 500 ms gab bei der Ankunft neuer Events gab. Abbildung 3.20 zeigt die Sinuskurve eines Devices.

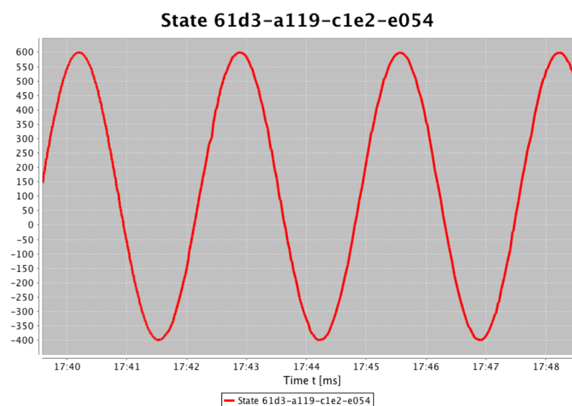


Abbildung 3.20: Sinuskurve eines Devices

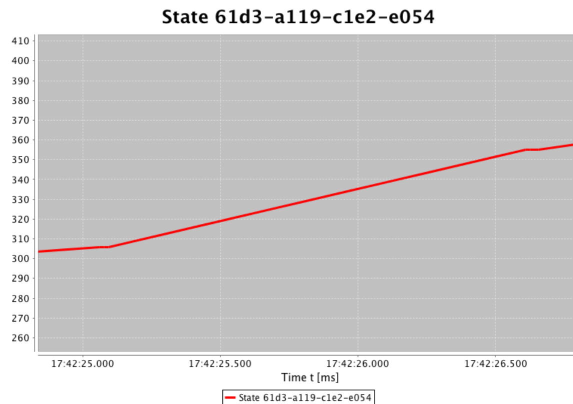


Abbildung 3.21: Ausschnitt

Abbilung 3.21 zeigt einen Ausschnitt des Graphen, bei der eine Verzögerung von ca. 1 sek. zu sehen ist. Solche Verzögerungen, so sporadisch sie auch auftauchen, gefährden mit den bis hierhin zur Verfügung gestellten Möglichkeiten das Real-Dezent Projekt, da Verhandlungen oder Regelungen zu spät erfolgen und wir mit dieser Verzögerung die Zeit einer Verhandlungsrunde mehr als doppelt überschritten hätten.

3.8.3 Ergebnisse

Die Tests zeigen, dass die Kommunikation der Devices und die Regelung des Netzes auf diese Weise nicht gewährleistet werden kann, wenn die Anzahl der zu regulierenden Teilnehmer ca. 50 überschreiten. Bei Tests mit weniger Devices (ca. 35-40) konnte ein ähnliches Verhalten beobachtet werden. Dies kann verschiedene Gründe haben. Daher ist es empfehlenswert, die Anzahl der Devices in einem Subnetz so gering wie möglich zu halten, um Paketverzögerungen zu vermeiden. Tests mit einer größeren Anzahl an Rechnern konnten nicht durchgeführt werden. Diese sind notwendig um die Gründe der Verzögerungen einzukreisen.

3.9 Hardwaresimulation

3.9.1 Simulation mit der ContainerGUI

Um die Hardwaren zu regeln, einzustellen und zu testen, müssen wir aus dem Sicherheitsgrund zuerst die eingesetzten Hardwaren simulieren. Zu unserem Projekt haben wir für die folgenden Hardwaren, nämlich die Synchongeneratoren, Asychrongeneratoren, die entsprechende Netzwerkumgebung, eine Simulation durchzuführen.

In diesem Kapitel werden einige wichtige Komponenten für die Hardwarensimulation erläutert. Die Klasse `ContainerGUI`, die zur Kontrolle der verschiedenen Generatoren in Hardwaresimulation dient, und die simulierten verschiedenen Generatoren sowie die den ensprechenden Generatorverhalten.

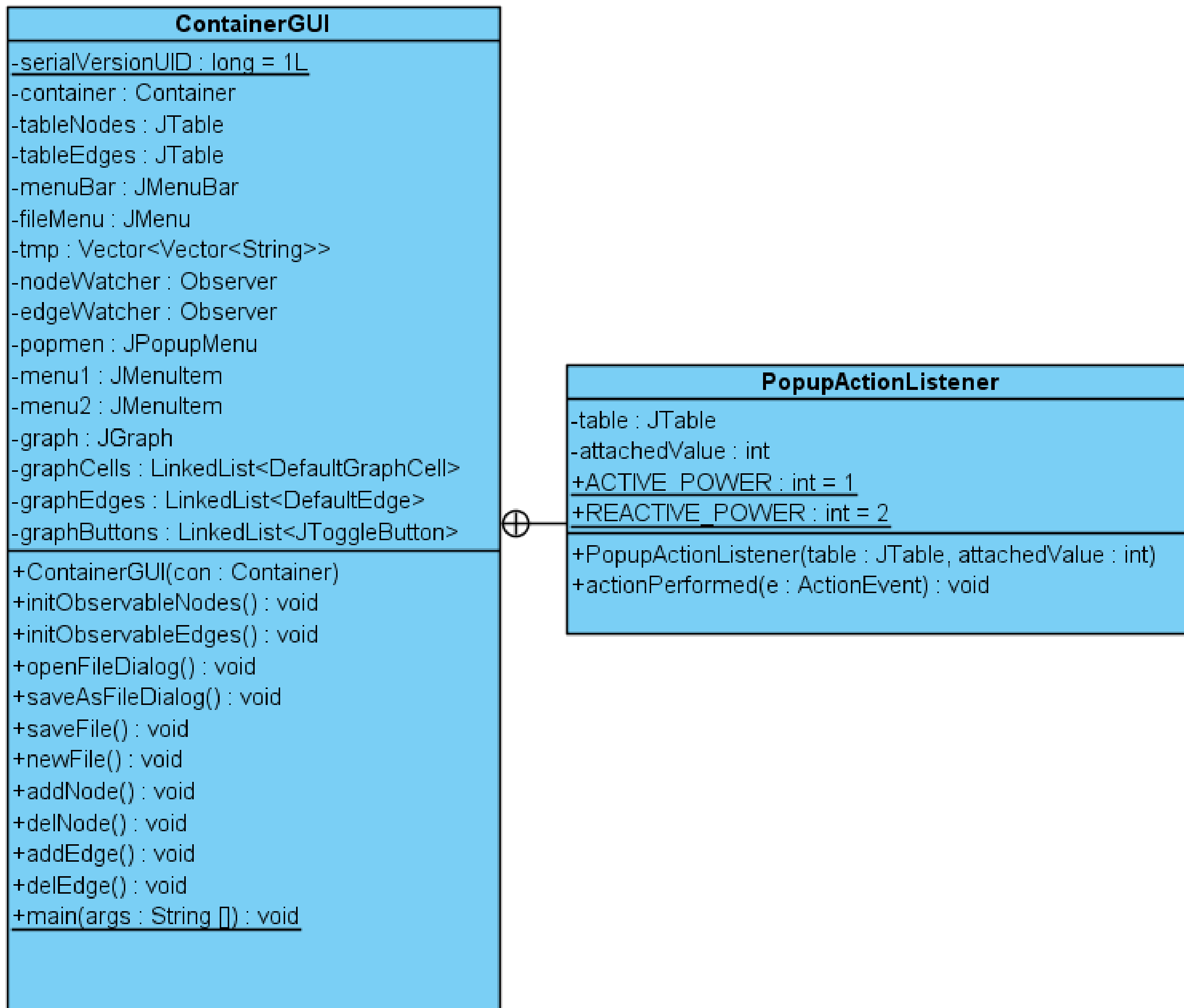


Abbildung 3.22: Klassendiagram von ContainerGUI

3.9.2 ContainerGUI

Die Klasse `ContainerGUI` implementiert einen Generatorerzeuger in der UPnP-Umgebung, mit dem verschiedene Generator zur Simulation erzeugt und manipuliert werden können. Die entsprechenden Parameter von verschiedenen Generatoren sind auch direkt in `ContainerGUI` zu addieren und zu verändern.

Die Abbildung 3.22 zeigt die Klassendiagramm von der Klasse `ContainerGUI`, die noch eine Subklasse `PopupActionListener` enthält.
Das zu simulierende Netz besteht aus Kanten und Knoten. Insgesamt gibt es für die Knoten vier verschiedene Type: SynchronGenerator, AsynchroGenerator, Consumer und Inverter. Die Kanten im virtuellen Netz simulieren die elektronischen Leitungen zwischen den Knoten. In der Benutzeroberfläche von `ContainerGUI` können die Knoten verschiedener Typen und die Kanten zwischen beliebigen Konten generiert, verwaltet und entfernt werden, indem die Knoten und Kanten als UPnP

Devices implementiert werden. Auch die graphische Darstellung von Knoten und Kanten ist möglich.

