CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Masterstudiengang Informatik

Masterarbeit

Adaptive State Estimation in Odysseus

vorgelegt von

Jan Sören Schwarz

Gutachter:

Dr. Marco Grawunder
Jun.-Prof. Dr. Sebastian Lehnhoff

Oldenburg, 9. Februar 2015

Zusammenfassung

Die Energiewende stellt das Stromnetz vor immer größere Herausforderungen. Insbesondere die voranschreitende Verbreitung von verteilten Energieerzeugern erschwert die Bestimmung und Überwachung des genauen Netzzustands. Diese sind jedoch für einen möglichst störungsfreien Betrieb zwingend nötig und erfordern somit neue Ansätze. Ein neues Verfahren, welches die herkömmlichen Ansätze der State Estimation und Leistungsflussrechnung miteinander kombiniert, ist die *Adaptive State Estimation* (ASE). Sie ermöglicht im Vergleich zu den herkömmlichen Methoden eine höhere Flexibilität in Bezug auf die benötigten Messdaten, da sie auch mit unter- und überbestimmten Netzen umgehen kann. Um das Stromnetz überwachen zu können, ist die Berechnung des Zustands nahezu in Echtzeit nötig, wobei kontinuierlich sehr große Datenmengen zu verarbeiten sind. Für derartige Anwendungsfälle bietet sich die Verwendung darauf spezialisierter Datenstrommanagementsysteme (DSMS) an.

So wurde im Rahmen dieser Arbeit der ASE-Algorithmus in Form eines neuen Operators in das DSMS Odysseus integriert. Dadurch stehen zum einen die spezialisierten Möglichkeiten eines DSMS zur Verarbeitung großer kontinuierlicher Datenmengen zur Verfügung und zum anderen ist eine flexible und austauschbare Konfiguration der ASE-Berechnung möglich. Die Ergebnisse des ASE-Operators lassen sich auch in Odysseus visualisieren, sodass der Zustand des überwachten Netzes übersichtlich dargestellt ist und gefährliche Netzzustände leicht erkannt werden können. Außerdem lassen sich dadurch auch die Möglichkeiten des ASE-Algorithmus gut evaluiert.

Als Datenquelle für den Operator fand die Smart Grid Simulationsumgebung mosaik Verwendung. Dazu wurden zwei Varianten der Anbindung beider Systeme umgesetzt, welche die Verarbeitung der Daten aus mosaik erstmalig direkt während der Simulation in Odysseus ermöglichen. Aufgrund ihrer sehr unterschiedlichen Vor- und Nachteile haben beide Ansätze für unterschiedliche Szenarien ihre Stärken. Die erste Variante setzt mit Hilfe von ZeroMQ eine eher datenstrombasierte Kommunikation um, während die zweite einen mosaik-Simulator direkt in Odysseus realisiert und eine deutlich engere Kopplung der beiden Systeme mit sich bringt. Beide Anbindungen sind dabei so allgemein gehalten, dass sie zum einen auch problemlos für andere Anwendungsfälle weiterverwendet und zum anderen auch als Grundlage für die ASE-Berechnungen durch reale Sensordaten eines Energienetzes ersetzt werden können.

Inhalt

1 Einleitung

Die Änderungen am Energiesystem, welche die Energiewende mit sich bringt, stellen die Stromnetze vor große Herausforderungen. Das Netz ist historisch gewachsen und darauf ausgelegt, dass große Kraftwerke Strom erzeugen und über das Netz an viele Verbraucher liefern. Der größere Einsatz von verteilten Energieerzeugern, wie z.B. Blockheizkraftwerken, Photovoltaik- oder Windkraftanlagen, verändert diese Struktur, da bisher nur als Verbraucher eingebundene Haushalte plötzlich zeitweise auch zu Erzeugern werden können. Für die zuverlässige Bereitstellung von Energie ist es, trotz dieser Herausforderungen, zwingend notwendig die Netze ausreichend zu dimensionieren und ihren Zustand zu überwachen, um Überlastung zu vermeiden.

Ein wichtiges Werkzeug für die Dimensionierung und Überwachung der Stromnetze ist die Leistungsflussrechnung, die Anwendung findet, um den Fluss von Wirk- und Blindleistung im Netz zu ermitteln [Sch12, S. 777ff.]. Diese beiden Werte sind wichtig, da sie die über die Zeit übertragene Energie und somit die Auslastung der Leitungen darstellen. Die Wirkleistung ist dabei die Leistung, welche tatsächlich von den Verbrauchern umgesetzt werden kann, während die Blindleistung durch induktive und kapazitive Effekte des Netzes hervorgerufen wird und nicht genutzt werden kann.

Das Problem der Berechnung des Leistungsflusses ist nichtlinear und somit müssen iterative Verfahren zur Berechnung verwendet werden. Die verbreitetste Methode ist hierbei das *Newton-Raphson-Verfahren* (NR-Verfahren), welches bei vollständig bestimmten Messdaten für das Netz verwendet werden kann [Sch12, S. 794], [TH67]. Vor dem Durchführen einer Leistungsflussrechnung wird mit den Messdaten aus dem Netz normalerweise eine sogenannten *State Estimation* (*Zustandsbestimmung*) durchgeführt [Sch12, S. 767f.], [SW70]. Bei dieser wird anhand der vorhandenen Mess- und Schätzdaten der aktuelle Zustand des Netzes abgeschätzt. Zum Durchführen der herkömmlichen State Estimation muss das Netz jedoch überbestimmt sein – es müssen also mehr Mess- und Schätzdaten vorhanden sein, als zu berechnende Knoten des Netzes.

Die Verbreitung von Sensoren im Netz ist jedoch noch nicht so groß, dass jedes zu berechnende Netz ausreichend gut bestimmt wäre. Daher müssen die Berechnungen häufig auf kleineren Teilnetzen durchgeführt werden, wodurch es zum Verlust von Informationen kommt. Der in Kooperation der Universitäten Dortmund, Oldenburg und Queensland entwickelte Ansatz der *Adaptive State Estimation* (ASE) verbindet State Estimation und NR-Verfahren, um sowohl mit vollständig bestimmten, überals auch unterbestimmten Netzen umgehen zu können. Das Ergebnis wurde im ASE-Algorithmus umgesetzt [KL12, KL].

1.1 Ziele der Arbeit

Bei den beschriebenen Netzberechnungsverfahren kommt es kontinuierlich zu großen Datenmengen, auf denen möglichst in Echtzeit die Berechnungen durchgeführt werden sollen. Für derartige Szenarien bieten sich Datenstrommanagementsysteme (DSMS) an, da diese auf Operationen und Berechnungen in großen kontinuierlichen Datenmengen spezialisiert sind. Durch eine Integration des ASE-Algorithmus in ein DSMS kann somit der Zustand des Netzes über die Zeit überwacht werden. Zudem ist es möglich die Überwachung durch bessere Konfigurierbarkeit schnell an neue Gegebenheiten anzupassen. So können z. B. verschiedene kontinuierliche Anfragen erstellt werden, welche unterschiedliche Werte berechnen. Im Rahmen dieser Masterarbeit wird der ASE-Algorithmus in das an der Universität Oldenburg entwickelte DSMS-Framework Odysseus eingebunden.

Odysseus ist für die flexible Verarbeitungen von großen Datenmengen in Form von Datenströmen entwickelt worden und kann Daten in vielen verschiedenen Daten- und Protokollformen wie z.B. CSV, JSON, BSON, XML, HTTP, ZeroMQ, RabbitMQ, UDP und TCP einlesen und anschließend mit verschiedensten Operatoren verarbeiten. Zudem ist Odysseus modular aufgebaut, um möglichst einfach erweiterbar zu sein [AGG+12].

Zunächst soll der bereits existierende ASE-Algorithmus in Odysseus integriert werden. Dazu ist ein Operator in Bezug auf seine Konfigurierbarkeit und die Integration in Odysseus zu konzipieren. Zudem soll eine Visualisierung der Ergebnisse umgesetzt werden, welche das überwachte Netz übersichtlich darstellt und die Resultate des Operators anzeigt.

Als Grundlage für die Berechnungen des ASE-Algorithmus finden Daten aus der Smart Grid Simulationsumgebung mosaik Verwendung. Mosaik wurde am OFFIS Institut für Informatik in Oldenburg entwickelt und ermöglicht es große Smart Grid Szenarien aus verschiedensten Simulatoren und Simulationsmodellen zu erstellen und Simulationen auf ihnen durchzuführen [Sch14]. Dadurch wird es ermöglicht, bereits aus verschiedenen Domänen vorhandene Modelle zu verwenden, ohne aufwendige Reimplementierungen vornehmen zu müssen. Anhand der simulierten Szenarien können dann Tests mit Steuerungsalgorithmen, wie z.B. dem ASE-Algorithmus, durchgeführt werden.

Der Erfolg der Umsetzung wird durch Testen in verschiedenen Szenarien gezeigt. Dazu werden die in Odysseus berechneten Daten mit herkömmlich errechneten Daten in Bezug auf Korrektheit und Genauigkeit verglichen. Die herkömmlichen Daten stammen hierbei von mosaik, da aus der Simulation der genaue Zustand des Netzes bekannt ist. Zudem wird die Geschwindigkeit und Latenz der Berechnung betrachtet und die Möglichkeiten der Kooperation von Odysseus und mosaik genauer untersucht.

1.2 Aufbau der Arbeit

Diese Arbeit ist in insgesamt sieben Kapitel gegliedert. Das erste Kapitel beinhaltet dabei die Einleitung, die Ziele und den Aufbau der Arbeit. In Kapitel 2 werden zunächst die theoretischen Grundlagen erläutert. Angefangen bei der Modellierung von Leitungen und Netzen, werden Leistungsflussrechnung, State Estimation und der darauf aufbauende Adaptive State Estimation (ASE) Algorithmus genauer erklärt und das DSMS Odysseus sowie die Simulationsumgebung mosaik vorgestellt. In Kapitel 3 wird aufbauend auf den theoretischen Ausführungen der Entwurf der Umsetzung der ASE-Berechnung in Odysseus beschrieben. An erster Stelle stehen dabei die Anforderungen an die Implementierung. Daraufhin werden die Anbindung von mosaik an Odysseus, die Umsetzung des ASE-Operators und die darauf aufbauende Visualisierung in Odysseus diskutiert. Die Implementierung wird in Kapitel 4 erläutert. Dabei wird die Umsetzung des Entwurfs vorgestellt, die sich wiederum in die Anbindung von mosaik und Odysseus und die Visualisierung unterteilt. Korrektheit, Geschwindigkeit und Möglichkeiten der Implementierung werden in Kapitel 5 genauer evaluiert. Es werden darin zunächst die betrachteten Szenarien beschrieben. Anschließend geht es um die konkreten Evaluationsfragen für die Anbindung von mosaik an Odysseus, den ASE-Operator und die ASE-Berechnung im Allgemeinen. In Kapitel 6 werden die Ergebnisse der Arbeit noch einmal zusammengefasst und ein Ausblick auf mögliche weiterführende Entwicklungen gegeben.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für die weiteren Ausführungen vorgestellt. Zu Beginn erfolgt als Basis für alle weiteren Ausführungen die Beschreibung einiger mathematischer Grundlagen, Definitionen und Notationen. Danach wird auf die Aspekte der Energiedomäne dieser Arbeit eingegangen, indem die Netzberechnung und der ASE-Algorithmus eingeführt wird. Daraufhin wird das aus der Domäne der Informationssysteme stammende Odysseus vorgestellt, in welchem der zuvor beschriebene ASE-Algorithmus integriert wurde. Abschließend wird mosaik erläutert, welches die Datenquelle für die Berechnungen in Odysseus darstellt.

2.1 Mathematische Grundlagen, Definitionen und Notationen

Da gerade die Notation für Matrizen in der Literatur oft sehr unterschiedlich ist, soll an dieser Stelle eine Übersicht über die in dieser Arbeit verwendeten Notationen gegeben werden. Zudem werden einige grundlegende Begriffe der linearen Algebra kurz erläutert, die an späterer Stelle Anwendung finden. Für genauere Erklärungen sei auf [KB13] und [CD06] verwiesen.

Komplexwertiger Vektor ($\underline{A}$): Ein Vektor aus komplexen Zahlen: $\underline{A} = [a_1, \ldots, a_n]$ mit $a_i = (b_i + i \cdot c_i)$ Alternative Darstellung aus zwei reellen Zahlen: $\underline{A} = [b_1\ c_1, \ldots, b_n\ c_n]$

Transponierte Matrix (A^T): Sei $A = (a_{i,j}) \in \mathbb{R}^{(m,n)}$.
Die *transponierte* Matrix $A^T \in \mathbb{R}^{(n,m)}$ ist somit definiert durch $A^T = (b_{i,j})$ und $b_{i,j} := a_{j,i}, i = 1, \ldots, n, j = 1, \ldots, m$ [KB13, S. 64].

Inverse Matrix (A^{-1}): Die Matrix $A \in \mathbb{R}^{n,n}$ ist die inverse Matrix zur Matrix $B \in \mathbb{R}^{n,n}$, wenn gilt $AB = \mathbb{1}_n$ und $BA = \mathbb{1}_n$. $\mathbb{1}$ ist hierbei die Einheitsmatrix deren Elemente auf der Hauptdiagonale alle 1 und alle sonstigen Elemente 0 sind [KB13, S. 199].

Pseudoinverse ($A^\dagger$): Die Pseudoinverse ist eine verallgemeinerte Inverse, die selbst für nichtquadratische und singuläre Matrizen existiert, welche nicht direkt invertierbar sind. Sie entspricht dabei der Inversen $\left(A^\dagger = A^{-1}\right)$, wenn A quadratisch und nichtsingulär und somit invertierbar ist [Pen55], [KB13, S. 237]. Durch ihre Eigenschaften spielt sie eine wichtige Rolle für die Adaptive State Estimation.

Orthogonale Matrix: Eine orthogonale Matrix ist eine quadratische, reelle Matrix, deren Zeilen- und Spaltenvektoren paarweise orthonormal bezüglich des Standardskalarprodukts sind. Damit ist die Inverse einer orthogonalen Matrix gleichzeitig ihre Transponierte: $A^T = A^{-1}$. Die Spalten (Zeilen) bilden dabei eine Orthonormalbasis [KB13, S. 214].

Orthonomalbasis (ONB): Eine Basis heißt ONB, wenn ihre einzelnen Vektoren paarweise aufeinander senkrecht stehen – also z. B. für die Basis $B = (u, v)$ mit $u \neq v$ gilt $(u \cdot v) = 0$ – und alle Vektoren die Länge 1 besitzt [KB13, S. 118].

Singuläre Matrix: Eine singuläre Matrix ist eine quadratische Matrix, die mindestens zwei linear abhängige Spalten oder Zeilen und keine eindeutige Inverse besitzt. Die Summe jeder Zeile und Spalte ist 0 [KB13, S. 199ff.]. Diese Eigenschaft kann zum Überprüfen der Korrektheit der Daten verwendet werden.

2.2 Netzberechnung

Energienetze sind sehr komplexe Systeme und um ihre Kapazitäten und Belastungen planen zu können ist es notwendig die zugrunde liegenden Strukturen mathematisch zu modellieren. Daher werden im Folgenden die vorhandenen Netztopologien und die Modellierung von Leitungen und Netzen vorgestellt. Auf den vorgestellten Modellen kann anschließend mit der State Estimation der Zustand des Stromnetzes abgeschätzt werden und mit Hilfe der Leistungsflussrechnung die Belastungen ermittelt werden.

2.2.1 Aufbau des Energienetzes

Das Energienetz lässt sich zunächst in verschiedene Spannungsebenen, die jeweils durch Transformatoren miteinander verbunden sind, einteilen. Da bei hohen Spannungen die geringsten Verluste auf den Leitungen zu verzeichnen sind, werden für längere Strecken im Transportnetz Spannungen von 220 kV oder 380 kV verwendet. Auf dieser Ebene speisen z. B. auch große Kraftwerke ihren Strom in das Netz ein. Übertragungsnetze verwenden üblicherweise 110 kV und die Mittelspannungsverteilnetze in Deutschland 10 kV oder 20 kV. Auf der Mittelspannungsebene können z. B. Windkraftanlagen an das Netz angeschlossen werden. Am Niederspannungsnetz wiederum hängen die meisten normalen Verbraucher wie z. B. Haushalte. Hier sind Spannungen von 690 V, 400 V oder 230 V üblich [Sch12, S. 435ff.].

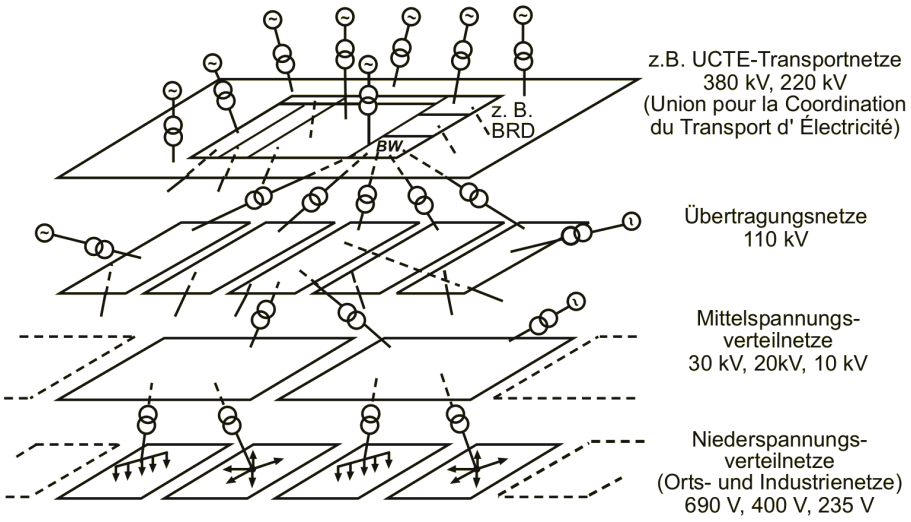
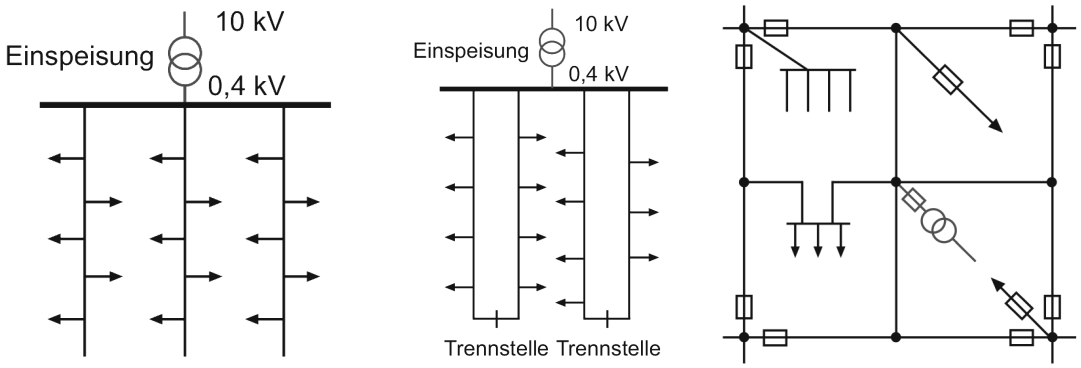


Abbildung 2.1: Spannungsebenen [Sch12, S. 436]

Bei der Verteilung von Energie kann zudem nach verschiedenen Netztopologien unterschieden werden. Eine Übersicht über die drei wichtigsten Arten Strahlen-, Ring- und Maschennetze ist in Abbildung 2.2 zu sehen.