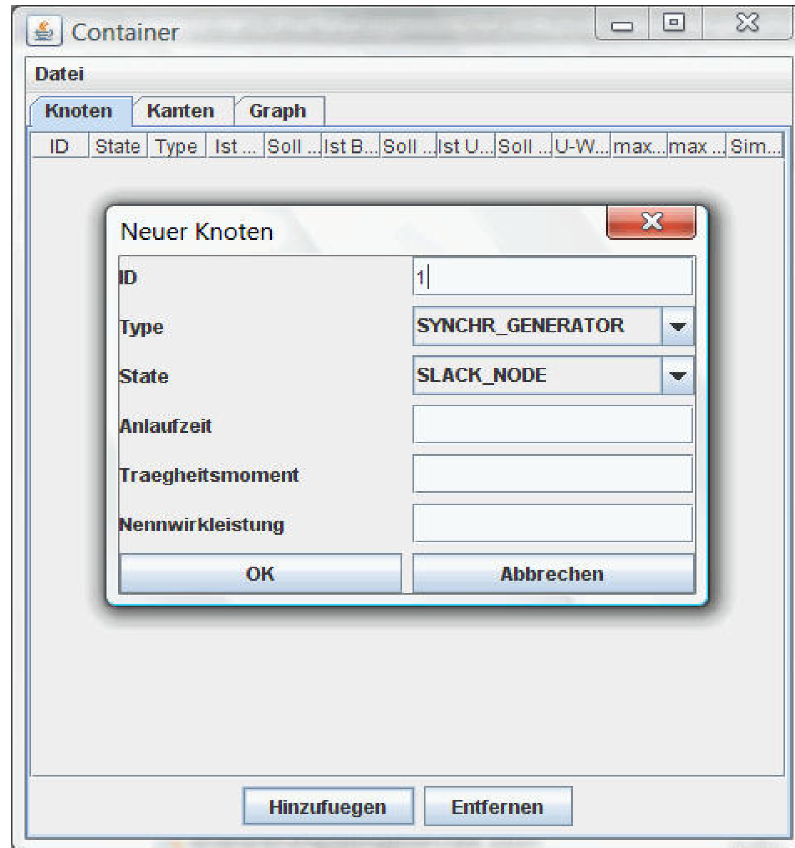


Abbildung 3.23: Erzeugen des neuen Knoten

Wie in Abbildung 3.23 gezeigt, sollen das ID ,der Typ

- *sychrone Generatoren*,
- *asynchrone Generatoren*,
- *Consumer*
- *Inverter*

und dessen Zustände

- *SLACK_Node oder sogenannten Refenrenzknoten*
- *PQ_Node*
- *PU_Node*

beim Erzeugen eines neuen Knoten bestimmt werden. Für synchrone Generatoren kann man auch die Anlaufzeit, den Trägheitsmoment und die Nennwirkleistung einstellen. Die Parameter der erzeugten Knoten bzw. Kanten werden in Tabelle angezeigt.

Container

Datei

Knoten | Kanten | Graph

ID	State	Type	Ist Wirkl.	Soll Wirkl.	Ist Blindl.	Soll Blindl.	Ist U-Betrag	Soll U-Betr...	U-Winkel	max Blind ...	max Blind ...	Simulation
1	SLACK_N...	SYNCHR...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-10.0	10.0	☐
2	PU_NODE	SYNCHR...	0.0	1.5	0.0	0.0	1.0	1.0	0.0	-8.0	8.0	☐
3	PU_NODE	ASYNCH...	0.0	3.0	0.0	0.0	2.0	2.0	0.0	-5.0	5.0	☐
4	PQ_NODE	CONSUM...	0.0	3.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	☐
5	PU_NODE	CONSUM...	0.0	2.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	☐

Hinzufuegen | Entfernen

Abbildung 3.24: die Knoten bei Simulation

Durch den Doppelclick in Tabelle kann man einzelnen Parameter von Konten oder Kanten einstellen.

In diesem Fenster `Container`(sieht in Abbildung 3.24) entstehen drei Tabs

- *Knoten*
- *Kanten*
- *Graph*

und ein Menü *Datei*. Das Menü `Datei` unterstützt die HWS Konfiguration Datei erneut zu erzeugen, abzuspeichern, sowie zu öffnen und zu schließen. Im Tab *Knoten* werden alle erzeugten Knoten und deren Parameter

- *ID*: Index der Knoten
- *State*: Zustand der Knoten wie vorher vorgestellt.
- *Type*: Typ der Knoten
- *Ist Wirkl.*: Ist-Wirkleistung der Knoten
- *Soll Wirkl.*: Soll-Wirkleistung der Knoten
- *Soll Blindl.*: Soll-Blindleistung der Knoten
- *Ist U-Betrag*: Ist-Spannungsbetrag der Knoten

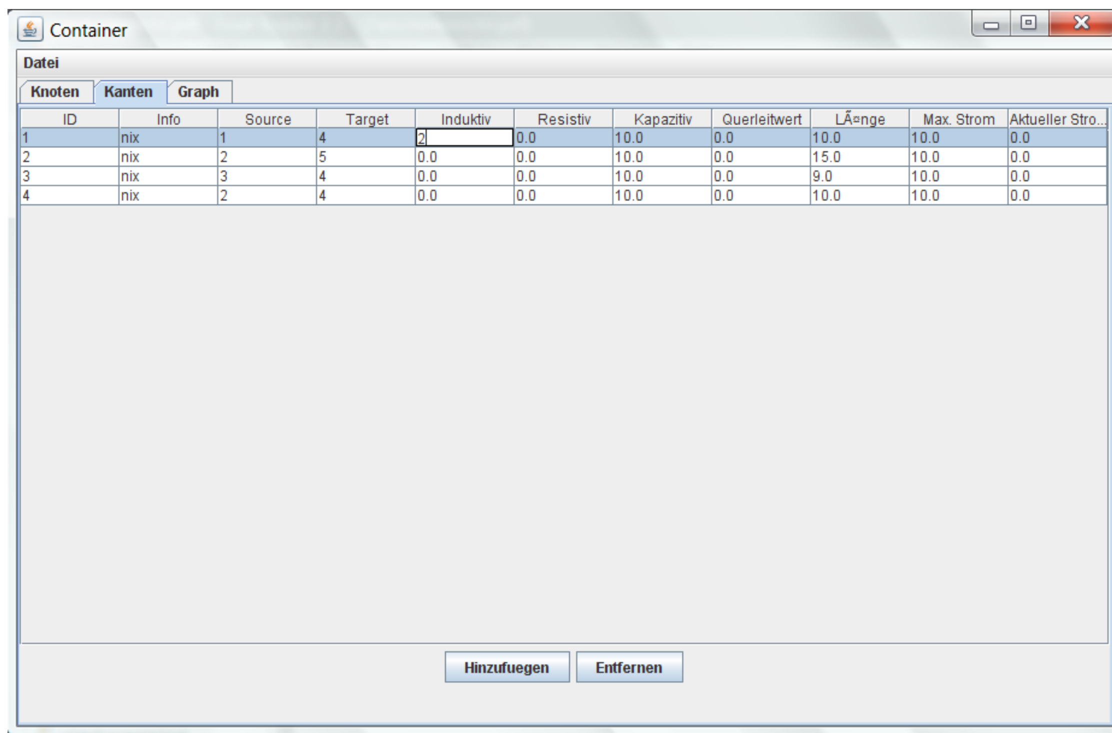


Abbildung 3.25: die Kanten bei Simulation

- *Soll U-Betrag*: Soll-Spannungsbetrag der Knoten
- *U-Winkel*: der Spannungswinkel
- *max Blind ent*: maximale entnommene Blindleistung der Knoten
- *max Blind ein*: maximale eingespeiste Blindleistung der Knoten

in einer Tabelle aufgelistet. Einzelne Parameter in der Tabelle sind mit Doppelklick vor der Simulation einzustellen. Die letzte Spalte ist eine Option für die Simulation. Die angekreuzten Devices(Knoten oder Kanten) werden später in der Simulation erscheinen.
Das Entfernen des ausgewählten Knoten erfolgt durch den Button *Entfernen*, der sich am Boden des Fensters befindet.

Wie die Knoten, werden die Kanten auch in einer ähnlichen Form dargestellt (sieht in Abbildung 3.9.2). Sie besitzt die folgende Parameter

- *ID*: Index der Kanten. Die ist beim Hinzufügen automatisch generiert.
- *Info*: die Information über die Kanten
- *Source*: der Quellknoten der Kanten
- *Target*: der Zielknoten der Kanten
- *Induktiv*: Imaginärteil der Admitanz[10] der Kanten

[10] sieht in Abschnitt Lastflußberechnung

- *Resistiv*: Realteil der Admitanz der der Kanten
- *Kapazitiv*: Kapazität der der Kanten
- *QuerLeitwert*: der Querleitwert der Kanten
- *Länge*:die Länge der Kanten
- *Max. Strom*: maximaler Strom, der durch die Kanten fließt
- *Aktuelle Strom*: aktueller Strom in der Kanten

Wie bei Knoten können die oben genannten Parameter separat per Doppelklick in der Tabelle modifiziert werden. z.B. *Source*, *Target*, *Induktiv*, *Resistiv* und *Länge* usw. Hinzufügen bzw. Entfernen von Kanten erfolgen durch Klick auf die entsprechende Buttons.

3.9.3 Generatorverhalten für die Simulation

Für die Simulation werden einige unterschiedliche Generatoren implementiert, welche bei der Simulation von realen Generatoren sowie bei der Analyse bzw. Testen der Implementierung des DEZENT-Algorithmus hilfreich sind.

ConstantGenerator

Ein Generator, der nur die Leistungen mit Konstantwert liefert, heisst ConstantGenerator. In der Simulation benutzen wir solche ConstantGenerator um die Sychronmaschine zu simulieren, die das Netz ständig mit einer fixen Leistung versorgt.

Sourcecode von ConstantGenerator:

```
package edu.udo.dezent.upnp.container.generator;
public class ConstantGenerator implements Generator {

private double value;

public ConstantGenerator(double value) {
this.value = value;
}

public double calculateNextValue() {
return value;
}

public double getValue() {
return value;
```

```
}

public void setValue(double value) {
this.value = value;
}
}
```

Das Klassendiagramm von ConstantGenerator sieht wie folgend aus:

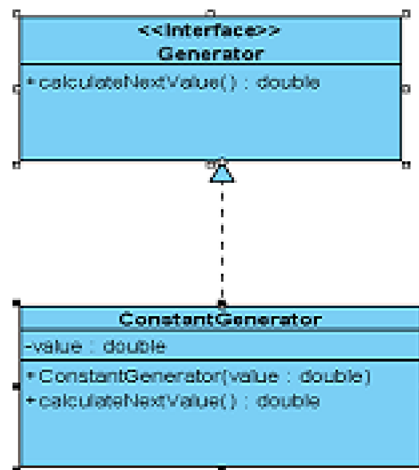


Abbildung 3.26: Klassendiagramm des ConstantGenerator

RandomGenerator

Zur Ergänzung von ConstantGenerator haben wir einen anderen Generator eingeführt, der sich RandomGenerator nennt. Im Vergleich zum Constantgenerator generiert der RandomGenerator nur die randomisierten Leistungswerten. Bei der Implementierung werden eine minimale und eine maximale double Zahl als Grenze angegeben. Zwischen den beiden Zahlen wählen wir mit einer Randomfunktion einen Wert aus als der zu erzeugende Leistungswert. Da die ins Netz eingespeisten Leistungen in der Realwelt häufig variieren und nicht fest sind,setzen wir den RandomGenerator ein ,um diese Situation zu simulieren.

Sourcecode von RandomGenerator:

```
package edu.udo.dezent.upnp.container.generator;

public class RandomGenerator implements Generator {

private double max, min;

public RandomGenerator() {
```

```
max = 10;
min = -10;
}

public RandomGenerator(double min, double max) {
this.max = max;
this.min = min;
}


public double calculateNextValue() {
return (Math.random()*(max-min))+min;
}

public double getMax() {
return max;
}

public void setMax(double max) {
this.max = max;
}

public double getMin() {
return min;
}

public void setMin(double min) {
this.min = min;
}

}
```

Das Klassendiagramm sieht wie folgt aus:

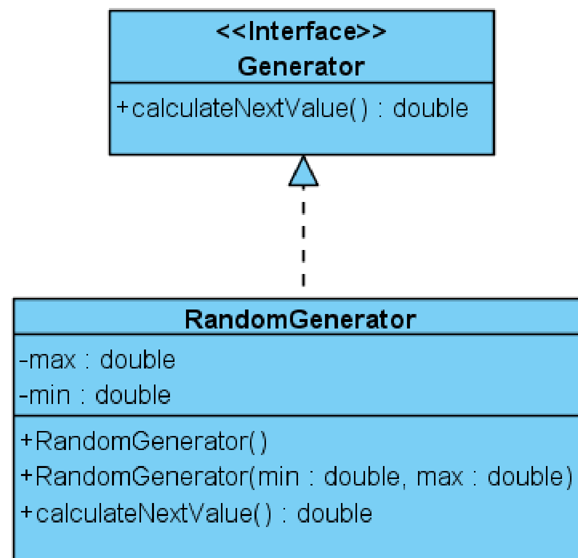


Abbildung 3.27: Klassendiagramm RandomGenerator

LocalRandomGenerator

LocalRandomGenerator ist eine Erweiterung des RandomGenerators. Bei der Erzeugung der Zufallswerte wird eine Variable Step eingeführt, mit der man innerhalb des vorgegebenen Min-Maxwert Intervalls den Sollleistungswert auf Basis des alten Wertes in eine bestimmte Richtung vergrößern oder verkleinern kann.

Die Sourcecode von LocalRandomGenerator:

```
package edu.udo.dezent.upnp.container.generator;
public class LocalRandomGenerator implements Generator {
private double max, min;
private double value;
private double step;

public LocalRandomGenerator() {
max = 10;
min = -10;
value = 0;
step = 2;
}
public LocalRandomGenerator(double max, double min,
                            double value, double step) {
this.max = max;
this.min = min;
this.value = value;
this.step = step;
```

```
}
public double calculateNextValue() {
double oldValue = value;
double maxStep = step * 0.5; double minStep = -step * 0.5;
    value = value + (Math.random() * (maxStep-minStep)) + minStep;

    if (value<=max & value>=min)
    return value;

    value = oldValue;
    return oldValue;
}
public double getMax() {
return max;
}
public void setMax(double max) {
this.max = max;
}
public double getMin() {
return min;
}
public void setMin(double min) {
this.min = min;
}
public double getStep() {
return step;
}
public void setStep(double step) {
this.step = step;
}
}
```

Klassendiagramm:

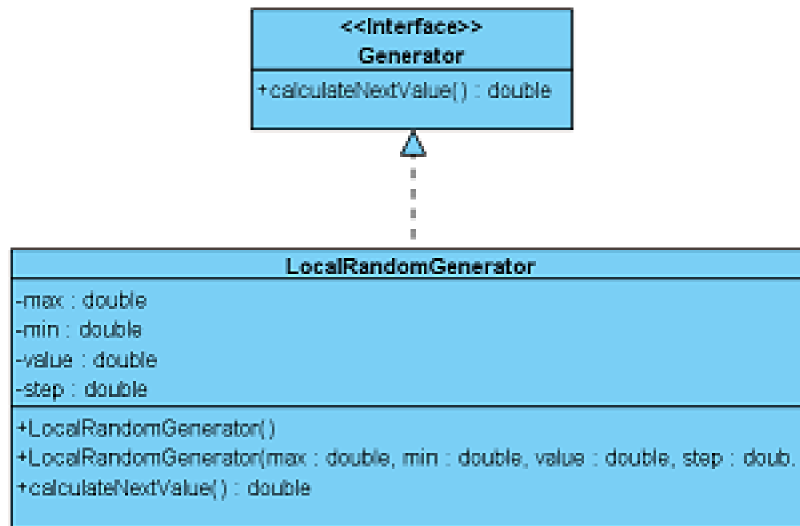


Abbildung 3.28: Klassendiagramm LocalRandomGenerater

SinusGenerator

Um die Kapazität unseres Computernetz für die Simulation zu testen, haben wir einen sogenannten Sinusgenerator implementiert. Dieser erbt die Klasse `Generator` und generiert die Leistungen,deren Werte mit den Werten der Sinusfunktion übereinstimmen. Durch ein Experiment zur Betimmung der maximalen Anzahl von Sinusgeneratoren wissen wir, wie viele Generatoren oder UPnP Devices unser Netz maximal simmulieren kann. Und durch die Beobachtung der graphischen Kurven, die wir durch unser Experiment erhielten, können wir Rückschlüsse auf Paketkollisionen schließen.