Strahlennetze kommen hauptsächlich im Niederspannungsnetz vor und werden z. B. zur Versorgung von Häusern einer Straße verwendet. Sie sind am unkompliziertesten zu planen und zu warten, haben aber den Nachteil, dass es zu einem Spannungsabfall mit zunehmendem Abstand zur Einspeisung kommt und bei einem Kurzschluss sämtliche Verbraucher vom Netz getrennt werden. Bei



(a) Strahlennetz *(b) Ringnetz* *(c) Maschennetz*

Abbildung 2.2: Verschiedene Netztopologien

Ringnetzen können die bereits beschriebenen Strahlen jeweils an ihren Enden miteinander verbunden werden, sodass bei einem Kurzschluss die Versorgung von dem ungestörten Strahl übernommen werden kann. Sie sind am verbreitetsten in Nieder- und Mittelspannungsnetzen. Maschennetze liefern eine noch größere Ausfallsicherheit als Ringnetze, da in der vollständigen Ausführung alle Knoten mit mehreren anderen Knoten verbunden sind und so Ausfälle auf relativ kleine Bereiche im Netz beschränkt bleiben. Die Vermaschung kann jedoch aus Kostengründen auch verringert werden, wodurch jedoch das Ausfallrisiko wieder steigt [Sch12, S. 511ff.].

2.2.2 Modellierung von Leitungen

Zu den wichtigsten Elementen des Energiesystems gehören die Leitungen, über welche die verschiedenen Verbrauchen und Erzeuger miteinander verbunden sind. Mit Hilfe des in Abbildung 2.3 gezeigten Ersatzschalbildes einer einphasigen Leitung lassen sich die unterschiedlichen Arten von Leitungen, wie Erdkabel, Freileitungen und Transformatoren, abbilden. Das Ersatzschaltbild dient dabei als Vereinfachung der tatsächlichen Schaltung, welche die Berechnungen vereinfacht, aber im Allgemeinen ausreichend genau ist. Die verschiedenen Arten von Leitungen unterscheiden sich in den folgenden vier relevanten Größen [Bü13, S. 5ff.], [Leh14, S. 30], [Sch12, S. 450ff.]:

Wirkwiderstand R (serielle Resistanz): Der Wirkwiderstand ist abhängig von Material, Temperatur, Länge und Querschnitt des Leiters.

Blindwiderstand X (serielle Reaktanz): Der induktive Blindwiderstand bildet die Verluste durch Magnetfelder anderer Leiter und Selbstinduktion ab. Durch den größeren Abstand zwischen den einzelnen Leitern ist bei Freileitungen z. B. mit einem geringeren Wert zu rechnen als bei Erdkabeln.

Ableitwiderstand G (Ableit-Konduktanz): Der Ableitwiderstand stellt Verlustströme durch die Luft (Koronaentladungen), Kriechströme oder mangelhafte Isolation, dar, die vor allem bei einem Betrieb mit hoher Spannung auftreten können.

Kapazitiver Blindwiderstand B (Ableit-Suszeptanz): Der kapazitive Blindwiderstand beschreibt Lade- und Entlade-Vorgänge der Leitung, die darauf beruhen, dass die Leiter sich wie Kondensatoren verhalten.

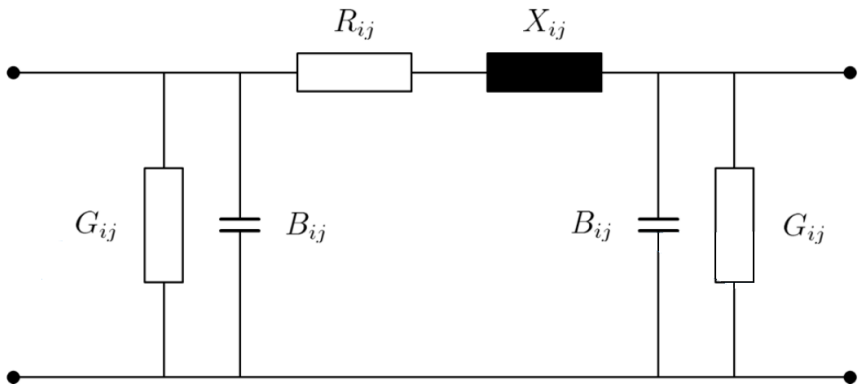


Abbildung 2.3: π äquivalentes Ersatzschaltbild eines einphasigen Leiters [Leh14, S. 30]

Das in Abbildung 2.3 gezeigt Ersatzschaltbild wird π-Glied genannt und aus mehreren von ihnen lassen sich längere Übertragungsleitungen abbilden. Die Leitungskapazität und der Ableitwiderstand werden dabei näherungsweise auf den Anfang und das Ende eines Gliedes aufgeteilt. Das Ersatzschaltbild stellt eine Idealisierung dar, da darin angenommen wird, dass die vier relevanten Größen gleichmäßig für die komplette Leitung gelten. Diese Näherungen sind jedoch für allgemeine Berechnungen ausreichend genau [Sch12, S. 464ff.].

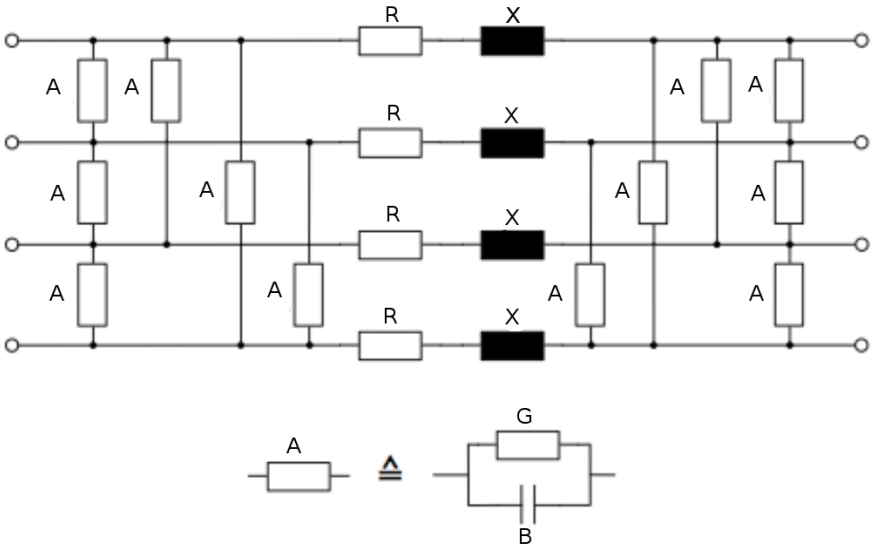


Abbildung 2.4: π äquivalentes Ersatzschaltbild einer dreiphasigen Leitung mit Neutralleiter

Der einphasige Leiter dient zunächst als Vereinfachung, da in der Realität normalerweise dreiphasige Leitungen Verwendung finden. Diese sogenannten Drehstromsysteme haben den Vorteil, dass im Verhältnis weniger Leiter gebraucht werden und mehr Leistung übertragen werden kann [MW11, S. 256ff.]. Wie in Abbildung 2.4 zu sehen, lässt sich das Ersatzschaltbild für einphasige Leiter auch auf eine dreiphasige Leitung mit Neutralleiter erweitern. Dazu werden äquivalent zum einphasigen Ersatzschaltbild in jedem Leiter Wirkwiderstand und induktiver Blindwiderstand in Reihe geschaltet

dargestellt und zwischen allen Leitern sind Ableitwiderstand und kapazitiver Blindwiderstand vorhanden [Leh14, S. 154].

2.2.3 Modellierung von Netzen

Nachdem die Modellierung von einzelnen Leitungen vorgestellt wurde, soll nun auf die Modellierung von zu einem Netz zusammengeschlossenen Leitungen, Verbraucher- sowie Erzeugereinheiten eingegangen werden. Zur Modellierung von Stromnetzen wird hauptsächlich auf Matrizen zurückgegriffen, da für die darauf aufbauenden Berechnungen häufig das Lösen von Gleichungssystemen nötig ist, was am einfachsten durch Matrizenrechnungen durchzuführen ist. Die wichtigste Matrix ist hierbei die Knotenadmittanzmatrix (im Folgenden lediglich Admittanzmatrix genannt). Diese Matrix gibt die Admittanzen zwischen allen verbundenen Knoten des Netzes an. Bei der Admittanz ($\underline{Y}$) handelt es sich um den Scheinleitwert, welcher der Kehrwert der Impedanz (Wechselstromwiderstand) ist und somit wie in Formel (2.1) zu sehen berechnet wird. Die Admittanzmatrix wird der Impedanzmatrix vorgezogen, da die Admittanz für nicht verbundenen Knoten 0 ist, während die Impedanz unendlich groß wäre. Daher ist mit der Admittanzmatrix deutlich besser zu rechnen [MW11, S. 165f.].

$$\underline{Y} = \frac{1}{\underline{Z}} = \frac{1}{R + j \cdot X} \tag{2.1}$$

Im ASE-Algorithmus wird zwar die Tor-Darstellung der Admittanzmatrix verwendet, da diese jedoch deutlich komplizierter ist, soll an dieser Stelle zum Verständnis der Funktion einer Admittanzmatrix die in [Sch12, S. 779ff.] und [Leh14, S. 32ff.] beschriebene Variante vorgestellt werden. Sie lässt sich folgerdermaßen aufstellen:

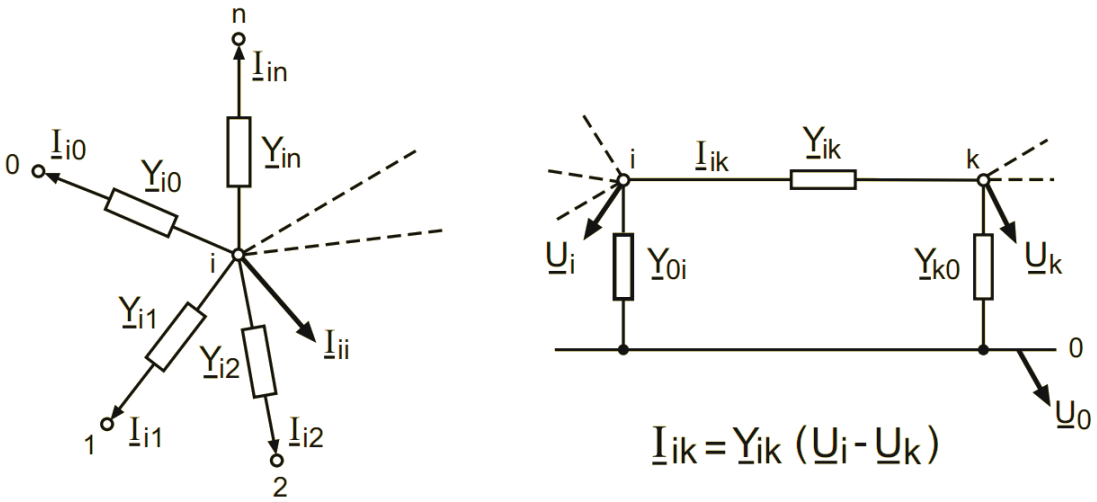


Abbildung 2.5: Strom und Admittanz zwischen Knoten [Sch12, S. 780]

1 Der Mittelpunktleiter bzw. Neutralleiter N wird mit dem Index 0 markiert.

2 Alle weiteren Knoten werden von 1 bis n durchnummeriert. Somit besitzt das Netz schließlich inklusive des Neutralleiters $n + 1$ Knoten.

3 Jedem Knoten wird eine Knotenspannung gegen einen gemeinsamen frei wählbaren Bezugspunkt zugewiesen.

4 Bei Verwendung des Erzeugerpfeilsystems werden vom Knoten abfließende Ströme als positiv gezählt – bei Verwendung des Verbraucherpfeilsystems als negativ.

5 Der komplexe Belastungsstrom oder Knotenstrom wird in einem Knoten i mit $\underline{I}_{ii}$ bezeichnet (siehe auch Abb. 2.5 links).

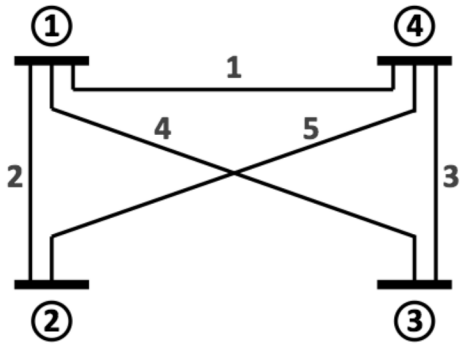
Das Ergebnis dieses Vorgangs ist in Abbildung 2.5 dargestellt. Auf der linken Seite sind die von 0 bis n nummerierten Knoten zu sehen. Der Knoten mit dem Index i ist dabei im Mittelpunkt und zwischen ihm und jedem anderen Knoten k sind Admittanz Y_{ik} und Strom I_{ik} vorhanden, mit Ausnahme von Y_{ii}. Diese wird als die negative Umlaufadmittanz definiert:

$$\underline{Y}_{ii} = -\sum_{k=0,k\neq i}^{n} \underline{Y}_{ik} \tag{2.2}$$

Somit lässt sich die Admittanzmatrix $\underline{\mathrm{Y}}$ bilden, die $n+1$ Spalten besitzt, quadratisch, symmetrisch und singulär $(det\ \underline{Y} = 0)$ ist. Dass die Matrix singulär ist, bedeutet, dass die Summe jeder Zeile und Spalte 0 ist $(\sum_{i=0}^{n} \underline{Y}_{ik} = 0, \sum_{k=0}^{n} \underline{Y}_{ik} = 0)$. Diese Eigenschaft kann verwendet werden, um die Korrektheit der Daten zu überprüfen. Die Diagonalelemente stehen für die negativen Umlaufadmittanzen und die übrigen Elemente für die Admittanzen zwischen den jeweiligen Knoten. Nicht alle Knoten sind miteinander verbunden und daher sind viele Matrixelemente mit den entsprechenden Indizes 0 [Sch12, S. 781f.], [Leh14, S. 39].

$$\underline{Y} \cdot \underline{U} = \underline{I} \ \Rightarrow\ \underline{U} = \underline{I} \cdot \underline{Y}^{-1} \tag{2.3}$$

Mit Hilfe der Admittanzmatrix lassen sich nun bei gegebenen Knotenströmen, wie in Formel (2.3) zu sehen, die Knotenspannungen berechnen. Die absolute Spannung erhält man dabei erst durch Addition der Spannung des Referenzknotens. Bei gegebenen Knotenspannungen lassen sich die Knotenströme durch Einsetzen ermitteln [Sch12, S. 786].



(a) Beispielnetz

$$Y = \begin{pmatrix} -7 & 2 & 4 & 1 \\ 2 & -7 & 0 & 5 \\ 4 & 0 & -7 & 3 \\ 1 & 5 & 3 & -9 \end{pmatrix}$$

(b) Admittanzmatrix

Abbildung 2.6: Beispiel einer Admittanzmatrix

Zur Veranschaulichung ist in Abbildung 2.6 ein Beispielnetz (2.6a) mit der dazugehörigen Admittanzmatrix (2.6b) gegeben. Die Beschriftungen im Netz stehen dabei für die Admittanzen der jeweiligen Leitung, es handelt sich hierbei lediglich um reelle Zahlen.

2.2.4 Leistungsflussrechnung

Für den möglichst problemlosen Betrieb von Stromnetzen ist es unabdingbar die Leitungen ausreichend zu skalieren, um Überlastungen zu verhindern. Als wichtigstes Verfahren wird dazu die Leistungsflussrechnung *(engl. power-flow analysis)* angewandt.

Anders als bei vielen sonstigen Problemstellungen der Netzwerktheorie, sind bei der Leistungsflussrechnung normalerweise keine konkreten Bauteile bekannt, anhand derer Daten die Leistungen bestimmt werden könnten. Dies liegt daran, dass ein Teil der elektrischen Geräte über die Leistung oder Netzfrequenz gesteuert wird. Sie reagieren bei einer Absenkung der Spannung mit einer Erhöhung der Stromstärke oder eine Änderung der Netzfrequenz führt zu einer veränderten Stromstärke, was eine genaue Berechnung sehr aufwendig und praktisch selbst für kleine Szenarien nicht durchführbar macht. Daher sind die Leistungen nur aus Lastprognosen oder Messungen bekannt [Sch12, S. 778].

Als Eingabe der Leistungsflussrechnung dienen die vorhandenen Belastungen im Netz, während als Ergebnis Knotenspannungen geliefert werden. Aus deren Differenzen kann wiederum der Spannungsabfall an den Leitungen berechnet werden. Zudem lassen sich über ihre lineare Abhängigkeit die Leitungsströme ermitteln [Sch12, S. 779]. Zur Vereinfachung der Berechnungen kann das mehrphasige System durch Entkopplung auf ein einphasiges System reduziert werden, wie in [Sch12, S. 339ff.] beschrieben.

2.2.5 Newton-Raphson-Verfahren

Das Newton-Raphson-Verfahren (NR-Verfahren) ist das verbreitetste Verfahren zur Leistungsflussrechnung [Sch12, S. 794]. Die erste ausführlichere Erläuterung des Verfahrens ist in [TH67] zu finden.