Darunter ist der Sourcecode von Sinusgenerator:

```
package edu.udo.dezent.upnp.container.generator;
public class SinusGenerator implements Generator {

private double amplitude;
private double frequence;
private double angle;
private double yDefiation; // Y - Abweichung
private double step;



public SinusGenerator() {
amplitude = 1;
frequence = 1;
angle = 0;
yDefiation = 0;
```

```
step = 0.1;
}

public SinusGenerator(double amplitude, double frequence,
                      double yDefiation, double step) {
this.amplitude = amplitude;
this.frequence = frequence;
this.angle = 0;
this.yDefiation = yDefiation;
this.step = step;
}


public double calculateNextValue() {
angle += step;
double value=amplitude*Math.sin(angle*(2*Math.PI*frequence));
return (value + yDefiation);
}

public double getAmplitude() {
return amplitude;
}
  public void setAmplitude(double amplitude) {
this.amplitude = amplitude;
}
  public double getFrequence() {
return frequence;
}
  public void setFrequence(double frequence) {
this.frequence = frequence;
}
  public double getAngle() {
return angle;
}
  public void setAngle(double angle) {
this.angle = angle;
}
  public double getYDefiation() {
return yDefiation;
}
  public void setYDefiation(double defiation) {
yDefiation = defiation;
}
  public double getStep() {
return step;
}
  public void setStep(double step) {
```

```
this.step = step;
}

}
```

Aus dem Experiment erhalten wir die maximale Anzahl von Sinusgeneratoren, die von unserem Simulationsnetz unterstützt werden. Diese beträgt 50, also 50 UPnP Devices. Das folgende Diagramm zeigt,wie die Werte von 4 Sinusgeneratoren (UPnP Devices)im Simulationsnetz aussehen.

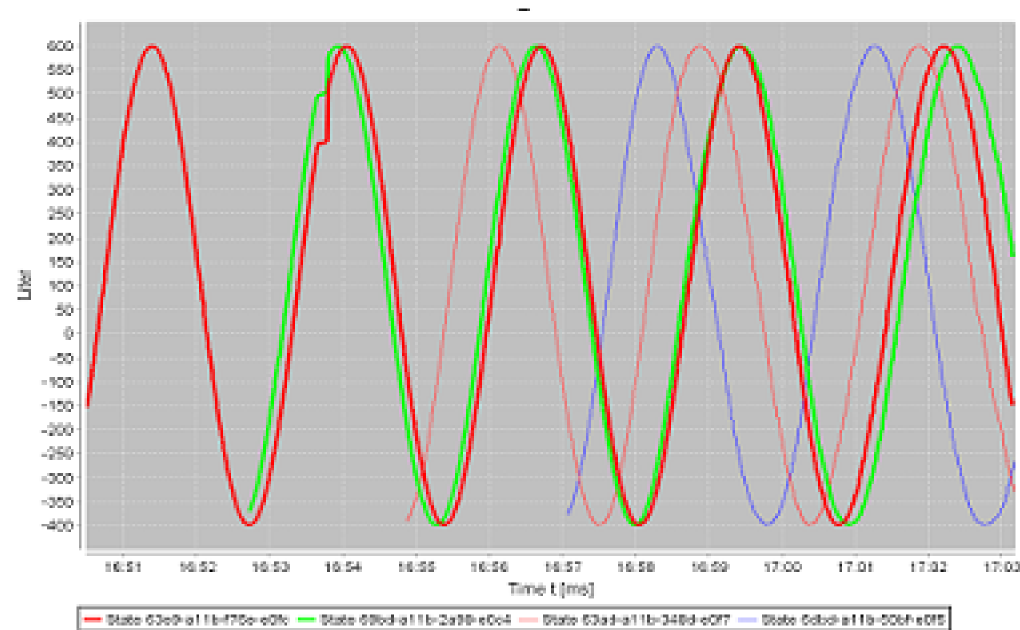


Abbildung 3.29: Klassendiagramm SinusGenerator

Im Graph ist zu sehen, dass die vier Sinuskurven nacheinander erkannt werden. Und zwischen den Zeiträumen 16:53 und 16:54 hat die rote Kurve eine kleine Verzögerung . Diese Verzögerung ist in diesem Experiment nicht so schlimm, weil die kurve später wieder in Ordnung ist. Aber wenn mehrere Sinusgeneratoren zeitgleich ankommen(z.B.50), dann könnte so eine Verzögerung die ganze Simulation lahmlegen.

Winddaten abspielen

Um die Anwendung von Windenergie zu simulieren, haben wir durch Vererbung der Klasse Generator einen Windgerator implementiert. Mit diesem Generator ist es möglich, die Leistungen, die durch Windkraft erzeugt werden, ins Stromnetz einzuspeisen. Da Windkraft variiert, müssen wir zuerst statistische Daten über Windkraft sammeln, um die von Wind erzeugten Leistungen auszurechnen. Diese Daten enthalten die wichtigen Parameter zur Berechnung von Windkraft. Die sind z.B. die Windgeschwindigkeit zu den entsprechenden Zeitpunkten. In unserem Projekt werden solche Daten in CVS Dateien abgespeichert. Zum Auslesen solcher Daten wird ein CSVParser entwicklt, der die Windaten aus den CVS Dateien ausliest und an den Windgenerator übermittelt. Bei der Simulation wird

der Windgenerator mit diesen von den CVS Dateien übermittelten Daten die Leistungen berechnen ,steuern und ins simulierte Stromnetz einspeisen.

Sourcecode von Windgenerator:

```
package edu.udo.dezent.upnp.container.generator;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Vector;

public class WindGenerator implements Generator {

private Vector<Double> data;
private int index;
private String file;

public WindGenerator(String csvfilename) {
file = csvfilename;
index = 0;
java.io.FileReader fr;
try {
fr = new java.io.FileReader(csvfilename);
java.io.BufferedReader br = new java.io.BufferedReader(fr);

System.out.println("Reading Wind...");
data = new Vector<Double>();
// Initial Capacity (1 Year), Capacity Increment (1 Day).

while(br.ready()) {
String line = br.readLine();
String cols[] = line.split(";");
// Komma nach Period und dann englisch parsen.
data.add( new Double( Double.parseDouble
                     (cols[1].replace(",", "."))));
}
fr.close();
System.out.println("...Done.");
} catch (Exception e) {
System.out.println(e.getMessage());
}

}
public double calculateNextValue() {
index = (index + 1) % data.size();
return Math.pow(data.get(index), 3) / 1000;
}
```

```
public String getFile() {
return file;
}

public void setFile(String file) {
this.file = file;
}

}
```

Klassendiagramm von Windgenerator:

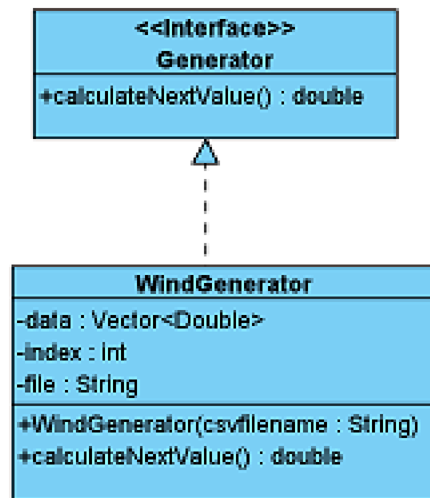


Abbildung 3.30: Klassendiagram WindGenerator

Realverhalten anhand des Nennwertes

Um das Realverhalten von Sychronmaschinen anhand des Nennwertes zu testen, haben wir einen RealBehaviourGenerator definiert.Wenn der Sollleistungswert und Istleistungswert vom Synchrongenerator in der Simulation nicht identisch sind, wird die Differenz berechnet und als ein Faktor zur Berechung des nächsten Sollleistungswertes angewendet.

Sourcecode von RealBehaviourGenerator:

```
package edu.udo.dezent.upnp.container.generator;

import edu.udo.cs.ls3.DEZENT.powerNet.Nodes.SynchroGenerator;


public class RealBehaviourGenerator implements Generator {
```

```
private SynchroGenerator node;
double simTime = 1; // in ms
double oldIstWirkleistung = 0;
double falt = 50;
private double oldSollWirkleistung = 0;
private double deltaT = 0D ;
private double neuP = 0D;
private double deltaP = 0D;
private int counter = 0;

public RealBehaviourGenerator(SynchroGenerator node,
                                     double simTime) {
this.node = node;
this.simTime = simTime;
oldIstWirkleistung = node.getIstWirkleistung();
oldSollWirkleistung = node.getSollWirkleistung();
node.setFrequenz(falt);
}

public double calculateNextValue() {
double t = node.getTraegheitsmoment();
if (oldSollWirkleistung == node.getSollWirkleistung()
                                     & counter > 0) {

neuP = ((simTime / deltaT) * deltaP);
node.setFrequenz(falt);
double neuF = node.adjustFrequenz(neuP, simTime);
falt = neuF;
counter--;

} else {

deltaP = node.getSollWirkleistung()-node.getIstWirkleistung();
deltaT = Math.abs(((t/2)*Math.pow(2*Math.PI*falt,2))/deltaP);
oldSollWirkleistung = node.getSollWirkleistung();
float tmp = (float)deltaT / (float)simTime;
counter = Math.round(tmp);
neuP = 0;
}
return node.getIstWirkleistung() + neuP;
}
}
```