Das NR-Verfahren beruht auf dem Prinzip des Newton'schen Näherungsverfahrens [KP09, S. 233f.]. Mit diesem lassen sich Nullstellen von stetig differenzierbaren Funktionen ermitteln (siehe Abb. 2.7). Dazu wird ein beliebiger Startpunkt (x_0) gewählt und eine Tangente an die Funktionskurve im Punkt $(x_0, f(x_0))$ angelegt. Ihr Schnittpunkt mit der x-Achse ergibt sich aus Formel (2.4). Dieser Schnittpunkt wird nun als Punkt x_1 betrachtet und wiederum eine Tangente an die Funktionskurve angelegt. Auf diese Weise verfährt man solange, bis die Folge konvergiert. Somit hat man schließlich die gesuchte Nullstelle gefunden.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.4}$$

Das Problem der Leistungsflussrechnung lässt sich auch als Suche nach der Nullstelle der Funktion in Formel (2.5) beschreiben. LF steht darin für den aus einem geschätzten Spannungsvektor berechneten Leistungsvektor (siehe auch Formel (2.6)). Die unterstrichenen Werte stehen hierbei für komplexwertige Vektoren und die mit einem Stern versehenen für die komplex konjugierte Matrix.

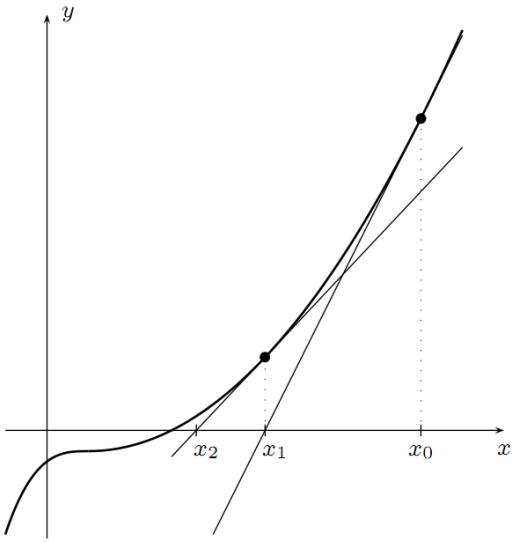


Abbildung 2.7: Prinzip des Newton-Verfahrens [KP09, S. 233])

Wenn die Differenz dieses berechnete Leistungsvektor $LF(\underline{U})$ zum gemessenen Leistungsvektor $\underline{S}$ 0 ist, dann entspricht der geschäzte Spannungsvektor $\underline{U}$ den tatsächlichen Spannungen [Leh14, S. 75].

$$f(\underline{U}) = \underline{S} - LF(\underline{U}) = 0 \tag{2.5}$$

$$LF(\underline{U}) = LF\begin{pmatrix}\underline{U}_1\\ \vdots \\ \underline{U}_n\end{pmatrix} = \begin{pmatrix}\underline{S}_1\\ \vdots \\ \underline{S}_n\end{pmatrix} = \begin{pmatrix}\underline{U}_1 \sum_{k=1}^{n} \underline{Y}^*_{1k}\underline{U}^*_k\\ \vdots \\ \underline{U}_n \sum_{k=1}^{n} \underline{Y}^*_{nk}\underline{U}^*_k\end{pmatrix} \tag{2.6}$$

$$\underline{U}^{k+1} = \underline{U}^k - \frac{f(\underline{U}^k)}{f'(\underline{U}^k)} \tag{2.7}$$

Die Formel (2.7) stellt den Iterationsschritt des Verfahrens dar. Sie lässt sich jedoch nicht ohne Weiteres lösen, da die Leistungsflussgleichungen nicht komplex differenzierbar sind. Daher wird, um die für das Verfahren benötigten Ableitungen bilden zu können, die sogenannte Jacobi-Matrix benötigt. Die Jacobi-Matrix einer differenzierbaren Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ entspricht einer $m \times n$-Matrix aller ihrer ersten partiellen Ableitungen.

Dabei muss $f : U \subset \mathbb{R}^n \to \mathbb{R}^m$ eine Funktion mit den Teilfunktionen $f := (f_1, \ldots, f_m)$ sein und für alle Teilfunktionen müssen partielle Ableitungen existieren. Zudem werden die Dimensionen im Urbildraum $\mathbb{R}^n$ als $x := (x_1, \ldots, x_n)$ bezeichnet. Die Jacobi-Matrix ist dann für einen Entwicklungspunkt $a \in U$ folgendermaßen definiert [Leh14, S. 75ff.]:

$$J_f(a) := \frac{\partial f}{\partial x}(a) := \frac{\partial(f_1, \ldots, f_m)}{\partial(x_1, \ldots, x_n)} := \begin{pmatrix}\frac{\partial f_1(a)}{\partial x_1} & \frac{\partial f_1(a)}{\partial x_2} & \cdots & \frac{\partial f_1(a)}{\partial x_n}\\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m(a)}{\partial x_1} & \frac{\partial f_m(a)}{\partial x_2} & \cdots & \frac{\partial f_m(a)}{\partial x_n}\end{pmatrix} \tag{2.8}$$

Die Jacobi-Matrix lässt sich ebenfalls für komplexwertige Funktionen $h := (h_1, \ldots, h_m) : V \subset \mathbb{C}^n \to \mathbb{C}^m$ bilden. Dafür gibt es zwei verschiedene Darstellungsformen. Erstens eine $m \times n$-Matrix mit komplexwertigen Einträgen, wie in Formel (2.9) für einen Entwicklungspunkt $z := (z_1, \ldots, z_n) \in V \subset \mathbb{C}^n$ zu sehen ist. Zweitens kann eine komplexwertige Funktion auch durch zwei reelwertige Funktionen dargestellt werden, sodass zwei Funktionen u und v existieren für die $h = u + jv$ mit $u, v : \mathbb{R}^n \to \mathbb{R}^m$ gilt. Für einen Entwicklungspunkt $z \in V$ mit $z_k := x_k + jy_k$ ist die Jacobi-Matrix dann wie in Formel (2.10) zu sehen definiert. In diesem Fall handelt es sich um eine $2m \times 2n$-Matrix mit reelwertigen Einträgen.

$$J_f^C(z) := \begin{pmatrix} \frac{\partial h_1(z)}{\partial z_1} & \frac{\partial h_1(z)}{\partial z_2} & \cdots & \frac{\partial h_1(z)}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_m(z)}{\partial z_1} & \frac{\partial h_m(z)}{\partial z_2} & \cdots & \frac{\partial h_m(z)}{\partial z_n} \end{pmatrix} \tag{2.9}$$

$$J_f^R(z) := \begin{pmatrix} \frac{\partial u_1(z)}{\partial x_1} & \cdots & \frac{\partial u_1(z)}{\partial x_n} & \frac{\partial u_1(z)}{\partial y_1} & \cdots & \frac{\partial u_1(z)}{\partial y_n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial u_m(z)}{\partial x_1} & \cdots & \frac{\partial u_m(z)}{\partial x_n} & \frac{\partial u_m(z)}{\partial y_1} & \cdots & \frac{\partial u_m(z)}{\partial y_n} \\ \frac{\partial v_1(z)}{\partial x_1} & \cdots & \frac{\partial v_1(z)}{\partial x_n} & \frac{\partial v_1(z)}{\partial y_1} & \cdots & \frac{\partial v_1(z)}{\partial y_n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial v_m(z)}{\partial x_1} & \cdots & \frac{\partial v_m(z)}{\partial x_n} & \frac{\partial v_m(z)}{\partial y_1} & \cdots & \frac{\partial v_m(z)}{\partial y_n} \end{pmatrix} \tag{2.10}$$

Mit Hilfe der Jacobi-Matrix lässt sich nun die Formel (2.7) für das NR-Verfahren mit Spannungsvektoren in Formel (2.11) spezifizieren.

$$\underline{U}^{k+1} = \underline{U}^k - \frac{f\left(\underline{U}^k\right)}{f'\left(\underline{U}^k\right)} = \underline{U}^k - \frac{\underline{S} - LF\left(\underline{U}^k\right)}{-J\left(\underline{U}^k\right)} = \underline{U}^k + \left(\underline{S} - LF\left(\underline{U}^k\right)\right) \cdot J^{-1}\left(\underline{U}^k\right) \tag{2.11}$$

Der Ablauf des NR-Verfahrens ist in Abbildung 2.8 dargestellt. Zunächst werden die Leistungen $\underline{S}^{(0)}$ an den Knoten gemessen und ein initialer Spannungsvektor $\underline{U}^{(0)}$ geschätzt. Anschließend wird die Jacobi-Matrix gebildet und invertiert. Falls das Invertieren nicht möglich ist, muss sie reduziert werden. Nach dem erfolgreichen Invertieren kann der nächste Spannungsvektor berechnet werden und darauf aufbauend auch der nächste Leistungsvektor. Nun werden dieser neu berechnete Leistungsvektor und der ursprünglich gemessene miteinander verglichen. Falls die Differenz nicht kleiner ist als ein zu Beginn festgelegtes ϵ, wird der Vorgang wiederholt. Wenn die Abbruchbedingung erfüllt ist, können abschließend die Knotenströme und -spannungen berechnet werden [Leh14, S. 83].

Das NR-Verfahren hat gegenüber anderen Verfahren der Leistungsflussrechnung den Vorteil, dass der Zeitbedarf nur quadratisch $\left(O(n^2)\right)$ und nicht exponentiell zur Knotenanzahl n zunimmt. Jedoch hat es auch den Nachteil, dass es nicht immer konvergiert. Zum einen kann es zu mathematisch möglichen Mehrfachlösungen kommen, die aber technisch nicht zulässig sind. Zum anderen nähert die Jacobi-Matrix das Verhalten der Leistungsflussgleichungen nur abhängig vom Entwicklungspunkt an. Für eine Konvergenz des Algorithmus ist es daher sehr wichtig, dass der initial gewählte Spannungsvektor bereits möglichst gut geeignet ist [Leh14, S. 84]. Damit der initiale Spannungsvektor möglichst gut gewählt werden kann, findet die State Estimation Verwendung, die im kommenden Abschnitt genauer erläutert wird.

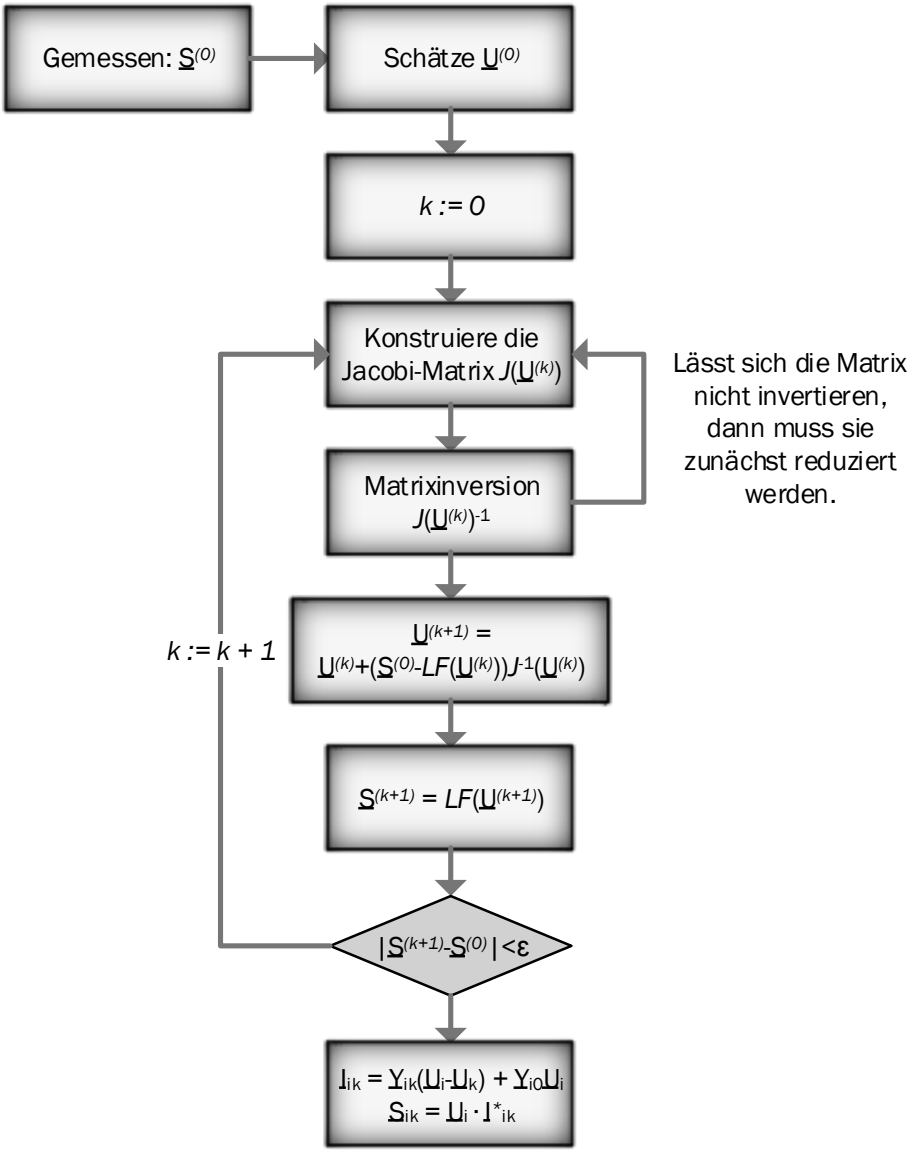


Abbildung 2.8: Algorithmus des NR-Verfahrens [Leh14, S. 83]

2.2.6 State Estimation

Für einen störungsfreien Betrieb des Netzes ist es wichtig möglichst genaue Informationen über den Netzzustand zu haben, da anhand dieser Daten weitere Diagnoseverfahren ausgeführt werden, wie z. B. die im vorangehenden Abschnitt beschriebene Leistungsflussrechnung. Die wichtigsten Daten

sind dabei die Knotenspannungen und die dazugehörigen Phasenwinkel und die Wirk- und Blindleistungen an den Netzknoten [Sch12, S. 767], [AE04, Kap. 2.6.1].

Die Grundlage für die Bestimmung des Netzzustandes sind Messwerte von Sensoren im Netz. Doch bei diesen kann es immer wieder zu Fehlern kommen. In [SH74] werden die möglichen Fehler folgenden Kategorien zugeordnet:

1 Kleine zufällige Fehler in der Sensorkommunikation.

2 Ungenauigkeiten der Systemparameter, die das Verhalten von z. B. Leitungen, Transformatoren oder Sensoren beschreiben.

3 Fehlerhafte oder fehlende Daten durch Schaltvorgänge, Messfehler und Kommunikationsfehler.

4 Fehlerhafte Netzwerkstruktur bedingt durch falsche Statusinformationen von Schaltanlagen oder Schutzeinrichtungen.

Das Ziel der State Estimation (*Netzzustandsschätzung*) ist daher fehlerhafte oder fehlende Daten zu finden und wenn möglich zu korrigieren. Um Fehler finden und beheben zu können, ist es wichtig, dass das Netz überbestimmt ist. In [AE04, Kap. 1.3] sind folgende Funktionen der State Estimation zusammengefasst:

Topologiebetrachtung: Verarbeitet Statusinformationen über das Netz und hält in Echtzeit den Netzplan des Systems vor.

Lösbarkeitsanalyse: Überprüft ob mit den vorhandenen Messwerten der Netzzustand ausreichend genau bestimmt werden kann. Teilabschnitte des Netzes werden erkannt, deren Zustand unter Umständen nicht bestimmt werden kann.

State Estimation Lösung: Liefert die optimale Schätzung des Netzzustandes anhand der vorhandenen Messwerte und der Netzstruktur. Die Lösung besteht dabei aus einem komplexwertigen Spannungsvektor.

Fehlererkennung in den Daten: Findet Fehler in den Messdaten und beseitigt sie, falls dafür genügend Informationen vorhanden sind. Für die Korrekturen kommen spezielle Algorithmen, wie z. B. Kalman-Filter, zum Einsatz, die mit stochastischen Methoden versuchen die wahrscheinlichsten Zustandswerte zu finden [Sch12, S. 767f.].

Betrachtung der Parameter und struktureller Fehler: Schätzt verschiedene Netzparameter, wie z. B. Parameter des Übertragungsnetzes oder der Transformatoren. Erkennt strukturelle Fehler in der Netzkonfiguration.

Der Fokus dieser Arbeit und des ASE-Algorithmus liegt dabei in der Kategorie State Estimation Lösung, auf deren Funktion im Folgenden genauer eingegangen wird.

Der Datenfluss der State Estimation wird in Abbildung 2.9 dargestellt. Als Eingabewerte werden zum einen die Messdaten des Netzes – hauptsächlich in Form von Knotenleistungen – und zum anderen die statische und dynamische Netzstruktur benötigt. Die statische Struktur ist dabei als grundsätzlich gleichbleibend anzusehen und erfährt höchstens durch Ausbau des Netzes Veränderung, während die dynamische Struktur sich durch kurzfristige Schaltvorgänge ändern kann. Das Ergebnis der State Estimation soll schließlich ein möglichst genauer Netzzustand sein, der in Form der Spannungen und Phasenwinkel aller Netzknoten ausgegeben wird. Außerdem wird das Netzwerkmodell geliefert [SH74].