Das Diagramm von RealBehaviourGenerator ist wie folgend:

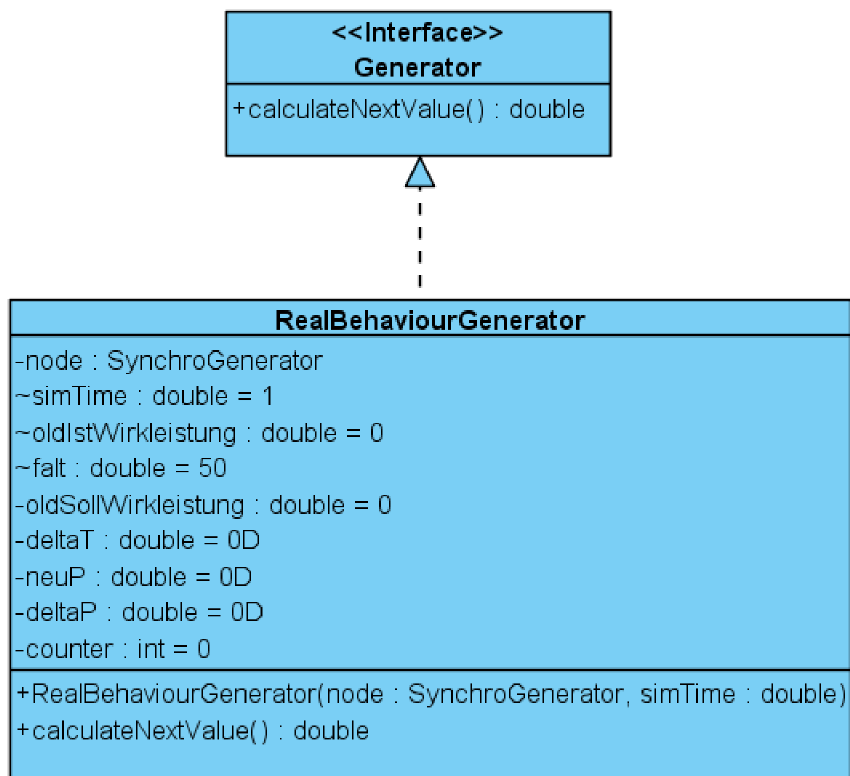


Abbildung 3.31: Klassendiagramm RealBehaviourGenerator

Kapitel 4

Experimente

4.1 Ziele

Ziel der Projektgruppe Real-Dezent ist es, die theoretisch entwickelten und simulierten Algorithmen in einer realen Umgebung zu testen. Dazu wird die Laborumgebung verwendet, die in Kapitel 2.3 beschrieben wurde.

Die Projektgruppe hat zur Netzregelung einen Algorithmus zur Wirkleistungsbilanzierung entwickelt. Wird dieser auf ein reales Energienetz aufgesetzt, so sollte die Wirkleistungsbilanz dieses Netzes stets ausgeglichen sein. Dies ist eine Voraussetzung für einen späteren Inselnetzbetrieb.

Aufgrund äußerst ungenauer Messgeräte mit großen Schwankungen u.a. der Messwerte für die Wirkleistung kann dieses Ziel in der Laborumgebung nicht erreicht werden. Daher werden die Erwartungen an die Experimente darauf beschränkt, dass ein von der Wirkleistungsbilanzierung geregeltes Netz deutlich weniger Leistung von außerhalb beziehen bzw. abgeben muss, als ein ungeregeltes Netz. Dabei sollen einige der im Netz vorhandenen Erzeuger reale Energieerzeuger wie Windkraftanlagen oder Solaranlagen simulieren, und die Verbraucher das Leistungsprofil realer Verbraucher nachbilden.

Für einen aussagekräftigen Test des Dezent-Algorithmus bietet die Laborumgebung zu wenig regelbare Erzeuger und Konsumenten, so dass ein Experiment zum Testen dieses Algorithmus' keinen Sinn macht.

4.2 Beschreibung

4.2.1 Umgebung

Die Experimente finden in der Laborumgebung statt. Zum Einsatz kommen zwei Synchrongeneratoren, ein Asynchrongenerator, ein dreiphasiger Verbraucher sowie ein einphasiges Gleichstromaggregat mit nachgeschaltetem Wechselrichter.

Die Asynchronmaschine simuliert eine Windkraftanlage. Dazu wird ihre Leistung entsprechend eines real gemessenen Windprofils angepasst. Ebenso simuliert das Gleichstromaggregat eine Solaranlage und der Verbraucher reale Konsumenten.

Es werden zwei verschiedene Versuche durchgeführt: in einem Versuch A wird die Wirkleistungsbilanzierung dazu verwendet, die Synchrongeneratoren so zu steuern, dass sie die aus den Profilen resultierenden Leistungsschwankungen ausgleichen. In einem zweiten Versuch B werden die Synchronmaschinen nicht geregelt, sondern geben eine konstante Leistung ab.

Die maximale Wirkleistung im Profil der Asynchronmaschine beträgt 300 Watt, die des Gleichstromaggregats 500 Watt. Die Synchronmaschinen werden mit jeweils 600 Watt nach oben begrenzt.

4.2.2 Steuerung der Maschinen und Messwertaufnahme

Das steuern der Maschinen, die ein statisches Leistungsprofil abfahren, erfolgt direkt über die jeweiligen Kontroll-PCs.

Die Regelung der Synchrongeneratoren in Versuch A erfolgt über UPnP : auf einem beliebigen Netzwerkrechner wird die Wirkleistungsbilanzierung gestartet. Die von der Bilanzierung errechneten Sollwerte werden über UPnP an die Kontrollrechner der Synchronmaschinen übertragen und von dort an die Steuerungs-SPS.

Die Messwerte werden direkt an den Maschinen von Sineax-Messgeräten aufgenommen und über die SPS an den Steuerungsrechner weitergegeben. Dieser stellt die Daten über UPnP im Netzwerk zur Verfügung. Der für das Logging zuständige Rechner meldet sich über UPnP für die Messwerte an und wird so über alle Änderungen informiert. Alle Messwerte werden in Log-Dateien geschrieben. Nicht nur die Istwerte der Maschinen werden so an den Logger übertragen, sondern auch die vom Algorithmus vorgegebenen Sollwerte.

4.2.3 Simulationsdaten

Jeder Versuch dauert 30 Minuten. In diesem Zeitraum werden Leistungsprofile abgefahren, die 24 Stunden in der Realität entsprechen. Auf ein Signal hin, das über UPnP versendet wird, starten alle Teilsimulationen gleichzeitig.

4.3 Ergebnisse

Abb. 4.1, 4.2 und 4.3 zeigen die Leistungskurven der Geräte, die ein Leistungsprofil fahren. Da diese Kurven in beiden Versuchen nahezu identisch sind, werden nur die Kurven aus Versuch A (geregelt) gezeigt. Auch ist nur eine Phase des Verbrauchers dargestellt, da die anderen beiden Phasen zu der ersten identisch sind.

4.3.1 Versuch A: geregelt

Bei diesem Versuch wurden die beiden Synchronmaschinen von der Wirkleistungsbilanzierung geregelt. Abb. 4.4 und 4.5 zeigen die von ihnen produzierte Wirkleistung und die vom Algorithmus vorgegebenen Sollwerte.