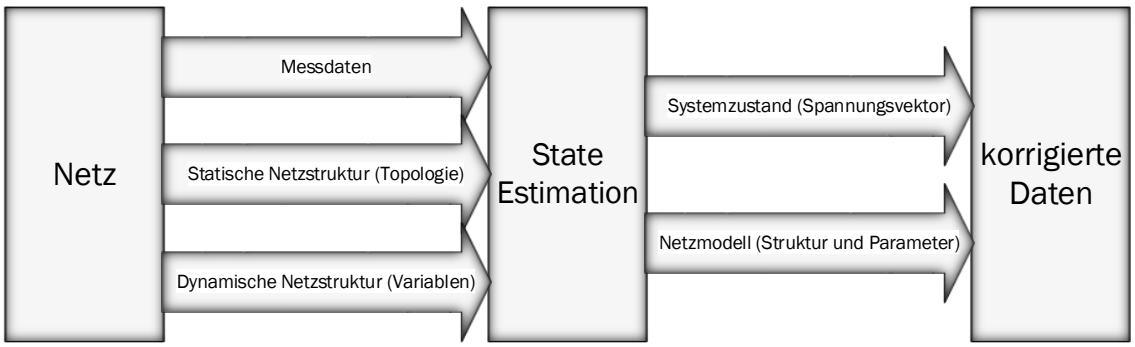


Abbildung 2.9: Datenfluss der State Estimation

Die grundlegende Methode der State Estimation, wie sie auch heutzutage noch Anwendung findet, wurde bereits 1970 von F. C. Schweppe in [SW70], [SR70] und [Sch70] veröffentlicht. Im Folgenden soll sie kurz vorgestellt werden. Aus Gründen der Einheitlichkeit wird dabei hauptsächlich die Notation aus [KL] und [AE04] verwendet. Das dabei zu lösende Problem ist die Minimierung des Fehlers in Gleichung (2.12).

$$z = \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} h_1(x_1, \ldots, x_n) \\ \vdots \\ h_m(x_1, \ldots, x_n) \end{bmatrix} + \begin{bmatrix} e_1 \\ \vdots \\ e_m \end{bmatrix} = h(x) + e \tag{2.12}$$

Hierbei steht z für den Vektor der tatsächlich gemessenen Werte und $h(x)$ für das Netzmodell. Die Werte des Netzmodells werden anhand des geschätzten Netzzustands x, welcher der komplexwertige Spannungsvektor (2.13) ist, berechnet und stellen die theoretischen Messwerte in Abhängigkeit des Netzzustandes dar. e ist die Differenz aus den tatsächlich gemessenen und den theoretischen Messwerten und somit der Fehler in der aktuellen Abschätzung des Netzzustands. Ziel ist es den Fehler e zu minimieren.

Es wird davon ausgegangen, dass einzelne Fehler e_i dabei den Erwartungswert $E(e_i) = 0$ haben und einer Normalverteilung mit der Standardabweichung σ_i folgen. Zudem wird angenommen, dass die einzelnen Fehler untereinander unabhängig sind, also eine Kovarianz $E(\sigma_i \sigma_j) = 0$ besitzen. Somit stellt ein Netzzustand x, der den Fehler e minimiert, den wahrscheinlichsten Netzzustand zu den gemessenen Werten z dar [SW70, S. 121f.], [KL].

$$x = [e_1 \; f_1, \ldots, e_n \; f_n]^T \text{ mit } x_i = (e_i + i \cdot f_i) \tag{2.13}$$

Zur Minimierung von e wird die mittlere quadratische Abweichung (engl.: *mean square error [MSE]*) verwendet. Dazu wird die Kovarianzmatrix R aufgestellt (2.14). Durch die Annahme, dass

die einzelnen Fehler unabhängig voneinander sind, entspricht die Hauptdiagonale den σ-Werten der einzelnen Fehler und alle Elemente außerhalb der Diagonale sind 0.

$$R = \begin{bmatrix} \sigma_1^2 & & 0 \\ & \ddots & \\ 0 & & \sigma_m^2 \end{bmatrix} \tag{2.14}$$

Aufgrund von $e_i = z_i - h_i(x)$ lässt sich somit der MSE wie in (2.15) oder in (2.16) als Matrizenoperation beschreiben.

$$MSE(x) = \sum_{i=1}^{m} \frac{(z_i - h_i(x))^2}{R_{ii}} \tag{2.15}$$

$$MSE(x) = [z - h(x)]^T R^{-1} [z - h(x)] \tag{2.16}$$

Um das Minimum des MSE zu bestimmen, ist ein x zu finden, dessen erste Ableitung 0 entspricht. Die Ableitung, also die Abhängigkeit des Netzmodells $h(x)$ zu Änderungen des Netzzustands x, wird als Sensitivität $H(x)$ bezeichnet und als Jacobi-Matrix, die in Formel (2.17) zu sehen ist, abgebildet.

$$H(x) = \left[\frac{\partial h(x)}{\partial x}\right] = \begin{bmatrix} \frac{\partial h_1(v)}{\partial e_1} & \frac{\partial h_1(v)}{\partial f_1} & \cdots & \frac{\partial h_1(v)}{\partial e_n} & \frac{\partial h_1(v)}{\partial f_n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial h_m(v)}{\partial e_1} & \frac{\partial h_m(v)}{\partial f_1} & \cdots & \frac{\partial h_m(v)}{\partial e_n} & \frac{\partial h_m(v)}{\partial f_n} \end{bmatrix} \tag{2.17}$$

Mit Hilfe dieser Definition lässt sich das Problem der Suche der kleinsten mittleren quadratischen Abweichung (*least mean square error [LMSE]*) über die Herleitungen in (2.18) und (2.19) wie in (2.20) zu sehen definiert. Hierbei gilt, dass der Faktor 2 bei der Verwendung häufig auch weggelassen wird.

$$g(x) = \frac{\partial MSE(x)}{\partial x} = \sum_{i=1}^{m} \frac{(2h_i(x) - 2z_i)\, H(x)}{R_{ii}} \tag{2.18}$$

$$g(x) = -2 \cdot \underbrace{\begin{bmatrix} \frac{\partial h_1(x)}{\partial x_1} & \cdots & \frac{\partial h_1(x)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_m(x)}{\partial x_1} & \cdots & \frac{\partial h_m(x)}{\partial x_n} \end{bmatrix}}_{H^T(x)} \cdot \underbrace{\begin{bmatrix} \frac{1}{\sigma_1^2} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{\sigma_m^2} \end{bmatrix}}_{R^{-1}} \cdot \underbrace{\left[\begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} - \begin{bmatrix} h_1(x) \\ \vdots \\ h_m(x) \end{bmatrix} \right]}_{e(x)} \tag{2.19}$$

$$g(x) = -2H^T(x)R^{-1}\left[z - h(x)\right] = 0 \tag{2.20}$$

Wie in [AE04] beschrieben, wird in der traditionellen State Estimation zum Lösen dieses Nullstellenproblems das Gauss-Newton-Verfahren verwendet. Dazu wird $G(x)$ als Ableitung von $g(x)$ verwendet, wie in (2.21) beschrieben.

$$G(x) = \frac{\partial g(x)}{\partial x} = H^T(x)R^{-1}H(x) \tag{2.21}$$

Damit das Problem lösbar ist, ist es jedoch zwingend notwendig, dass alle Zustandsvariablen vollständig durch $h(x)$ und z bestimmt sind. Dies liegt daran, dass $G(x)$ invertierbar sein muss, wie in [SW70, S. 124] und [KL] genauer hergeleitet wird. Bei der traditionellen State Estimation wird daher bei Unterbestimmtheit das zu betrachtende Netz soweit verkleinert, dass es genau bestimmt ist, wodurch jedoch auch vorhandene Informationen verloren gehen.

2.3 Adaptive State Estimation (ASE)

Wie im vorangehenden Abschnitt beschrieben, hat die traditionelle State Estimation das Problem, dass die Gain Matrix $G(x)$ invertierbar sein muss, was nur der Fall ist, wenn alle Zustandsvariablen vollständig bestimmt sind. Bei der ASE wird daher mit Hilfe der Singulärwertzerlegung (SVD) die Pseudoinverse gebildet, sodass auch unter- und überbestimmte Netze betrachtet werden können. Im Folgenden soll zunächst in die mathematischen Grundlagen von SVD und Pseudoinversen eingeführt werden. Daraufhin wird die Funktionsweise des ASE-Algorithmus und abschließend seine Implementierung vorgestellt.

2.3.1 SVD und Pseudoinverse

Die SVD bezeichnet eine Darstellung einer Matrix als Produkt der drei speziellen Matrizen U, Σ und V (2.22) und existiert für jede Matrix $A \in \mathbb{R}^{(m,n)}$. Dabei sind $U \in \mathbb{R}^{(m,m)}$, $V \in \mathbb{R}^{(n,n)}$ orthogonal [KB13, S. 521]. U heißt auch linker singulärer Vektor und V rechter singulärer Vektor.

$$A = U\Sigma V^T \tag{2.22}$$

Σ ist eine Diagonalmatrix mit genau $r = Rang(A)$ positiven Diagonalelementen σ_i. Diese Werte σ_i sind die positiven Singulärwerte und alle sonstigen Elemente der Matrix sind Null (2.23). Wenn die einzelnen Singulärwerte absteigend sortiert sind, heißt die SVD *normiert* [KB13, S. 516ff.].

$$\Sigma = \begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix} \tag{2.23}$$

Wie in Formeln (2.24) bis (2.27) zu sehen ist, sind die ersten r rechten und linken Singulärvektoren miteinander über σ_i mit $i \leqslant r$ verbunden. Für die übrigen Vektoren besteht keine direkte Beziehung [KB13, S. 516].

$$Av_i = \sigma_i u_i \qquad \text{für } i = 1, \ldots, r \tag{2.24}$$

$$Av_i = 0 \qquad \text{für } i = (r+1), \ldots, n \tag{2.25}$$

$$u_j^T A = \sigma_j v_j^T \qquad \text{für } j = 1, \ldots, r \tag{2.26}$$

$$u_j^T A = 0 \qquad \text{für } j = (r+1), \ldots, n \tag{2.27}$$

Mit Hilfe der SVD lässt sich nun die Pseudoinverse oder auch *Moore-Penrose-Inverse* bilden, wie in Formel (2.28) zu sehen ist. Sie ist eine verallgemeinerte Inverse für nichtquadratische und singuläre Matrizen, welche nicht direkt invertierbar sind. Wie in Lemma 1.3 in [Pen55] beschrieben wird, entspricht die Pseudoinverse der Inversen $\left(A^\dagger = A^{-1}\right)$, wenn A quadratisch und nichtsingulär und somit invertierbar ist. Für die Herleitung sei auf [Pen55] und [KB13, S. 525ff.] verwiesen.

$$A^\dagger = U^T \Sigma^{-1} V \tag{2.28}$$

2.3.2 Funktionsweise der adaptiven State Esimation

Wie im vorrangegangenen Abschnitt beschrieben, ist es für die traditionelle State Estimation nur möglich zu einem Ergebnis zu kommen, wenn die gemessenen Werte z und das Messmodell $h(x)$ vollständig bestimmt sind. Die nun vorgestellte adaptive Netzzustandsbestimmung soll genau dieses Problem beheben und es ermöglichen eine State Estimation auch für unter- und überbestimmte Fälle durchführen zu können [KL12, KL]. Für genauere Informationen zu der verwendeten Notation sei auf Abschnitt 2.1 verwiesen.

Bei der adaptiven State Estimation wird nicht explizit der MSE gebildet (vgl. Formel (2.15)), sondern stattdessen der restliche Schätzfehler $z - h(x^k)$ in der Iteration k in Relation zu dem erwarteten Schätzfehler Δx^{k+1} gesetzt. Dieser erwartete Schätzfehler wird mit Hilfe der Sensitivität (2.17) ermittelt. Aufbauend auf dem Iterationsschritt des NR-Verfahrens (vgl. (2.11) und (2.29)) lässt sich somit Formel (2.30) aufstellen.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{z - h(x^k)}{H(x^k)} \tag{2.29}$$

$$H(x^k)\Delta x^{k+1} = \left[z - h(x^k)\right] \tag{2.30}$$

$H(x^k)$ entspricht dabei der bereits in Formel (2.17) dargestellten Form.

Um die Qualität der verschiedenen Messungen zu gewichten, werden die Standardabweichungen in Form der Diagonalmatrix S verwendet (2.31).

$$S = \begin{bmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_m \end{bmatrix} \tag{2.31}$$

(2.30) lässt sich mit (2.31) zu (2.32) erweitern, sodass die einzelnen Fehler mit ihrer Standardabweichung gewichtet werden.

$$S^{-1}H(x^k)\Delta x^{k+1} = S^{-1}\left[z - h(x^k)\right] \tag{2.32}$$

Formel (2.33) zeigt den Iterationsschritt der adaptiven State Estimation. Dabei wird statt der Inversen $\left[S^{-1}H(x^k)\right]^{-1}$ die Pseudoinverse $\left[S^{-1}H(x^k)\right]^{\dagger}$ von $\left[S^{-1}H(x^k)\right]$ verwendet. Dies hat den wichtigen Vorteil, dass die Pseudoinverse, anders als die Inverse, immer gebildet werden kann, wie in Abschnitt 2.3.1 gezeigt wurde.

$$\Delta x^{k+1} = \left[S^{-1}H(x^k)\right]^{\dagger} S^{-1}\left[z - h(x^k)\right] \tag{2.33}$$

Der Übersichtlichkeit halber wird im Folgenden als Abkürzung $J(x)$ verwendet (2.34).

$$J(x) = \left[S^{-1}H(x^k)\right] \tag{2.34}$$

An dieser Stelle wird nun die im vorherigen Abschnitt eingeführte SVD verwendet, um $J(x)$ in linke Singulärvektoren $U_J(x)$, rechte Singulärvektoren $V_J(x)$ und die Singulärwerte $\Sigma_J(x)$ zu zerlegen (2.35). Zur besseren Lesbarkeit soll im Folgenden auf den J-Index verzichtet werden.

$$J(x) = V_J(x) \cdot \Sigma_J(x) \cdot U_J^T(x) \tag{2.35}$$

Die Singulärwerte $\sigma_i(x)$ der Zerlegung bilden positiv und in absteigender Ordnung sortiert die Matrix $\Sigma_J(x)$, deren Rang $r(x)$ nicht größer als m oder n sein kann (vgl. Abschnitt 2.3.1). Dabei bilden die m linken Singulävektoren $v_i(x)$ eine ONB, die den kompletten Raum der Messungen aufspannen (2.37) und die n rechten Singulävektoren $u_i(x)$ eine ONB, die den kompletten Zustandsraum aufspannen (2.36)

$$V(x) = [v_1(x) \quad \dots \quad v_m(x)] \tag{2.36}$$

$$U(x) = [u_1(x) \quad \dots \quad u_n(x)] \tag{2.37}$$

Wie bereits in Abschnitt 2.3.1 beschrieben, sind nur die ersten $r(x)$ linken und rechten Singulärvektoren durch Elemente größer Null miteinander verbunden. Die ersten $r(x)$ linken Singulärvektoren $v_1(x) \dots v_{r(x)}(x)$ spannen dabei eine lineare Annäherung des Unterraums des Messraums auf, der

alle Messkombinationen enthält, die durch $h(x)$ abgebildet werden, also mit dem Zustandsraum verbunden sind und daher – nach erfolgreichem Ablauf der ASE – ohne Fehler sind.

Für unterbestimmte Fälle $(r(x) < n)$ spannen die rechten Singulärvektoren $u_{(r(x)+1)}(x) \dots u_n(x)$ die lineare Näherung des Unterraums der nicht durch $h(x)$ bestimmten Zustandskomponenten auf. Dies bedeutet, dass sich diese Zustandsvariablen ändern können, ohne dass sich dies in den Messdaten zeigen würde.

Für $(r(x) < m)$ ist das System zumindest teilweise überbestimmt, während es für $(r(x) < n)$ zumindest teilweise unterbestimmt ist. Die beiden Bedingungen können auch zusammen auftreten.

Um die mit $h(x)$ und z zu beobachtenden Zustandsvariablen zu bestimmen, wird die Pseudoinverse $J^\dagger(x)$ gebildet. Diese Pseudoinverse invertiert die vorhandenen Abhängigkeiten in $h(x)$ und ist die lineare Näherung des invertierbaren Teils des Messmodells $h(x)$, welches den Raum der tatsächlichen Messungen auf den Unterraum der Zustände, die durch die vorhanden Messungen bestimmt sind, abbildet. Mit der invertierten Matrix $\Sigma_J^{-1}(x)$ (2.38), lässt sich die Pseudoinverse, wie in Formel (2.39) zu sehen, berechnen.

$$\Sigma_J^{-1}(x) = \begin{bmatrix} \frac{1}{\sigma_{J,1}(x)} & & & & & \\ & \ddots & & & & \\ & & \frac{1}{\sigma_{J,r(x)}(x)} & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix} \tag{2.38}$$

$$J^\dagger(x) = V_J^T(x) \cdot \Sigma_J^{-1}(x) \cdot U_J(x) \tag{2.39}$$

Da nun die Pseudoinverse berechnet werden kann, ist es möglich aus (2.33) den Iterationsschritt (2.40) aufzustellen.

$$x^{k+1} = x^k + \left[J^\dagger\left(x^k\right)\right] S^{-1}\left[z + h\left(x^k\right)\right] \tag{2.40}$$

Im Fall der Kovergenz wird keine weitere Iteration durchgeführt, wenn keine weitere Verbesserung mit (2.40) möglich ist, also Formel (2.41) erfüllt wird.

$$0 = \left[J^\dagger\left(x^k\right)\right] \underbrace{S^{-1}\left[z + h\left(x^k\right)\right]}_{e_n(x)} \tag{2.41}$$

Der beschriebene Ablauf des Algorithmus ist in Abbildung 2.10 illustriert. Außer dem Spannungsvektor als Ergebnis, liefert der Algorithmus zusätzlich die Güte der geschätzten Werte in Form der Modellabdeckung. Dazu wird die sogenannte Zustandsunbestimmtheit (*State Indeterminacy*) verwendet. Diese beschreibt das Phänomen, dass in zumindest teilweise unterbestimmten Systemen der geschätzte Systemzustand x nur einer von unendlich vielen möglichen ist, die alle durch $h(x)$ auf exakt den gleichen Messvektor abgebildet werden. Der Unterraum der Systemzustände, welche auch durch $h(x)$ zu keiner Veränderung im Messraum führen, wird vom *(Right Nullspace)* von $J(x)$ genähert.

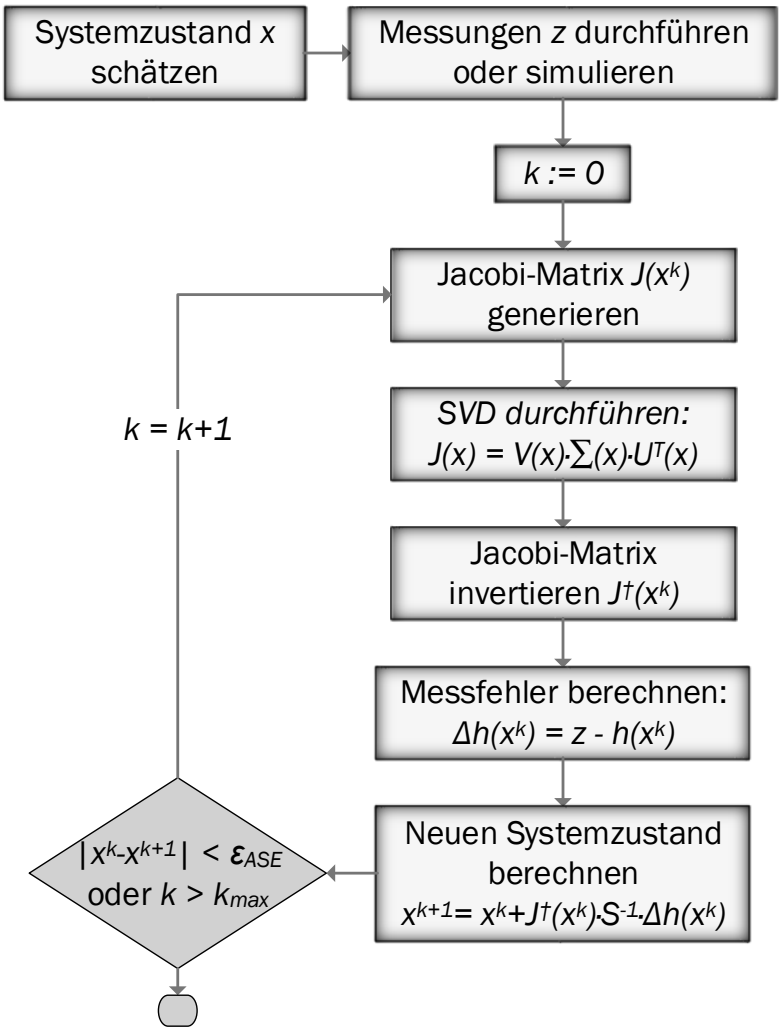


Abbildung 2.10: Ablauf des ASE-Algorithmus

Ein geschätzter Parameter erhält einen Wert von 0 für die Modellabdeckung, wenn er nicht mit dem (*right nullspace*) von $h(x)$ verbunden ist. Höhere Werte stehen für eine stärkere Sensitivität und damit einen unverlässlicher geschätzten Wert.

2.3.3 Aufbau der Implementierung

Den beschriebenen ASE-Algorithmus wurde von seinen Entwicklern auch als Implementierung in Java umgesetzt. Bisher beinhaltete er jedoch noch keine benutzerfreundliche Möglichkeit der Steuerung, sodass Szenarien im Quellcode definiert werden mussten. Im Rahmen dieser Arbeit soll durch die Integration in ein DSMS eine flexiblere Ausführung und Konfigurierbarkeit erzielt werden. Da-

für stand der gesamten Quellcodes bereit, der im Folgenden etwas genauer beschrieben wird. Dabei werden zunächst die drei in Abbildung 2.11 dargestellten Schnittstellen erläutert.

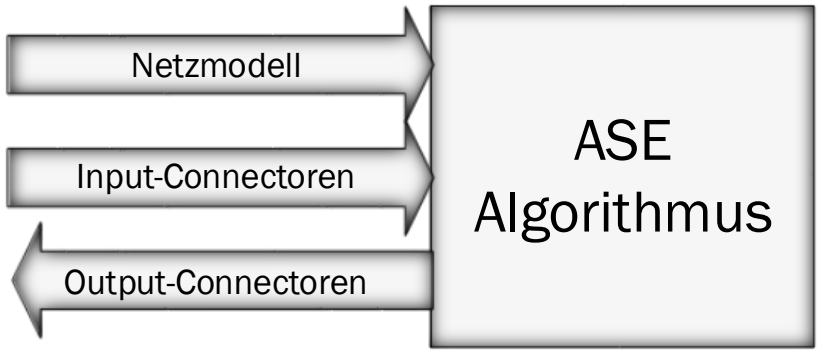


Abbildung 2.11: Schnittstellen des ASE-Algorithmus

Der genaue Datenfluss in der Umsetzung ist im Datenflussdiagramm A.1 auf Seite 87 dargestellt. Dabei wird der Zusammenhang der drei Schnittstellen verdeutlicht und der Ablauf des ASE-Algorithmus in einzelne Operationen aufgeteilt, um genauer zu zeigen für welche Vorgänge welche Daten benötigt werden und wo diese bereitstehen.

2.3.3.1 Netzmodell

Um die Berechnungen durchführen zu können, benötigt der ASE-Algorithmus Informationen über das zugrundeliegende Netz. Daher muss immer die Topologie des Netzes, die sich aus Knoten und Leitungen zusammensetzt, angegeben werden. Die Knoten stehen dabei für einen Verbraucher oder Erzeuger im Netz. Die Leitungen besitzen einige Eigenschaften, welche variabel sind und somit ebenfalls immer definiert werden müssen. Folgende Leitungsparameter können für jede Leitung festgelegt werden (vgl. Abschnitt 2.2.2):

- Länge
- Wirkwiderstände (Resistanzen)
- Blindwiderstände (Reaktanzen)
- Ableitwiderstände (Konduktanzen)
- Kapazitive Blindwiderstände (Suszeptanzen)

Der ASE-Algoritmus unterstützt dabei die Verwendung eines Vierleiter-Netzwerks (vgl. Abb. 2.4 auf S. 6) mit drei Außenleitern A, B, C und dem Neutralleiter N. Jedoch kann für vereinfachte Berechnungen auch ein einphasiges Netz angegeben werden.

2.3.3.2 Input-Connectoren

Die bekannten Messwerte werden dem ASE-Algorithmus über sogenannte Input-Connectoren zur Verfügung gestellt. Für verschiedene Arten von Messdaten gibt es unterschiedliche Connectoren-Typen und ein einzelner Input-Connector gibt dabei immer nur den Wert für eine Phase an – entweder

absolut oder relativ zu einer anderen Phase. Um also einen Knoten vollständig für alle drei Außenleiter und den Neutralleiter zu definieren, sind vier Connectoren nötig. Zudem muss jedem Connector die Genauigkeit der Messstelle in Form der Standardabweichung übergeben werden. Einige wichtige Typen von Connectoren, welche vom ASE-Algorithmus bereitgestellt werden, sind:

- Knotenstrom *(AbsoluteNodalPhaseCurrent)*
- Spannung *(PhaseToEarthVoltage / PhaseToPhaseVoltage)*
- Spannungswinkel *(PMU)*
- Wirkleistung *(SinglePhaseActivePower)*
- Blindleistung *(SinglePhaseReactivePower)*

Die einzelnen Connectoren setzen jeweils auch das dazugehörige Messmodell $h(x)$ (vgl. Formel (2.12) auf Seite 14) um, welches ein elementarer Bestandteil der ASE ist, da es für das Bilden der Jacobi-Matrix benötigt wird (vgl. Formel (2.17) auf Seite 15). So stellen die Connectoren für die Berechnung eine entsprechende Zeile der Jacobi-Matrix bereit.

2.3.3.3 Output-Connectoren

Durch die Angabe von Output-Connectoren wird definiert, welche Werte der ASE-Algorithmus ausgeben soll. Die verfügbaren Typen entsprechen dabei denen der Input-Connectoren und setzen somit ebenso das Messmodell $h(x)$ um. Zusätzlich zu dem berechneten Wert gibt der Algorithmus jeweils die Güte in Form einer Maßzahl der Modellabdeckung an. Hierbei ist die Aussage der Modellabdeckung genau entgegengesetzt zu der in Abschnitt 2.3.2 beschriebenen Deutung. Eine Modellabdeckung von $0°$ steht so für eine exakte Abdeckung des Werts und höhere Werte für eine schlechtere Genauigkeit.

2.4 Datenstromverarbeitung

DSMS sind dafür ausgelegt auf großen kontinuierlichen Datenströmen nahezu in Echtzeit Operationen durchzuführen. In der Theorie wird die Menge an Datenelementen eines Datenstroms als unendlich angesehen [GO10]. Derartige Daten können z. B. von Sensoren aus verschiedensten Anwendungsgebieten stammen. Im Rahmen dieser Arbeit sollen die Daten von mosaik geliefert werden, auf das in Abschnitt 2.5 genauer eingegangen wird.

Bei der Verarbeitung von Daten unterscheiden sich DSMS grundsätzlich von herkömmlichen Datenbankmanagementsystemen (DBMS), die alle Daten persistent speichern. Aufgrund der theoretisch unendlichen Datenmenge in einem Datenstrom, ist es in einem DSMS nicht möglich alle Daten dauerhaft zu speichern. Daher werden in DSMS kontinuierliche Anfragen gestellt, welche die ankommenden Daten verarbeiten und ein kontinuierliches Ergebnis als Ausgabedatenstrom liefern. Somit werden nur die wirklich benötigten Informationen aus den Daten extrahiert und ggf. persistent gespeichert [GO10], [BGJ+09], [GO03], [See04].

Es gibt einige DSMS, die als Forschungsprojekte entstanden sind, wie z. B. Odysseus [BGJ+09], STREAM [ABB+04], Borealis [AAB+05], PIPES [KS04] oder TelegraphCQ [CCD+03], die jedoch bis auf Odysseus schon seit einigen Jahren nicht mehr aktiv weiterentwickelt werden. Zudem gibt es kommerzielle Produkte, wie z. B. StreamBase [Str14], IBM Infosphere Streams [Inf14] oder SAP

Sybase CEP [Syb14] (früher Coral8). Für die Einbindung des ASE-Algorithmus wurde Odysseus gewählt, da es noch aktiv weiterentwickelt wird und der Quellcode frei verfügbar ist.

2.4.1 Architektur von Odysseus

Ziel von Odysseus ist es, eine möglichst leicht anpassbare Anfrageverarbeitung auf Datenströmen für die verschiedensten Anwendungsfälle zu bieten und dabei unabhängig von bestimmten Datenmodellen zu agieren. Um diese Flexibilität zu gewährleisten, verwendet es eine streng komponentenbasierte Architektur, deren Umsetzung auf OSGi [osg11] aufbaut. OSGi ist eine Spezifikation für dynamische Softwareplattformen, die besonders Wert auf Modularität legt und daher vorsieht das System in verschiedene Plug-Ins bzw. Bundles zu unterteilen. Die einzelnen Plug-Ins kommunizieren untereinander entweder direkt oder über sogenannte Services. Das Hinzufügen, Aktualisieren und Entfernen von Plug-Ins ist dabei auch im laufenden System möglich. Als Implementierung von OSGi kommt in Odysseus das von der Eclipse Foundation entwickelte Equinox zum Einsatz, welches Bestandteil der Eclipse IDE ist. Durch diese Modularisierung ist es jederzeit möglich Odysseus um weitere Features zu erweitern oder diese wieder zu deaktivieren.

Zusätzlich werden Flexibilität und Anpassungsmöglichkeit an vielen Stellen durch die Verwendung von fixen und variablen Punkten für die funktionale Basis und die anpassbaren Elemente erreicht. Der Aufbau von Odysseus lässt sich in Abbildung 2.12 erkennen und die wichtigsten Aspekte sollen nun erläutert werden.

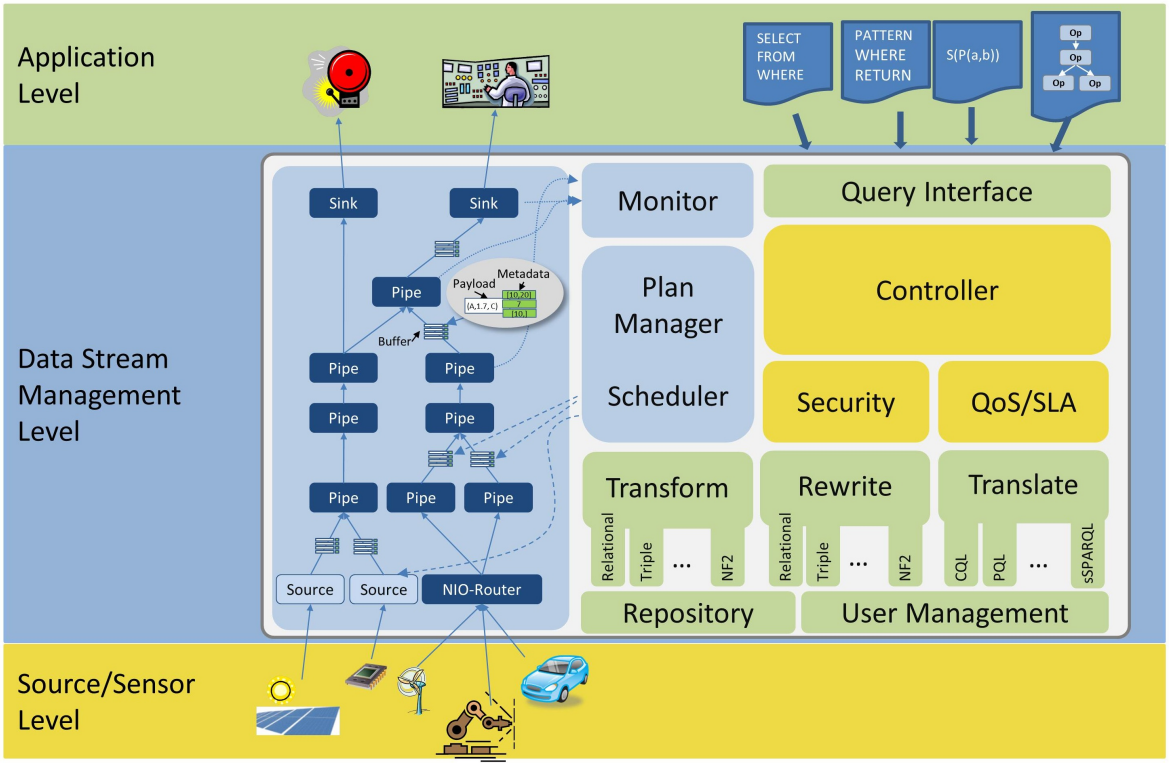


Abbildung 2.12: Architektur von Odysseus [BGJ+09]

Die Übersetzungskomponente (*Translate*) wandelt Anfragen aus verschiedenen Anfragesprachen in logische Anfragepläne um. Die unterstützten Anfragesprachen sind die Odysseus-eigene Procedural Query Language (PQL), die an SQL angelehnten Continuous Query Language (CQL) [ABB+04] und SASE+ [GWC+07]. Am besten mit den Funktionalitäten von Odysseus verknüpft ist dabei PQL, die später noch genauer erläutert wird. Die Anfragepläne bestehen aus logischen Operatoren, die noch keine Algorithmen implementieren und daher unabhängig vom Datenmodell sind. Bisher unterstützt Odysseus z. B. das relationale Datenmodell, Key-Value-Objekte und RDF-Daten.

Für die Restrukturierung (*Rewrite*) und Optimierung der logischen Anfragepläne ist die Restrukturierungskomponente zuständig. Dies ist nötig, da die Ergebnisse der Übersetzungskomponente nicht zwangsläufig optimal für die Ausführung sind. Der Fixpunkt ist hier eine *Ruleengine*, die durch variable Restrukturierungsregeln erweitert und angepasst werden kann. Beispiele für derartige Regeln sind z. B. das Entfernen von unnötigen Selektionen und Projektionen oder das Zusammenfügen oder Auseinanderziehen von Selektionen und Projektionen.

Damit die algorithmische Logik in die Anfragepläne gelangt, wandelt die Transformationskomponente (*Transform*) die logischen in physische Operatoren um. Der Fixpunkt ist hier wieder eine *Ruleengine*, die mit Transformationsregeln versehen werden kann. Die physischen Operatoren enthalten schließlich die Algorithmen für die Verarbeitung der Daten und müssen daher für jedes zu verwendende Datenmodell implementiert sein. Durch die Transformationsregeln wird dann der zum Datenmodell gehörende physische Operator gewählt.

Die Auführungskomponente setzt eine einheitliche Ausführung der Anfragepläne in Odysseus um (vgl. Abb. 2.12 links). Die Physischen Operatoren werden dabei über einen Publish-Subscribe-Mechanismus verknüpft und die Unabhängigkeit von bestimmten Datenmodellen wird durch Generics und das Strategie-Muster umgesetzt. Durch die Implementierung von Metadaten-Interfaces lassen sich einem Datenstrom auch flexibel Metadaten, wie z.B. Gültigkeitsintervalle zur Verwendung von Fenstern (vgl. Abschnitt 2.4.2) oder Latenzen zum Durchführen von Benchmarks, hinzufügen.

Damit die laufenden Anfragen überwacht werden können, gibt es zudem eine Monitoringkomponente (*Monitor*), die mit Hilfe von abstrakten Events den Fixpunkt des Monitorings liefert [BGJ+09].

2.4.2 Fenster

Eine wichtige Rolle in DSMS spielen Fenster, die verwendet werden, um das Blockieren von Operatoren zu verhindern. Dazu werden Berechnungen auf einen Teil des theoretisch unendlichen Datenstroms beschränkt. Dies ist sinnvoll, da die neuesten Daten für eine Anfrage meist am interessantesten sind. Auf den durch Fenster zusammengefassten Daten lassen sich z. B. Aggregationen wie der Durchschnitt, das Maximum oder Minimum, oder Joins durchführen [GO03], [See04]. In Odysseus werden drei verschiedene Typen von Fenstern umgesetzt, bei denen die Gültigkeit über Zeitstempel jeweils anders definiert wird.

Elementfenster Zum einen gibt es Fenster auf Basis von Elementen. Dabei wird die Größe des Fensters als Anzahl von darin enthaltenen Datenelementen angegeben. Die n aktuellsten Elemente werden dann durch das Fenster für Berechnungen bereitgehalten.

Zeitfenster Des Weiteren kommen Zeitfenster zum Einsatz. Dabei liegen die letzten Daten innerhalb eines vorgegebenen Zeitraums im Fenster. Zudem wird bei Zeitfenstern zwischen gleitenden (*Sliding Window*) und springenden (*Tumbling Window*) Fenstern unterschieden. Beim Sliding

Window gleitet das Fenster über die Daten und die Mengen im Fenster können sich über die Zeit überschneiden, während die Mengen im Tumbling Window immer genau disjunkt sind.