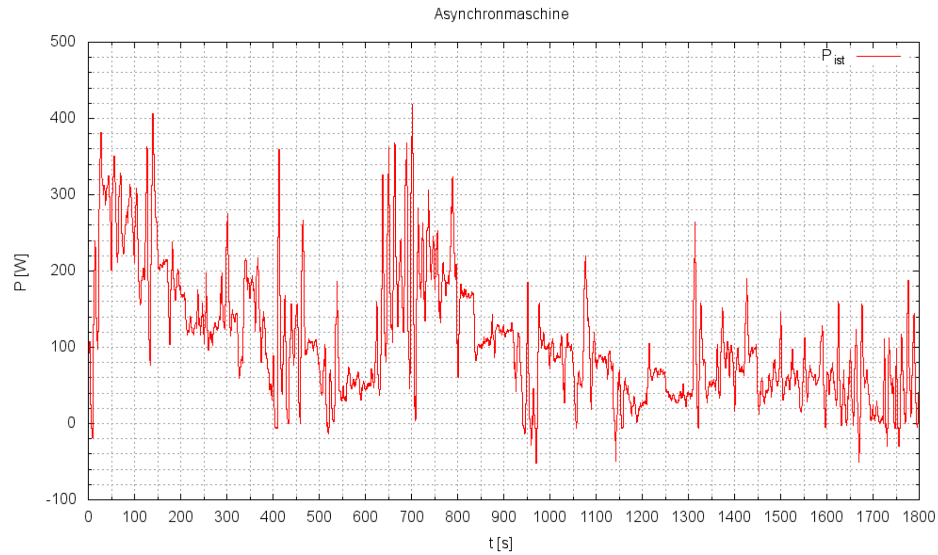


Abbildung 4.1: Asynchronmaschine (Windkraftanlage) geregelt

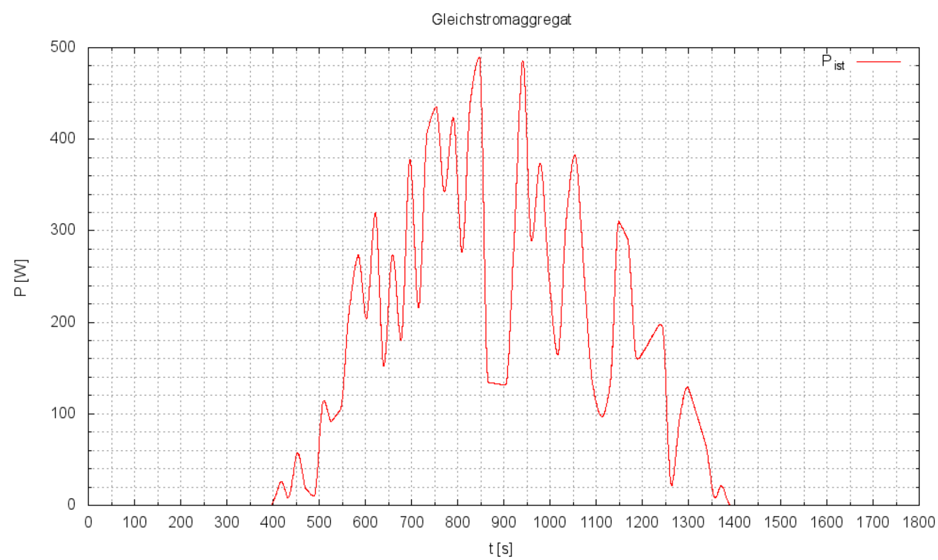


Abbildung 4.2: Gleichstromaggregat (Solaranlage) geregelt

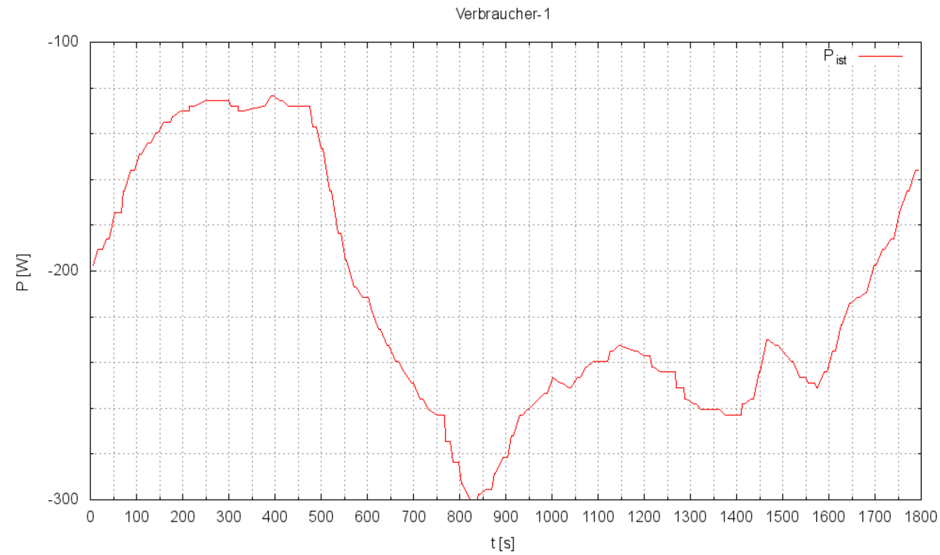


Abbildung 4.3: Verbraucher 1 geregelt

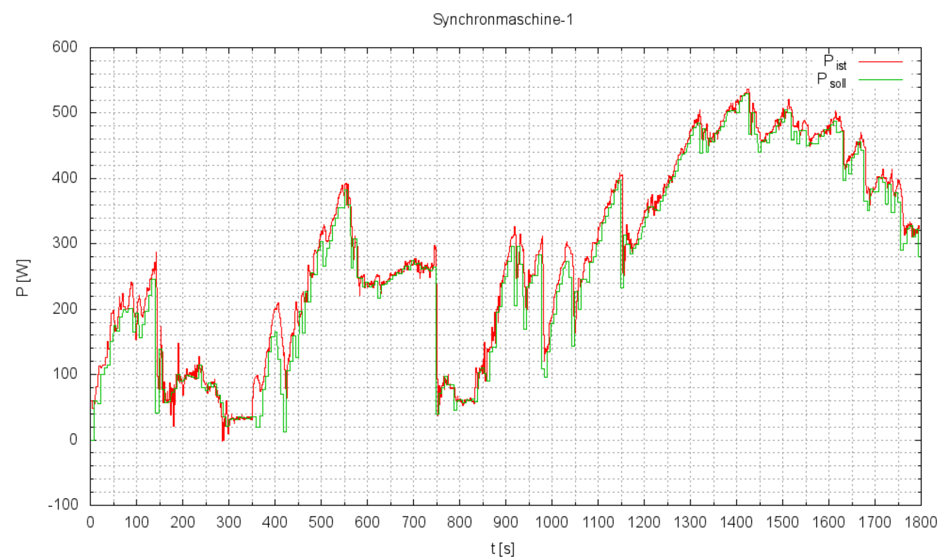


Abbildung 4.4: Synchronmaschine 1 geregelt

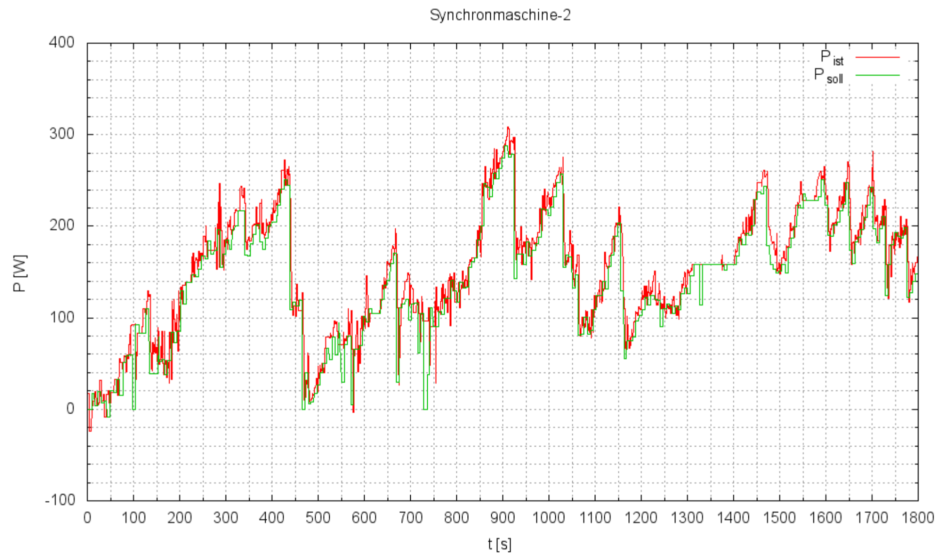


Abbildung 4.5: Synchronmaschine 2 geregelt

Abb. 4.6 addiert die Wirkleistung aller Erzeuger und Verbraucher auf. Ebenfalls dargestellt ist die Summe der Istleistungen der nicht von der Bilanzierung gesteuerten Maschinen (Wind, Solar, Verbraucher) und den Sollwerten der gesteuerten Maschinen (Synchrongeneratoren). Da letztere über UPnP versendet werden, können die angezeigten Sollwerte den tatsächlichen Sollwerten leicht hinterherhinken. Dadurch entstehen auch die Peaks in den Sollwerten: Sie entstehen, wenn bspw. ein Verbraucher über UPnP meldet, dass er mehr konsumiert. Die Bilanzierung und der Logger empfangen sofort diese Nachricht. Der Logger notiert die zusätzlich aufgenommene Leistung, gleichzeitig erhöht die Bilanzierung den Sollwert für einen Erzeuger. Von der Erhöhung des Sollwerts erfährt der Logger aber erst, wenn der Erzeuger ihm über UPnP den neuen Sollwert mitteilt. In der Zwischenzeit ist die Leistungsbilanz am Logger nicht ausgeglichen, und es entsteht ein Peak, der physikalisch gar nicht da ist. Somit können die Peaks im Diagramm für die Auswertung außer Acht gelassen werden.