Prädikatfenster Als dritter Typ werden in Odysseus Prädikatfenster umgesetzt. Diese Fenster besitzen ein Start- und ein Endprädikat – also einen Ausdruck der Prädikatenlogik – und öffnen bzw. schließen sich, wenn dieser erfüllt ist. Die Prädikate können sich dabei sowohl auf die Daten als auch auf die Metadaten beziehen und können so sehr flexibel eingesetzt werden.

2.4.3 Unterstützte Datenquellen

Dadurch, dass die Datenverarbeitung in Odysseus sehr flexibel durch Transport-, Protocol- und DataHandler gelöst wird, können ohne großen Aufwand verschiedenste Daten- und Protokolltypen zum Einlesen von Daten verwendet werden. Um die Möglichkeiten zu zeigen, sollen einige implementierte Typen aufgezählt werden:

TransportHandler: ZeroMQ, RabbitMQ, File, HDF5, HTTP, IMAP, POP3, Modbus TCP, RS232, SMTP, TCP, Twitter, UDC

ProtocolHandler: CSV, HTML, Document, JSON, NMEA, ByteBuffer, SVM, Text, Tika, XLS, XML, LMS1xx

DataHandler: Relationales Tuple, Prababilistisches Tuple, Key-Value-Objekt

Durch diese Flexibilität von Odysseus ist es ohne großen Aufwand möglich mosaik als Datenquelle zu ersetzen und z. B. direkt Daten von Sensoren aus dem Energienetz zu verwenden. Dadurch wäre es nicht nötig an der ASE-Integration an sich etwas zu verändern, sondern es müssten lediglich die Sensordaten für die Verarbeitung aufbereitet werden. Dazu gibt es in Odysseus bereits eine Vielzahl von Operatoren, die z. B. Selektionen und Projektionen oder auch Glättungen der Daten durchführen können.

2.4.4 PQL

Für alle Anfragen im Rahmen dieser Arbeit wird die Anfragesprache PQL verwendet, da diese alle vorhandenen Operatoren von Odysseus unterstützt. Daher soll im Folgenden eine kleine Übersicht über die Verwendung gegeben werden. Im Vergleich zu den anderen beiden unterstützten Sprachen CQL und SASE ist PQL nicht deklarativ, sondern eher prozedural aufgebaut. Es ermöglicht das Erstellen und Verbinden von logischen Operatoren auf folgende Weise:

```
1 OPERATOR({parameter1, paramter2, ...}, operatorinput)
```

Listing 2.1: Operator in PQL

Als Parameter eines Operatoren können die primitiven Datentypen *Integer*, *Long*, *Double* und *String* verwendet werden. Komplexere Datentypen sind *Predicate*, *List* und *Map*. Predicate ist dabei ein logischer Ausdruck in Form eines Strings, welcher nach wahr oder falsch aufgelöst werden kann. Der Typ List beinhaltet eine Menge an Werten und Map Key-Value-Paare.

Zwischenergebnisse der Anfrage können auch Variablen zugeordnet werden, um somit mehr Übersichtlichkeit der Anfragen zu erhalten. Während die über ein einfaches = definierten Zwischenergebnisse nur während des Übersetzens dieser einen speziellen Anfrage verfügbar sind, können sie auch

über := als Views definiert werden. Diese werden global verwaltet und können daher auch in anderen Anfragen verwendet werden.

```
result1 = OPERATOR1({parameter1})
result2 = OPERATOR2({parameter2}, result1)
view := result2
```

Listing 2.2: Variablen und Views in PQL

Als konkretes Beispiel von einigen wichtigen Operatoren sei folgende Anfrage gegeben:

```
input = ACCESS({
   source='json',
   wrapper='GenericPull',
   transport='File',
   protocol='CSV',
   dataHandler='Tuple',
   options=[['filename','data.csv']]
})
result1 = PROJECT({attributes = ['timestamp', 'info']}, input)
result2 = SELECT({predicate='timestamp > 1162304033'}, result1)
```

Listing 2.3: PQL Beispiel

Das Beispiel-Script zeigt zunächst wie mit dem ACCESS-Operator die Daten als Tupel aus einer CSV-Datei ausgelesen werden. Über die Parameter wird hierbei angegeben welche Data-, Protocol- und TransportHandler verwendet werden sollen. Der Options-Parameter wird in Form einer List angegeben, welcher in diesem Fall wiederum eine Liste mit einem Key-Value-Paar für den Dateinamen beinhaltet. Anschließend werden in der Projektion lediglich die beiden Attribute *timestamp* und *info* herausgefiltert. Der Attributes-Parameter wird dabei als Liste von Strings angegeben. In der Selektion werden schließlich alle Tupel herausgefiltert, welche den Predicate-Parameter nicht erfüllen.

2.4.5 Odysseus Script

Mit Hilfe von Odysseus Script können zusätzliche Funktionen beim Schreiben von Anfragen verwendet werden. Zur Verwendung werden Kommandos genutzt, die immer mit einem # beginnen.

Variablen Variablen ermöglichen das mehrfache Verwenden von Werten. Sie können über das Kommando #DEFINE angelegt werden und über ${ } eingebunden werden. Für das Berechnen einer Variable aus anderen Variablen gibt es das Kommando #EVAL.

```
#DEFINE time 1162304033
#EVAL time2 = time + 100
result2 = SELECT({predicate='timestamp > ${time}'}, result1)
```

Listing 2.4: Beispiel für Odysseus Script Variablen

Konstanten Konstanten können verwendet werden, um besondere Systemwerte zu verwenden, welche automatisch gesetzt werden. Die Konstante *NOW* steht z. B. für den aktuellen Zeitpunkt oder *WORKSPACE* für den absoluten Pfad zum aktuellen Workspace.

```
1 options=[['filename','${WORKSPACE}/data.csv']]
```

Listing 2.5: Beispiel für Odysseus Script Konstanten

Kontrollflüsse Zudem können Kontrollflüsse eingesetzt werden, um effiziente Anfragen zu erstellen. So können z. B. das *LOOP*-Kommando für Schleifen oder das *IF*-Kommando für Bedingungen verwendet werden.

```
1 #LOOP i 0 UPTO 10
2     #IF toInteger(time2) > 1234
3     result2 = SELECT({predicate='timestamp > ${time}'}, result1)
4     #ENDIF
5 #ENDLOOP
```

Listing 2.6: Beispiel für Odysseus Script Kontrollflüsse

2.5 Mosaik

Durch die immer weiter steigende Komplexität des Energiesystems wird die Entwicklung neuer Technologien eine immer größere Herausforderung. Der Weg zu mehr erneuerbare Energien sorgt für immer mehr verteilte Energieerzeuger und auch die Heterogenität und riesige Anzahl von Erzeugern und Verbrauchern führt zu Problemen. So wird es immer aufwendiger neue Kontrollstrategien für derartig aufgebaute Energienetze zu simulieren und zu testen. Die am OFFIS Institut für Informatik in Oldenburg entwickelte Smart Grid Simulationsumgebung mosaik setzt genau bei diesem Problem an, indem sie die Möglichkeit bietet leicht konfigurierbare Simulationsszenarien zu erstellen. Der Schwerpunkt liegt dabei auf der Möglichkeit der flexiblen Einbindung von bereits vorhandenen Simulationsmodellen einzelner Komponenten. Dadurch muss für eine Simulation kein neues Simulationsmodell für einzelne Komponenten reimplementiert werden, sondern es kann auf bereits vorhandene Umsetzungen zurückgegriffen werden. Die so eingebundenen Simulatoren können dabei in beliebigen Programmiersprachen umgesetzt sein, solange sie die SimAPI von mosaik implementieren [RLS+13].

Anhand der mit mosaik beschriebenen Simulationsszenarien können Tests mit Steuerungsalgorithmen durchgeführt werden. So sollen Daten aus mosaik auch als Grundlage für die Berechnungen des ASE-Algorithmus verwendet werden. Das Konzept hinter mosaik wird ausführlich in [Sch14] beschrieben. Bei der darin beschriebenen Implementierung handelt es sich jedoch noch um mosaik 1.0. Inzwischen ist bereits mosaik 2.0 veröffentlicht und dient auch als Grundlage für diese Arbeit. Für die nun folgenden Ausführungen wird jedoch noch mosaik 1.0 als Grundlage verwendet, da dieses und insbesondere auch seine theoretischen Grundlagen sehr ausführlich in [Sch14] dokumentiert sind, während die aktuelle Version von mosaik lediglich in der Online-Dokumentation beschrieben ist. Auf die Unterschiede der beiden Versionen soll bei der Beschreibung der Umsetzung eingegangen werden.

2.5.1 Layer-Konzept von mosaik

Das Architektur-Konzept von mosaik baut auf verschiedenen Ebenen auf, die in Abbildung 2.13 visualisiert und im Folgenden nur grob erläutert werden. Auf den *Syntactic Layer*, *Semantic Layer* und *Scenario Layer* wird anschließend noch näher eingegangen, da sie für die vorgenommene Anbindung

an Odysseus relevant sind. Das Konzept entspricht nicht bis ins Detail der Umsetzung, sondern dient eher der theoretischen Auseinandersetzung mit verschiedenen wichtigen funktionellen Aspekten. Genauere Ausführungen über das komplette Konzept und die Umsetzung sind in [Sch14, S. 78ff.] zu finden.

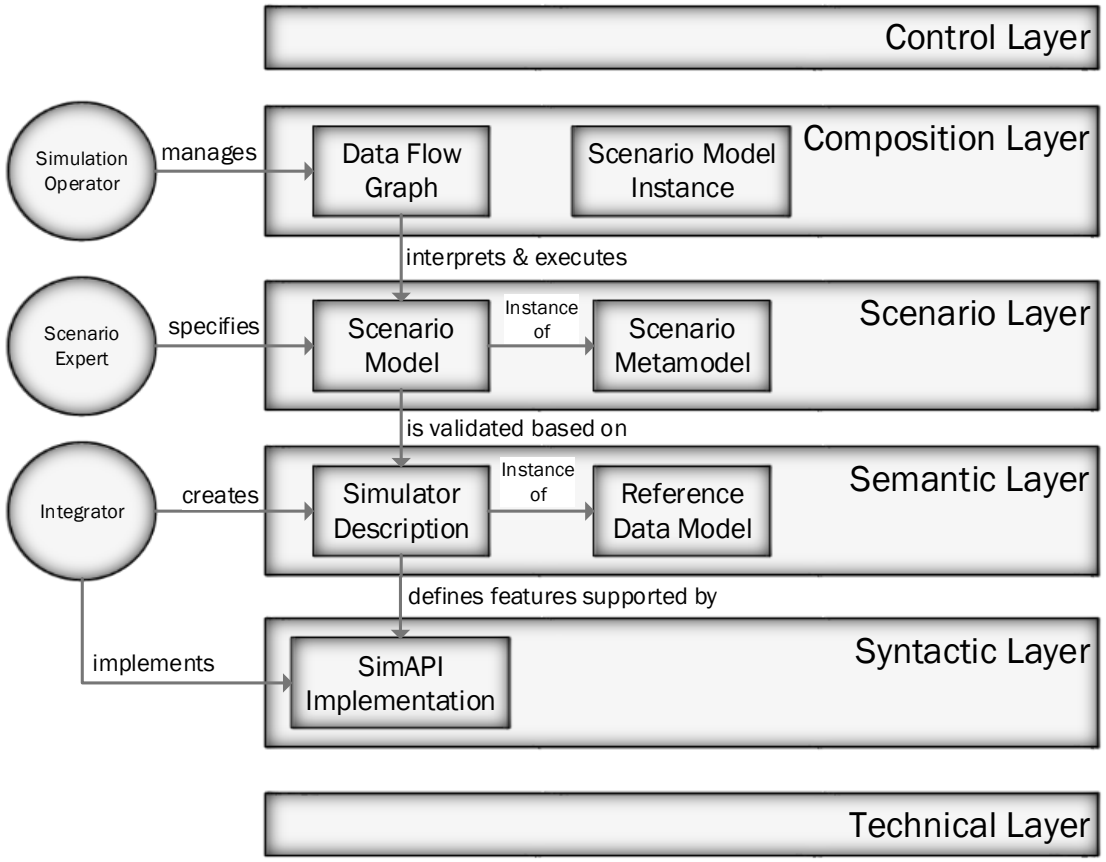


Abbildung 2.13: Layer-Architektur in mosaik

Technical Layer: Der Technical Layer beinhaltet Computer- und Netzwerkhardware und die Teile von mosaik, die sich um die Ausführung von weiteren Prozessen, z. B. von Simulatoren kümmern. Er wird in [SS12] genau beschrieben.

Syntactic Layer: Der Syntactic Layer stellt die SimAPI als Interface bereit, welches von jedem Simulator implementiert werden muss, der in mosaik verwendet werden soll. Durch die SimAPI wird die Kommunikation zwischen mosaik und den Simulatoren sichergestellt (siehe auch Abschnitt 2.5.1.1).

Semantic Layer: Der Semantic Layer beinhaltet ein Referenz-Datenmodell zur Beschreibung der von den Simulatoren ausgetauschten Daten. Darin werden z. B. Informationen wie die unterstützen Schrittgrößen, Konfigurationsparameter inklusive ihrer gültigen Wertebereiche, maximale Anzahl von möglichen Instanzen oder benötigte und bereitgestellte Datenflüsse definiert. Dieser Layer

bildet dabei auch die Basis für die Szenariodefinition und automatische Zusammensetzung und Validierung der Simulation auf den höheren Layern (siehe auch Abschnitt 2.5.1.2).

Scenario Layer: Der Scenario Layer beschreibt das vom Benutzer gewünschte Simulationsszenario, indem er z. B. die Anzahl an Modellen, ihre Konfiguration und die Verbindungen beinhaltet. Die Informationen können dabei mit Hilfe des Semantic Layers validiert werden (siehe auch Abschnitt 2.5.1.2).

Composition Layer: Der Composition Layer beinhaltet Konzepte und Methoden zum Zusammensetzen, Validieren und Ausführen der Simulation. Dazu werden die in den unteren Layern erstellen Daten verwendet, um die richtigen Simulatoren anzusteuern.

Control Layer: Der Control Layer stellt die ControlAPI bereit, mit welcher sowohl auf die Szenariostruktur als auch auf die Datenstrukturen des Semantic Layers zugegriffen werden kann. Sie wird verwendet, um z. B. Steuerungsalgorithmen zu testen.

2.5.1.1 Syntactic Layer

Der Syntactic Layer stellt die SimAPI bereit, welche es ermöglicht Simulatoren in mosaik einzubinden. Dazu müssen diese die SimAPI entweder direkt oder über eine Schnittstelle implementieren [Sch14, S. 95]. Die SimAPI stellt die wichtigsten Methoden zur Kommunikation zwischen mosaik und den Simulatoren bereit, welche nun kurz vorgestellt werden. Dabei ist zu beachten, dass es Unterschiede in der SimAPI von mosaik 1.0 und 2.0 gibt. An dieser Stelle soll die ursprüngliche in [Sch14, S. 288] beschriebe API als Grundlage dienen. Auf die aktuelle Version, welche in der Dokumentation von mosaik 2.0 [mos14] erläutert wird, soll erst später bei der Umsetzung der Anbindung in Abschnitt 4.1.2 eingegangen werden.

init(step_size, sim_params, model_config): Dies ist die erste Methode, die mosaik in einem Simulator ausführt. Durch sie wird der Simulator konfiguriert und das Simulationsmodell erstellt.

get_relations(): Gibt eine Liste von Tupeln zurück, welche Paare von IDs verbundener Objekte enthalten.

get_static_data(): Gibt eine Liste aller Objekte und einer jeweiligen Map mit den statischen, also während der Simulation nicht veränderlichen, Attributen eines jeden Objekts zurück.

get_data(model_name, etype, attributes): Gibt die aktuellen Werte der angeforderten Attribute für die Objekte des gewünschten Typs und des gewünschten Modelnamens zurück.

set_data(data): Das übergebene Data-Objekt enthält eine Liste mit Key-Value-Einträgen für einen oder mehrere Simulatoren. Die entsprechenden Werte der übermittelten Simulatoren und Attribute werden dementsprechend angepasst.

step(): Durch Aufrufen der Step-Methode wird die Simulation des Simulators um die beim Initialisieren festgelegte step_size durchgeführt. Der Rückgabewert ist die aktuelle Simulationszeit.

2.5.1.2 Semantic und Scenario Layer

Der Semantic Layer bildet mit seinem Metamodell für die Simulation die Grundlage für die Definition von Szenarien, die wiederum über ein Metamodell beschrieben wird. Aufbauend auf diesem Szenario-Metamodell wird die Mosaik Specification Language (MoSL) in [Sch11] eingeführt und in [Sch14]

weiter ausgeführt. Umgesetzt wurde MoSL mit XText[1], einem Framework der Eclipse Foundation zur Entwicklung eigener DSL. In mosaik 2.0 wird MoSL in der beschriebenen Form jedoch nicht mehr verwendet, stattdessen ist ein Nachfolger in der Entwicklung. MoSL wird jedoch im Folgenden trotzdem erläutert, da das Prinzip einer DSL zu Beschreibung von Szenarien auch für eine Anbindung an Odysseus interessant ist.

Zur Beschreibung eines Szenarios sind Informationen über die zu simulierenden Einheiten, ihre Anzahl und ihre Verbindung zum Stromnetz nötig. Zudem benötigen die Simulatoren unterschiedliche Parameter zur genauen Konfiguration. Diese Parameter lassen sich in MoSL sehr frei definieren, damit die Anforderungen von verschiedensten Simulatoren erfüllt werden können. Da die mit mosaik zu simulierenden Szenarien möglicherweise sehr groß werden, ist die Skalierbarkeit eine wichtige Anforderung an eine Beschreibungssprache. Die Handhabbarkeit von besonders großen Netzen soll dadurch verbessert werden, dass nicht zwingend jede einzelne Verbindung im Netz einzeln definiert wird, sondern auch die Definition über Regeln und Statistiken möglich ist. So kann z. B. angegeben werden, dass jeder dritte Haushalt eine PV-Anlage besitzt, ohne genau festzulegen welche Häuser dies sind. Dies ist besonders für Zukunftsszenarien oder für sonstige Szenarien, für welche keine genaueren Daten vorliegen, sinnvoll [Sch14, S. 150f.].