4.3.2 Versuch B: ungeregelt

Auch hier wird die Leistung aller Erzeuger und Verbraucher addiert, dabei werden die Synchronmaschinen jedoch nicht berücksichtigt, da sie sowieso nur konstante Leistung produzieren. Dadurch wird deutlich mehr Leistung verbraucht als erzeugt und die Kurve liegt weit im negativen Bereich. Um sie dennoch mit den Werten aus Versuch A vergleichbar zu machen, denke man sich einen zusätzlichen Erzeuger, der zwar eine konstante Leistung erzeugt, aber „zufällig“ genau den Mittelwert der verbrauchten Leistung trifft. Dadurch wird der Mittelwert der Kurve auf die x-Achse verschoben. Abb. 4.7 zeigt diese verschobene Kurve.

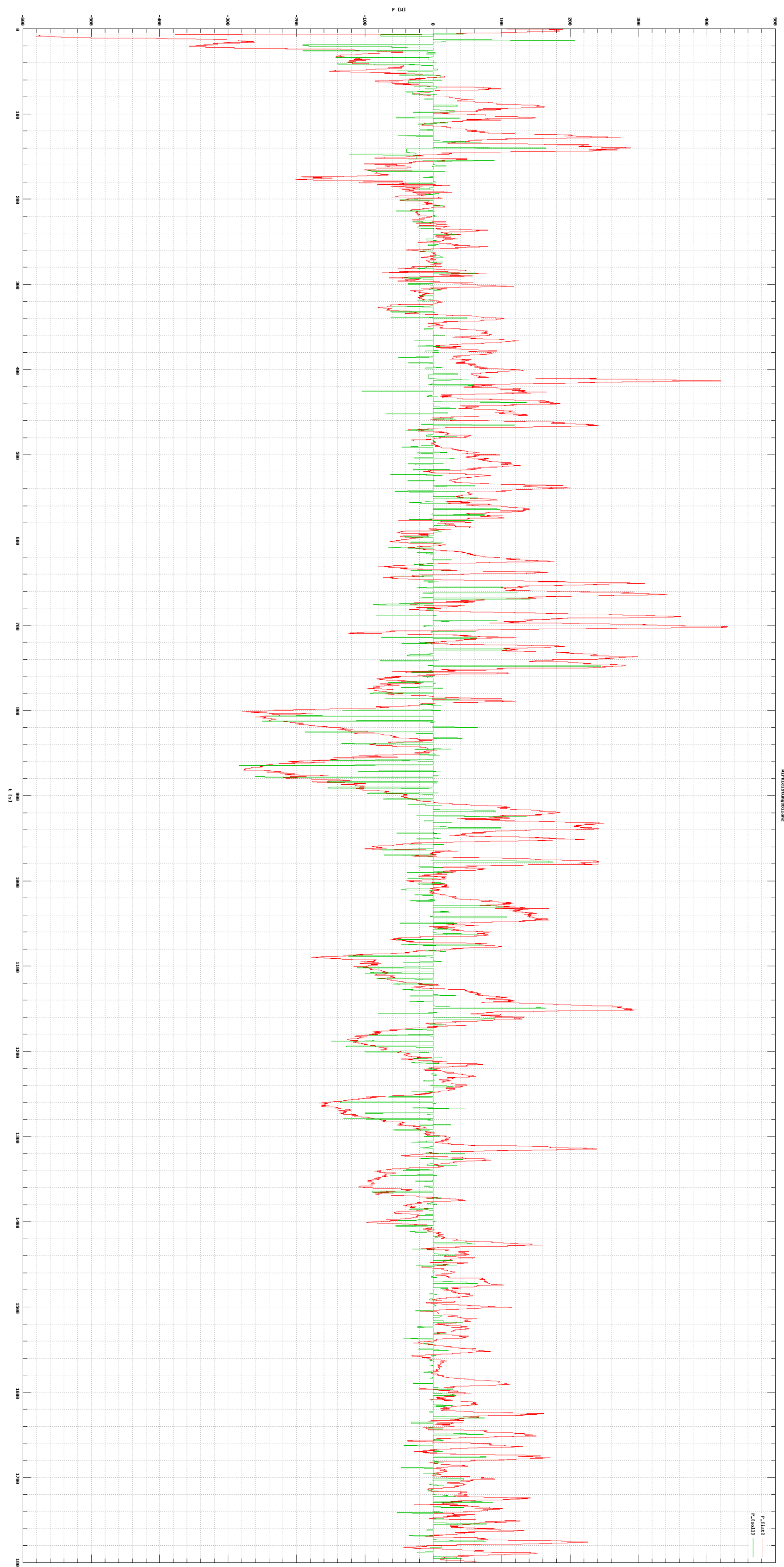


Abbildung 4.6: Leistungsbilanz geregelt

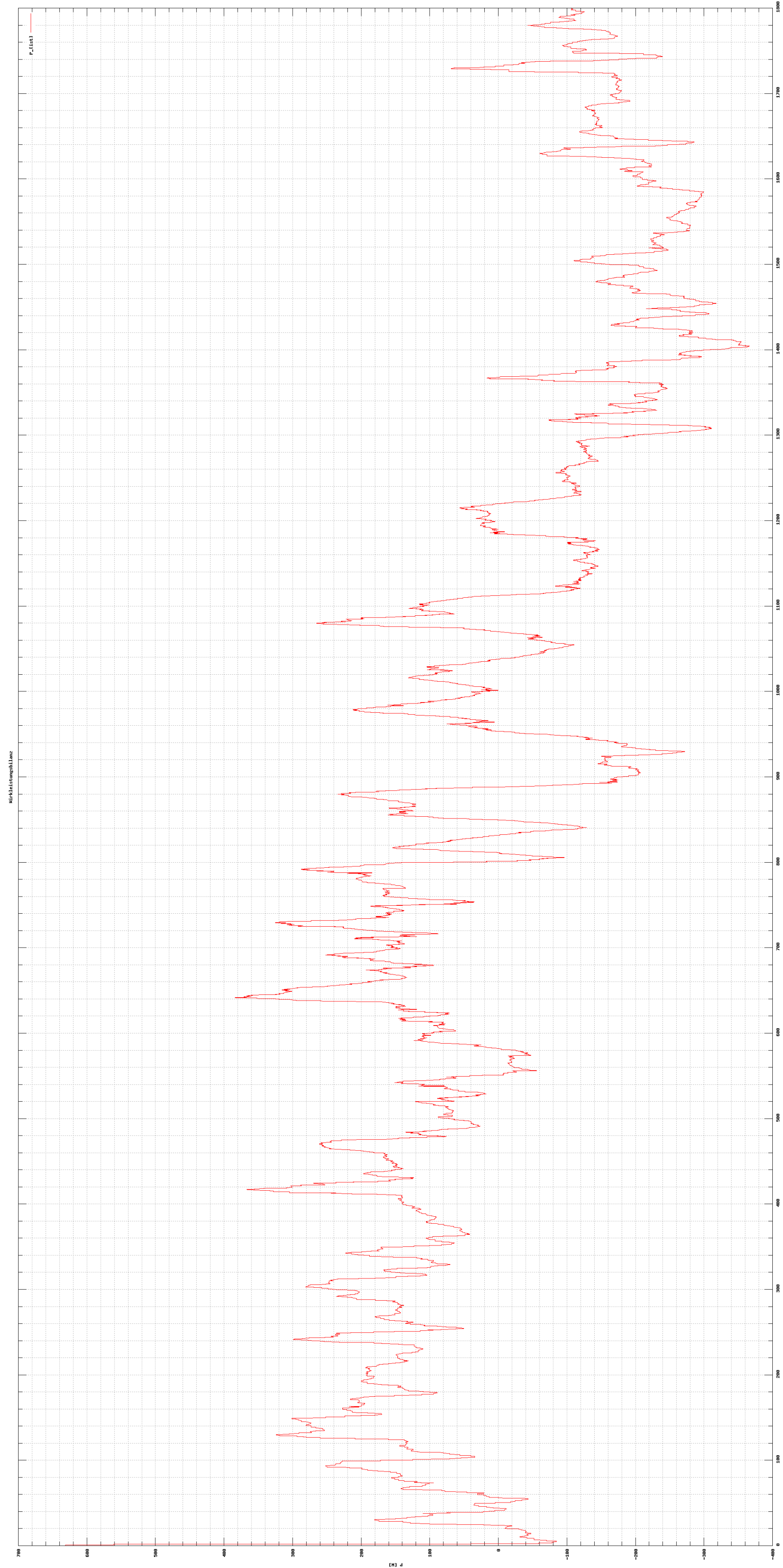


Abbildung 4.7: Leistungsbilanz ungeregelt

4.3.3 Vergleich

Zum Vergleich beider Experimente wird die Energie berechnet, die das Netz der Laborumgebung an das öffentliche Netz abgegeben hat bzw. von diesem bezogen hat. Die Leistung ergibt sich aus

$$E = \int |P| dt \tag{4.1}$$

Das Integral wird numerisch aus den Messwerten berechnet. Es ergibt sich für Versuch A (geregelt):

$$E_{ger} = 130170 \text{ Ws}$$

und für Versuch B (ungeregelt):

$$E_{ung} = 264760 \text{ Ws}$$

Somit fließt im geregelten Fall nur etwa halb so viel Energie durch den Netzzugangsknoten wie im ungeregelten Fall. Dass das Ergebnis nicht besser ist, lässt sich im Wesentlichen auf die überaus schlechte Qualität der Messwerte zurückführen. Direkt aus den verrauschten Messwerten folgt natürlich, dass die Wirkleistungsbilanzierung fehlerhafte Messwerte erhält und somit unter Umständen auf falschen Werten rechnet. Aber auch die Regelung der Maschinen wird von schlechten Messwerten stark negativ beeinflusst.

Kapitel 5

Fazit und Ausblick

Zurückblickend auf die Arbeit der Projektgruppe sind viele Probleme zu erkennen, die gelöst oder umgangen wurden. Für manche Hindernisse wurden jedoch nur provisorische Lösungen gefunden, die zu Verbesserungen und Weiterentwicklungen motivieren.

Am Ende wurden Ergebnisse erzielt, die zeigen, dass der Weg in die richtige Richtung führt und das geplante Konzept umsetzbar ist. Die Ergebnisse zeigen auch, dass genügend Potenzial bleibt, die gewonnenen Erkenntnisse und Ergebnisse weiterzuentwickeln und zukünftige Arbeiten darauf aufzubauen.

So konnte die Maschinensteuerung beispielsweise softwareseitig recht schnell so weit entwickelt werden, dass alle Maschinen bezüglich aller relevanter Stellgrößen steuerbar und teilweise auch regelbar waren. Auch können aktuelle Betriebsgrößen abgefragt werden. Hardwareseitig behindern aber qualtitativ nicht ausreichende Messgeräte, falsche Spezifikationen der Hersteller und ungenaue Antriebe der drehenden Maschinen die Präzision der Regelbarkeit. Die Messwerte schwanken um bis zu 20% der Nenngrößen.

Auch ist fraglich, ob die Größe der Laborumgebung ausreichend für Experimente ist, die über eine Machbarkeitsstudie hinausgehen. Der Dezent-Algorithmus setzt für seine Verhandlungen wenigstens mehrere Dutzend hierarchisch organisierter Erzeuger und Verbraucher voraus. Die Laborumgebung bietet jedoch nur fünf Erzeuger und einen Verbraucher. Natürlich könnten virtuelle Netzknoten erzeugt werden, deren Leistungssumme dann auf ein physikalisches Gerät abgebildet wird. Dann aber müsste sich eine weitergehende Arbeit damit befassen, wie trotzdem die Eigenschaften der Leitungen, die die Netzknoten in der Realität miteinander verbinden, berücksichtigt werden können.

Bezüglich des Dezent-Algorithmus stellt sich sich Frage, inwieweit reale Konsumenten regelbar sein können. In dieser Projektgruppe wurde vorausgesetzt, dass sie wahlweise überhaupt nicht regelbar oder vollkommen regelbar sind. In ersterem Falle sind die produzierenden Netzknoten, wie in der klassischen Netzregelung, die einzigen Knoten, die das Netz kontrolliert beeinflussen können. Im zweiten Fall wird vorausgesetzt, dass die Verbraucher komplett abschaltbar sind bzw. beliebig viel Leistung aufnehmen können. Diese Annahme ist für eine Machbarkeitsstudie sicherlich ausreichend. In Zukunft müssten Konsumenten aber in der Lage sein, eine Wunschleistung anzugeben, bzw. eine minimale Leistung, die sie auf jeden Fall benötigen. Evtl. ist dies jetzt schon möglich über den Dezent-Verhandlungsalgorithmus. Dies muss jedoch in weiteren Experimenten belegt werden.

Für den Verhandlungsalgorithmus wird eigentlich Echtzeitkommunikation benötigt, da sehr viel Information in einem fest definierten Zeitintervall ausgetauscht werden muss. Die Projektgruppe benutzt zur Zeit allerdings UPnP über ein TCP-Netzwerk. Diese Kombination ist alles andere als echtzeitfähig - UPnP bringt durch das XML-Format eine große Menge Overhead mit sich, und TCP-Verbindungen sind schon vom Design her nicht echtzeitfähig. Es müssen Experimente angestellt werden, die untersuchen, ob die Kommunikationszeiten bis zu einem gewissen Informationsaufkommen bei ausreichender Bandbreite ausreichend sind. Dabei ist zu beachten, dass die Kommunikation unter Umständen nicht in einem lokalen Netzwerk, sondern regional oder überregional stattfinden muss.

Abgesehen von den Probleme bezüglich der Hardware und der Kommunikation, in der sicherlich noch Erweiterungen einer intensiveren Arbeit gebraucht werden, ist das Ziel dieses Projekts in weitem Sinne in die richtige Richtung gestoßen. Durch die Verkürzung der Regelungsintervalle, die über die Bilanzierungsaufgaben im Netz in einem enorm kürzeren Zeitraum vollbracht werden können, ist nicht nur deutlich geworden, dass der Einsatz regenerativer Energien viel effizienter genutzt werden kann, sondern auch, dass durch die verkürzte Regelungszeit enorme Mengen an Energie allein durch die bessere Anpassung von produzierter und verbrauchter Energie gespart werden können. Somit wurde durch die Arbeit dieses Projekts der ökologische sowie auch ökonomische Sinn und Zweck des Dezent-Konzepts vertieft und dient sicherlich auch weiteren Investitionen als Motivation.

Die vielleicht wichtigste Errungenschaft der Projektgruppe ist, wenn auch durch Materialprobleme etwas verschleiert, der Beweis, dass sich die entwickelten Algorithmen zur Netzregelung eignen. Somit ist durch die bisherigen Entwicklungen ein Fundament für viele weitere Arbeiten und Entwicklungen gelegt. Auch ist der Grundstein für eine funktionierende Experimentierumgebung gelegt.

Kapitel 6

Literaturverzeichnis

[1] Wago -I/O-System,*WAGO-I/O-Pro 32 Handbuch*, Technical Report Version 2.0.0.

[2] Wago -I/O-System 750,*WAGO-I/O-System, Modulares I/O-System, ETHERNET TCP/IP Handbuch*, Technische Beschreibung, Installation und Projektierung Version 2.2.1.

[3] http://www.user.fh-stralsund.de/ emasch/1024x768/Dokumentenframe/Kompendium/Antriebstechnik/Zusatzinfo_Syn

[4] Dipl.- Ing. Hickmann *Versuch EET4 Asynchronmaschine* www.fh-nordhausen.de/uploads/media/EET4_Asynchronmaschine.pdf

[5] Edmund Handschin, *Netz- und Energiemanagement I*, Foliensatz Kapitel 5, Universität Dortmund 2005

[6] Edmund Handschin, *Elektrische Energieübertragungssysteme*, 2. Auflage, Dr. Alfred Hüthig Verlag, Heidelberg 1987

[7] UPnP-Forum, *Upnp device architecture 1.0* www.upnp.org, 2003.

[8] MicrosoftCorporation, *Understanding universal plug and play.* www.upnp.org, 2000.

[9] Satoshi Konno *CyberGarage UPnP API* http://www.cybergarage.org/net/upnp/java/index.html

[10] Martin Kaiser, *Proseminar ambient intelligence - meeting the computational needs of intelligent environments: The metaglue system.* http://w5.cs.uni-sb.de/ stahl/proseminarami/ web/themen/index.html, 2005.

[11] Michael Kleemann, *Aufbau einer Steuerung und Visualisierung für eine Netzsimulationsanlage*, Studienarbeit, Universität Dortmund 2006

[12] 3S-Smart Software Solutions GmbH, *Handbuch für SPS Programmierung mit CoDeSys 2.3*, Kempten 2005

[13] Annotations open-source library *http://www.fusionsoft-online.com/annotation.php*

[14] Wikipedia*http://de.wikipedia.org/wiki/SOAP*

[15] Wikipedia *http://de.wikipedia.org/wiki/SSDP*

[16] Wikipedia *http://de.wikipedia.org/wiki/Wechselrichter*

[17] Wikipedia *http://de.wikipedia.org/wiki/Photovoltaikanlage*

[18] Camille Bauer AG, *Sineax DME4 Schnittstellen Definition*, CH-5610 Wohlen/Switzerland 02.1998

[19] Grundlagen der Elektrotechnik IV, Prof. Dr.-Ing. D. Peier, Manuskript von M. Hebbel und A. Örtel, Aufl. 1998

[20] JFreeChart *http://www.jfree.org/jfreechart*