2.5.2 Architektur von mosaik

Grundsätzlich ist mosaik wie in Abbildung 2.14 gezeigt konzipiert. Mit Hilfe eines Szenario Editors soll das zu simulierende Szenario auf dem Client konfiguriert werden. Anschließend läuft die Zusammenstellung des Szenarios und die Simulation auf dem mosaik-Server ab und die Ergebnisse werden in eine HDF5-Datenbank geschrieben. Auf dieser Datenbank können schließlich wieder auf der Client-Seite vom Benutzer Analysen durchgeführt werden.

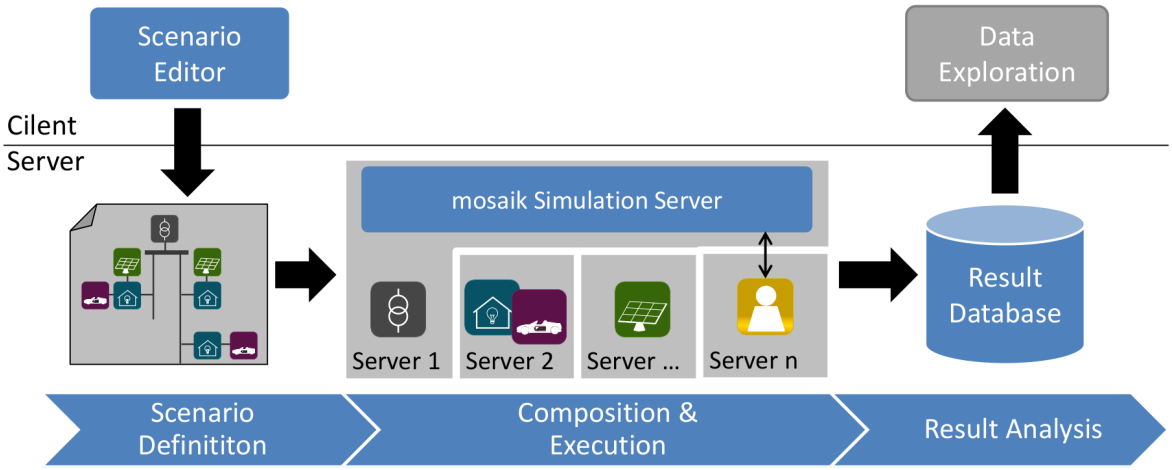


Abbildung 2.14: Überblick über die mosaik Architektur [Sch14, S. 9]

Wie bereits in den vorherigen Abschnitten beschrieben, soll im Rahmen dieser Arbeit jedoch direkt auf dem von mosaik erzeugten Datenstrom gearbeitet werden. Daher ist ein Teilaspekt dieser Arbeit auch die Anbindung von mosaik an Odysseus.

[1] `www.eclipse.org/Xtext`

2.6 Zusammenfassung

In diesem Kapitel wurden die Grundlagen dieser Arbeit erläutert. Dazu wurde zunächst die Modellierung von Leitungen und Netzen vorgestellt mit Hilfe des π-äquivalenten Ersatzschaltbildes die wichtigen Parameter für die Leitungsmodellierung Resistenz, Reaktanz, Konduktanz und Suszeptanz vorgestellt. Bei der Netzmodellierung spielt die Admittanzmatrix eine sehr wichtige Rolle. Sie gibt die Admittanzen (Scheinleitwerte) der Leitungen zwischen den einzelnen Knoten wieder.

Anschließend wurde die allgemeine die Leistungsflussrechnung und im speziellen das NR-Verfahren erläutert. Dieses wird zur Planung und Überwachung von Leitungen verwendet, da es die Leistungsflüsse – also die Belastung – ermittelt. Es beruht dabei auf dem Prinzip des Newton'schen Näherungsverfahrens und geht somit iterativ vor. Da die Messwerte aus dem Netz jedoch Fehler aufweisen können oder durch Ausfälle fehlen können, wird zudem die State Estimation verwendet, um die Fehler in den Daten zu finden und soweit möglich zu korrigieren. Diese beiden Verfahren funktionieren jedoch nur auf zumindest vollständig bestimmten Daten, da eine Matrixinversion durchgeführt werden muss und die Matrix somit invertierbar sein muss.

Die ASE verbindet daher das NR-Verfahren und die State Estimation und kommt auch mit unter- und überbestimmten Netzen zurecht. Dazu wird eine Pseudoinverse durch Singulärwertzerlegung gebildet. Mit dem Verfahren kann auch in unterbestimmten Fällen eine Aussage über nicht bestimmte Messstellen getroffen werden. Wie zuverlässig dieser Wert ist, wird über die Modellabdeckung angegeben. Das Verfahren wurde bereits implementiert und stand zur Verwendung bereit.

Im Rahmen dieser Arbeit wurde das ASE-Verfahren in das DSMS Odysseus integriert. DSMS sind im Allgemeinen auf die kontinuierliche Verarbeitung von Datenströmen spezialisiert. Der Datenstrom wird dabei als in der Theorie unendlich angesehen, sodass nicht alle Daten gespeichert werden können, sondern versucht wird, mit Hilfe von kontinuierlichen Anfragen, möglichst viele Informationen zu gewinnen, um z. B. bei einer Überlastung des Netzes einen Alarm auszugeben oder den aktuellen Zustand des Netzes in einer Leitzentrale anzuzeigen.

Als Datenquelle für die Berechnungen in Odysseus kommt die Smart Grid Simulationsumgebung mosaik zum Einsatz. Zur Simulation können dabei Simulatoren eingebunden werden, die aus verschiedenen Quellen stammen und auch in verschiedenen Programmiersprachen geschrieben sein können. Somit können aufwendige Neuentwicklungen für die Simulation vermieden werden. Auf dem simulierten Szenario lassen sich zudem Steuerungsalgorithmen testen. Bisher wurden die Ergebnisse der Simulation in eine HDF5-Datenbank geschrieben und anschließend ausgewertet, was im Rahmen dieser Arbeit um eine Online-Verarbeitung in Odysseus erweitert wird.

3 Entwurf

Dieses Kapitel legt den Entwurf zur Umsetzung des ASE-Algorithmus in Odysseus genauer dar. Dazu werden zunächst die Anforderungen erläutert und anschließend die verschiedenen Möglichkeiten für die Anbindung von mosaik und Odysseus erörtert. Daraufhin folgen die Umsetzung des ASE-Operators in Odysseus sowie eine Diskussion der Visualisierung.

3.1 Anforderungen

Zu Beginn des Entwurfs sollen die Ziele der Umsetzung, vorgegebene Systemgrenzen und Systemabhängigkeiten beschrieben werden. Dazu werden die Anforderungen hergeleitet und in Tabelle 3.1 zusammengefasst. Zudem wird das zu entwickelnde System in seine Umgebung eingebettet und die Abhängigkeiten erläutert, wie in Abbildung 3.1 zu sehen.

Im Rahmen dieser Arbeit soll die beschriebene ASE-Berechnung des Netzzustands in Odysseus integriert werden (**O**-1). Dadurch lässt sich die Spezialisierung von DSMS auf die Verarbeitung von großen kontinuierlichen Datenmengen nutzen, wie sie bei der Überwachung von Stromnetzen auftreten.

Der erste Teil der Umsetzung ist dabei die Anbindung von mosaik und Odysseus (**A**-1). Dabei soll auf Unabhängigkeit geachtet werden, sodass mosaik später als Datenquelle austauschbar ist (**A**-2). So soll es sich z. B. durch echte Sensorwerte ersetzen lassen. Um dies zu simulieren ist es wichtig, dass Odysseus sich in den von mosaik erzeugten Datenstrom ein- und ausklinken kann (**A**-3). Zudem bietet sich durch die geschaffene Anbindung von mosaik und Odysseus die Möglichkeit, zukünftig in weiteren Projekten oder Studienarbeiten mit Odysseus Verarbeitung von mosaik-Daten vorzunehmen (**A**-4, **A**-5). Damit die Simulation in mosaik durch die Anbindung nicht gestört wird, soll die Simulation dabei möglichst nicht blockieren (**A**-6).

Der zweite Teil ist die Integration der ASE-Berechnung in Odysseus (**O**-2). Diese lässt sich wiederum aufteilen in die Implementierung des ASE-Operators und die Visualisierung der am ASE-Operator anfallenden Daten (**O**-3, **O**-4, **O**-5). Durch die Umsetzung lassen sich anschließend in passenden Evaluationsszenarien die Ergebnisse des ASE-Algorithmus für verschiedene Konfigurationen miteinander vergleichen (**E**-1, **E**-2).

Mögliche Nutzer der ASE-Berechnung in Odysseus werden hauptsächlich Energieexperten sein, da für das Konfigurieren des ASE-Algorithmus und zum Interpretieren der Ergebnisse gewisse energietechnische Vorkenntnisse vorhanden sein müssen. Da nicht davon ausgegangen werden kann, dass diese Nutzergruppe bereits Erfahrungen in der Verwendung von Odysseus oder DSMS im Allgemeinen hat, soll die Verwendung des ASE-Operators möglichst einfach möglich sein (**O**-6).

Die im Rahmen dieser Arbeit durchzuführenden Implementierungen sind in den zwei Systemen mosaik und Odysseus vorzunehmen. Dabei wird an verschiedenen Stellen eingegriffen. In Abbildung 3.1 ist der stark vereinfachte erste Entwurf des geplanten Systems dargestellt. Zunächst muss ein Konzept zur Anbindung von mosaik und Odysseus erstellt werden, sodass die Daten aus mosaik versendet und in Odysseus empfangen werden können. Dabei müssen ggf. neue Sender oder Empfänger umgesetzt werden oder es kann dabei auf bereits vorhandene Mechanismen zurückgegriffen werden. Anschließend ist ein Szenario in mosaik zu erstellen, welches sicherstellt, dass die benötigten Daten gesendet werden. Es ist nicht notwendig in den mosaik-Simulationsserver als solchen

Kategorie	Anforderung
Anbindung	**A**-1 Odysseus muss für die ASE-Berechnung Daten aus mosaik verwenden können. **A**-2 Mosaik soll als Datenquelle für die ASE-Berechnung austauschbar sein. **A**-3 Odysseus sollte sich jederzeit in den von mosaik erzeugten Datenstrom ein- und ausklinken können. **A**-4 Das Senden der Daten in mosaik soll allgemein umgesetzt werden, sodass es auch für andere Anwendungsfälle wiederverwendbar ist. **A**-5 Das Senden der Daten in mosaik soll für zukünftige Verwendung der mosaik-Odysseus-Anbindung zur Verfügung gestellt werden. **A**-6 Die Anbindung an Odysseus soll die Simulation in mosaik nicht blockieren.
Odysseus	**O**-1 Die ASE-Berechnung muss in Odysseus durchführbar sein. **O**-2 Die ASE-Berechnung soll in einem einminütigem Intervall in Echtzeit auf den Simulationsdaten ablaufen können. **O**-3 Das dem ASE-Operator zugrunde liegende Stromnetz sollte visualisiert werden. **O**-4 Die Ergebnisse des ASE-Operators sollten visualisiert werden. **O**-5 Die Visualisierung des ASE-Operators soll das Netz möglichst übersichtlich darstellen. **O**-6 Der ASE-Operator soll sich möglichst einfach konfigurieren lassen.
Evaluation	**E**-1 Die Performance der Umsetzung soll überprüft werden. **E**-2 Die Skalierbarkeit der Umsetzung soll überprüft werden.

Tabelle 3.1: Anforderungsliste

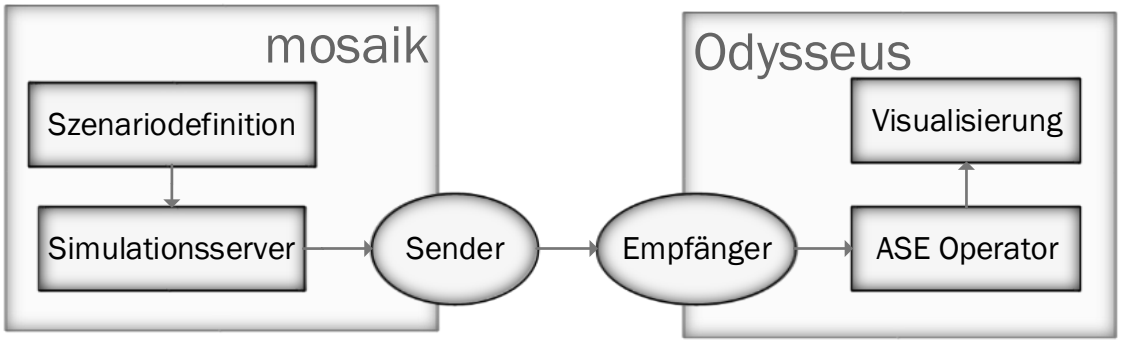


Abbildung 3.1: Produkteinbettung

einzugreifen. Für die Verarbeitung in Odysseus muss der ASE-Operator unter Verwendung der zur Verfügung gestellten Implementierung des ASE-Algorithmus umgesetzt werden und außerdem soll eine Visualisierung des ASE-Operators möglich sein.

3.2 Anbindung von mosaik an Odysseus

Im Rahmen dieser Arbeit dienen Daten aus mosaik als Grundlage für die Berechnungen (**A**-1). Dazu ist zunächst eine Anbindung dieser beiden Systeme notwendig, es soll aber auch darauf geachtet werden, dass später auch andere Datenquellen einsetzbar sind (**A**-2). Im Folgenden werden nun einige Entwurfsentscheidungen genauer erläutert. Zunächst wird auf die Abhängigkeiten von mosaik und Odysseus eingegangen. Anschließend werden die verschiedenen Optionen für die Übertragung zwischen mosaik und Odysseus vorgestellt, die verschiedenen Möglichkeiten diskutiert und schließlich die tatsächlichen Umsetzungen entworfen.

3.2.1 Abhängigkeiten von mosaik und Odysseus

Da mosaik und Odysseus zwei unabhängige Programme sind, stellt sich für die Implementierung einer Anbindung der beiden die Frage, in welche Reihenfolge und von wem sie aufgerufen bzw. gestartet werden sollen. Es wäre z. B. möglich zunächst nur eines der beiden Programme zu starten und daraufhin beim Starten der Anfrage (Odysseus) bzw. des Szenarios (mosaik) das jeweils andere Programm mit den gewünschten Optionen zu starten.

Diese Startmöglichkeit würde sich anbieten, wenn es nur einen bestimmten Anwendungsfall gibt oder sich in jedem vorgesehenen Anwendungsfall diese Reihenfolge der Ausführung als praktisch erweist. Dies würde jedoch den Anforderungen **A**-2 und **A**-4 widersprechen. Damit die Anbindung also auch allgemein weiterverwendet werden kann, sollen mosaik und Odysseus einzeln durch den Anwender gestartet werden. Dies ermöglicht außerdem, dass Odysseus sich jederzeit in den Datenstrom ein- und ausklinken kann, ohne eine explizite Initialisierung mit mosaik zu benötigen (**A**-3).

3.2.2 Möglichkeiten der Kommunikation

Für die Kommunikation zwischen mosaik und Odysseus stehen einige Möglichkeiten zur Auswahl. Odysseus ist darauf ausgelegt möglichst viele verschiedene Transportprotokolle zu unterstützen und auch leicht um neue erweiterbar zu sein. Für viele Protokolle gibt es daher bereits sogenannte TransportHandler wie z. B. für TCP, ZeroMQ oder RabbitMQ. Diese TransportHandler können direkt über die Deklarationssprache PQL in Odysseus zum Empfangen und Senden von Daten verwendet werden.

Mosaik verwendet zum internen Datenaustausch ZeroMQ und bietet für die Einbindung von Simulatoren die SimAPI an, über welche Remote Procedure Calls (RPCs) umgesetzt werden können. Im Folgenden sollen die relevantesten Möglichkeiten vorgestellt werden. Dabei handelt es sich um Sockets, RCPs, RabbitMQ und ZeroMQ, deren Vor- und Nachteile anschließend diskutiert werden.

3.2.2.1 Sockets

Eine direkte Möglichkeit der Kommunikation zwischen verschiedenen Programmen bieten Sockets, die Schnittstellen zwischen der Umsetzung von Netzwerkprotokollen des Betriebssystems und der eigentlichen Software darstellen. Diese Einordnung lässt sich gut anhand des verbreiteten OSI-Modells (*open systems interconnection*) verdeutlichen (vgl. Abb. 3.2).

Das OSI-Modell beschreibt die verschiedenen Schichten eines Netzwerks. Dabei werden die beiden unteren als durch die Hardware gegeben angesehen. Die Netzwerk-Schicht wird durch IPv4 oder

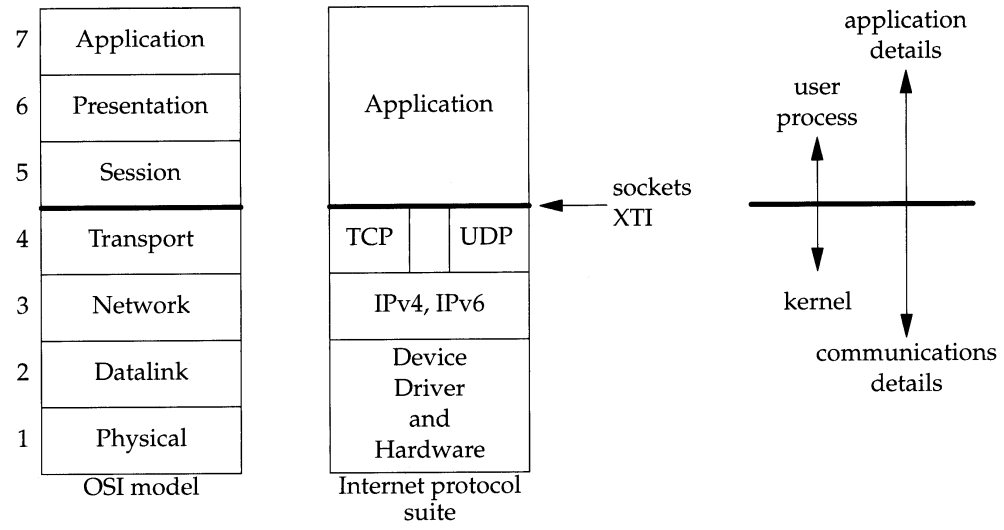


Abbildung 3.2: Einordnung der Sockets in OSI Modell [Ste98, S. 18]

IPv6 umgesetzt und die Transport-Schicht hauptsächlich durch TCP und UDP. Die darüber liegenden Schichten werden in Abbildung 3.2 zusammengefasst als Applikation. Zwischen der Transport-Schicht und der Session-Schicht sind nun die Sockets zu sehen, welche es ermöglichen aus der Applikation heraus auf die Netzwerkschichten zuzugreifen. Wie auf der rechten Seite der Abbildung zu sehen ist, entspricht diese Trennung auch der Grenze zwischen Benutzerprogrammen und dem Betriebssystem, da die unteren vier Ebenen des Modells durch den Betriebssystemkernel bereitgestellt werden [Ste98, S. 18f.].

Die verbreitetsten Typen von Sockets sind Stream Sockets und Datagram Sockets, wovon wiederum Stream Sockets am gebräuchlichsten sind. Sie bieten eine bidirektionale Verbindung und verwenden das TCP Protokoll [Pos81]. Daher wird auch die Reihenfolge von gesendeten Nachrichten sicher erhalten und durch eine Prüfsumme ihre Korrektheit überprüft. Demgegenüber verwenden Datagram Sockets das UDP Protokoll [Pos80] und sind schneller aber fehleranfälliger [Ste98].

Sockets lassen sich in allen gängigen Programmiersprachen und somit auch in den verwendeten Sprachen Java und Python einsetzen, bieten jedoch nur sehr grundlegende Übertragungsfunktionen. Dies bedeutet, dass z. B. aktiv auf das korrekte Aufbauen und fehlerfreie Aufrechterhalten der Verbindung zu achten ist. Außerdem werden die Nachrichten nicht automatisch serialisiert und deserialisiert, weshalb ein geeignetes Protokoll für die Kommunikation entwickelt werden muss.

3.2.2.2 RPC

Aufbauend auf den zuvor beschriebenen Sockets, mit deren Hilfe Daten zwischen verschiedenen Prozessen über das Netzwerk ausgetauscht werden können, ermöglichen RPCs das Ausführen von Prozeduren in anderen Programmen auf dem gleichen oder einem anderen Computer. In Java lassen sich RPCs z. B. über RMI (*Remote Method Invocation*) umsetzen [SFR99], [KS09, S. 1153ff.].

Mosaik bietet die Möglichkeit Simulatoren durch Implementieren der Sim-API umzusetzen. Die so in beliebigen Programmiersprachen entwickelten Simulatoren lassen sich durch RPCs von mosaik

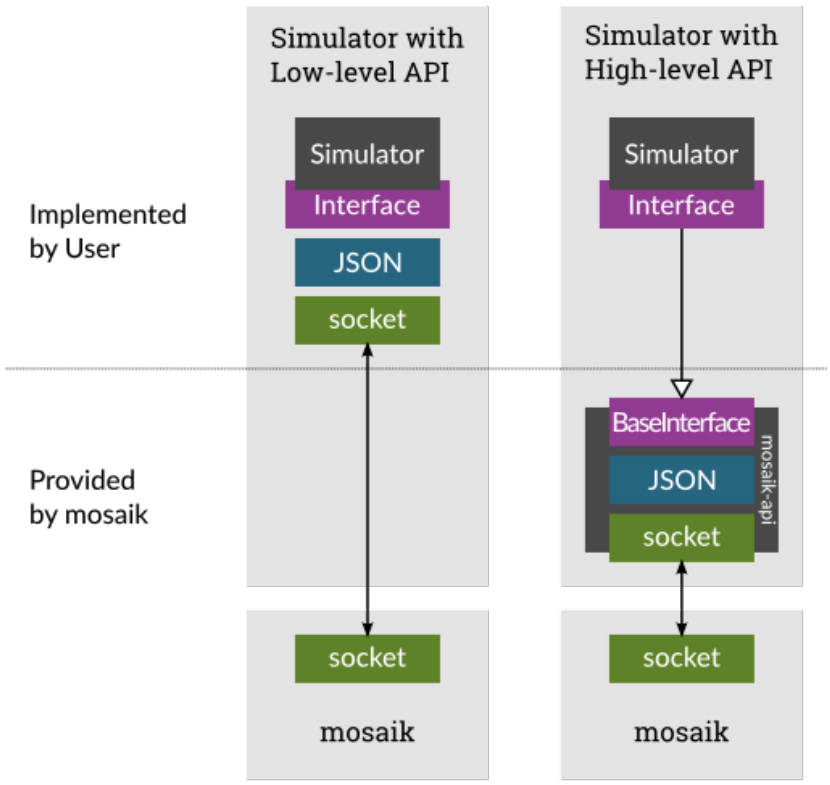


Abbildung 3.3: mosaik API [mos14]

ansteuern. Dabei werden die durch die Sim-API definierten Methoden aufgerufen. Wie in Abbildung 3.3 zu sehen, wird dabei zwischen der High- und der Low-Level API unterschieden.

Die High-Level API steht für Python und Java (in Form des Pakets *mosaik-api-java*[1]) bereit und stellt ein Interface für die Erstellung von Simulatoren bereit und wickelt zudem die Kommunikation zwischen mosaik und dem Simulator ab. Die damit erstellten Simulatoren müssen jedoch aus mosaik heraus gestartet werden. Dazu müssen sie als ausführbare Datei vorliegen, um von mosaik gestartet zu werden und sich anschließend als Socket-Clients zu verbinden. Da Odysseus jedoch unabhängig von mosaik laufen soll, muss auf die Low-Level API zugegriffen werden. Dabei läuft die Kommunikation ebenso wie bei der High-Level API über Sockets ab, jedoch nimmt nicht mosaik die Rolle des Servers ein, sondern Odysseus. Dadurch kann mosaik anschließend auf den laufenden Socket-Server zugreifen und eine Verbindung aufbauen.

Die Kommunikation der Low-Level API läuft über TCP Sockets ab. Die dabei ausgetauschten Nachrichten sehen aus wie in Abbildung 3.4 gezeigt. Der Header beinhaltet die Länge des Payloads in Byte. Der Payload ist eine JSON-Liste, die zunächst den Typ der Nachricht beinhaltet. Hierbei steht 0 für eine Anfrage, 1 für eine erfolgreichen Antwort und 2 für eine fehlerhafte Antwort. Die darauffolgende ID ist eindeutig für jede Nachricht und muss bei Anfrage und sich darauf beziehender Antwort identisch sein. Der Inhalt (*Content*) der Nachricht beinhaltet als Zeichenkette den Namen der aufzurufenden Methode und die zu übergebenen Parameter [mos14].

[1] `https://bitbucket.org/mosaik/mosaik-api-java`

Bei der Umsetzung in Odysseus kann für die Kommunikation über TCP Sockets auf bereits vorhandene TransportHandler zurückgegriffen werden. Somit ist lediglich ein ProtocolHandler zu implementieren, welcher die empfangenen RPCs verwerten kann.

Abbildung 3.4: Nachricht der mosaik Low-Level API [mos14]

3.2.2.3 AMQP (RabbitMQ)

Das Advanced Message Queuing Protocol (AMQP) ist ein von einem Konsortium aus Finanz- und Softwarefirmen entwickeltes Netzwerkprotokoll. Darin wird die Kommunikation zwischen Clients mit Hilfe eines zentralen Servers bzw. Brokers beschrieben. Es gibt zwei unterschiedliche Versionen des Protokolls. Zum einen die Version 0.9, in der das Verhalten des Brokers relativ genau vorgegeben werden kann [AAA+08] und die Version 1.0, welche die Kommunikation von einem Client zu einem anderen und nicht die genaue Konfiguration des Brokers beschreibt [OAS12]. Eine der verbreitetsten Umsetzungen des AMQP ist RabbitMQ, welches AMQP 0.9 umsetzt, aber über ein experimentelles Plug-In auch AMQP 1.0 Kompatibilität liefert. Das Prinzip einer AMQP-Kommunikation soll anhand des RabbitMQ gezeigt werden.

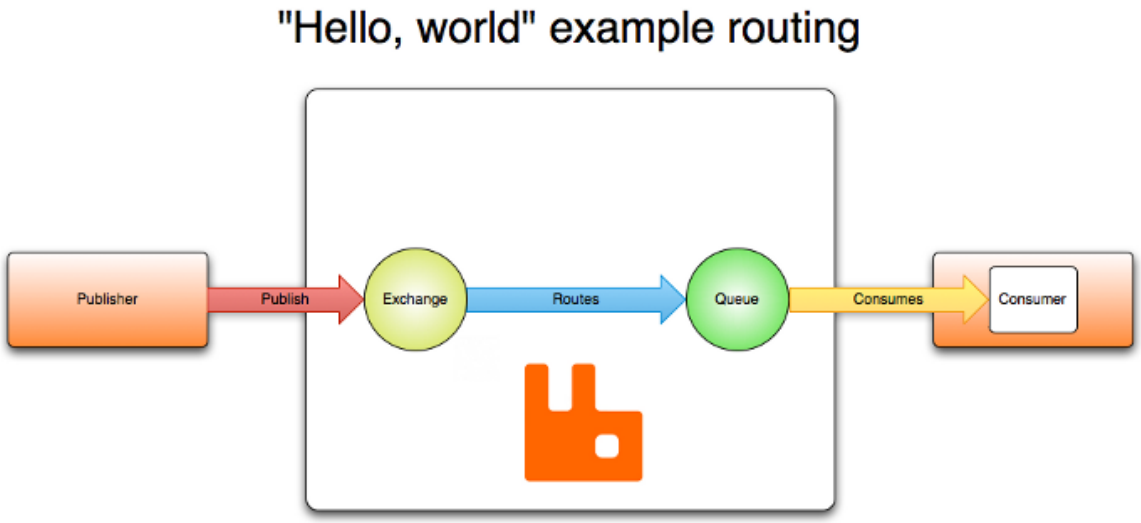


Abbildung 3.5: Routing in RabbitMQ [Piv14]

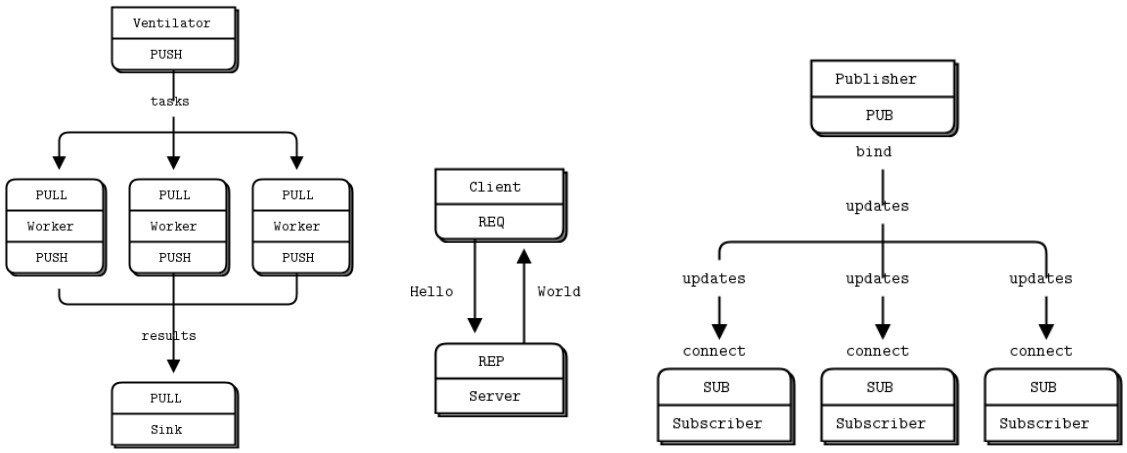
Den standardmäßigen Aufbau einer auf dem AMQP basierenden Verbindung zeigt Abbildung 3.5. Die zu verbindenden Clients lassen sich je nach ihrer Funktion aufteilen in Publisher, die Daten versenden, und Consumer, die Daten empfangen. Der Publisher sendet Nachrichten (publish) an so-

genannte Exchanges, die auf dem Broker liegen und mit einem Briefkasten verglichen werden können. Von dort aus werden die Nachrichten vom Broker über Regeln (auch als Routes oder Bindings bezeichnet) an Queues weitergeleitet. Die in diesen Queues bereitsgestellten Nachrichten werden schließlich entweder pull-basiert vom Consumer abgeholt oder push-basiert vom Broker an den Consumer gesendet. Beim Ausliefern der Nachrichten an die Consumer wartet der Broker immer auf die Bestätigung des korrekten Erhalts bevor er eine Nachricht aus der Queue entfernt. Dies stellt sicher, dass die Nachrichten ankommen, selbst dann wenn Publisher und Consumer nie gleichzeitig verbunden waren, was der große Vorteil einer brokerbasierten Kommunikation ist [AAA+08], [Piv14].

Wie bereits beschrieben, unterstützt Odysseus bereits RabbitMQ als Datenquelle. Die Kommunikation zwischen mosaik und Odysseus über RabbitMQ könnte somit am besten durch einen in Python implementierten Simulator umgesetzt werden, da sich dieser direkt in das in Python geschriebene Mosaik einbinden ließe. Der Simulator muss keine Simulationsdaten berechnen oder bereitstellen, sondern lediglich alle Daten, die bei einem Step an ihn übermittelt werden per RabbitMQ senden. Es handelt sich dabei also um die Verwendung der High-Level API (vgl. Abb. 3.3), da die Kommunikation zwischen dem Simulator und mosaik von der bereitgestellten API übernommen wird.

3.2.2.4 ZeroMQ

ZeroMQ (oder auch ∅MQ) ist ein Messaging System, das auf hohe Performanz beim Nachrichtenaustausch ausgelegt ist. Zudem ist ein zentrales Ziel von ZeroMQ, dass es in großen Szenarien eingesetzt werden kann. Um diese beiden Punkte zu erreichen, wurde komplett auf einen zentralen Server verzichtet und statt einer Serveranwendung eine dezentrale Messaging Library bereitgestellt [Sú14]. Durch den Verzicht auf zentrale Server bietet ZeroMQ eine sehr niedrige Latenz und einen hohen Datendurchsatz. Ohne Server kann es bei Verbindungsproblemen jedoch leicht zum Verlust von Nachrichten kommen. Da ZeroMQs oberste Priorität die Geschwindigkeit ist, bietet es von Haus aus keine Mechanismen zum Absichern der Kommunikation, sondern diese müssen bei Bedarf selbst implementiert werden [Zera].



(a) Pull-Push *(b) Request-Reply* *(c) Publisher-Subscriber*

Abbildung 3.6: Verschiedene Messaging Pattern von ZeroMQ [Zerb]

Da kein Server benötigt wird, ist die Verwendung von ZeroMQ sehr einfach möglich und Implementierungen in vielen verschiedenen Programmiersprachen sorgen für eine große Verbreitung. Um in verschiedenen Anwendungsszenarien arbeiten zu können, gibt es in ZeroMQ verschiedene sogenannte Messaging Pattern, von denen drei im Folgenden kurz erläutert werden sollen (vgl. Abb. 3.6) [Zerb].

Push-Pull: Das Push-Pull-Pattern ist für das Parallelisieren von Tasks vorgesehen. Als feste Punkte gibt es einen Auftraggeber, der die Tasks in Form von Nachrichten an Arbeiter sendet und eine Senke, welche die Ergebnisse der Arbeiter sammelt (vgl. Abb. 3.6a). Das dynamische Einbinden von Arbeitern ist möglich und Tasks werden gleichmäßig verteilt *(fair queueing)*, es kann jedoch zu Ungeleichgewichten kommen, wenn die Arbeiter nacheinander gestartet werden. Zum Beheben dieses Problems sind Implementierungen von mächtigeren Load Balancing Konzepten nötig.

Request-Reply: Beim Request-Reply-Pattern findet die Kommunikation – vergleichbar zum RPC-Pattern – immer abwechselnd zwischen zwei beteiligten Akteueren statt. Der Client ruft dazu immer wieder nacheinander die beiden Methoden *zmq_send()* und *zmq_recv()* auf (vgl. Abb. 3.6b). Durch das abwechselnde Senden von Nachrichten ist dieses Pattern aber anfälliger für Fehler.

Publish-Subscribe: Dieses Pattern ist am besten für Szenarien geeignet, in denen Daten nur von einem Akteur gesendet werden, während ein oder mehrere Clients Daten empfangen. Dazu bindet der Publisher ein Socket auf einem Port *(bind)* und die Subscriber verbinden sich mit dem Socket auf diesem Port *(connect)* (vgl. Abb. 3.6c). Anschließend müssen die Subscriber die Nachrichten des Publishers abonnieren *(subscribe)* und können dabei zusätzlich einen Nachrichtenfilter angeben, falls sie nur bestimmte Nachrichten erhalten möchten. Das Senden und Empfangen läuft hierbei asynchron ab.

Für die Anwendung dieses Pattern ist es wichtig zu wissen, dass von ZeroMQ nicht garantiert wird, dass ein Subscriber nach dem Abonnieren sofort die Nachrichten des Publishers erhält. Durch geringe Verzögerungen von ein paar ms beim Aufbau der Verbindung kann es dazu kommen, dass die ersten Nachrichten nicht ankommen. Daher müssen entweder zusätzliche Synchronisationsmechanismen umgesetzt werden oder die Nachrichten müssen als Datenstrom betrachtet werden, der theoretisch weder Anfang noch Ende hat und somit der Verlust der ersten Nachrichten nicht weiter problematisch ist.

Die Sockets haben – je nach ihrem Typ – entweder Sende- oder Empfangspuffer oder auch beides. Bei großen übertragenen Datenmengen besteht dabei die Gefahr von Pufferüberläufen, wenn ein Client nicht schnell genug die erhaltenen Daten verarbeiten kann. Um dies zu vermeiden wird das HWM *(high-water mark)* Konzept eingesetzt. Über den HWM Parameter lässt sich die Größe der Puffer bestimmen. Wenn sie komplett gefüllt sind, werden weitere Nachrichten gelöscht (z. B. bei PUB-Sockets) oder nicht angenommen (z. B. bei SUB-Sockets).

Wie bereits beschrieben, unterstützt Odysseus auch ZeroMQ bereits als Datenquelle. Die Umsetzung dieser Anbindung in mosaik könnte, ebenso wie im vorangehenden Abschnitt zu RabbitMQ beschrieben, durch die Verwendung der Python High-Level API erfolgen.

3.2.3 Diskussion

Sockets stellen die unmittelbarste Möglichkeit zur Kommunikation zwischen verschiedenen Prozessen dar. Sie bieten jedoch auch nur die grundlegenden Funktionalitäten an und benötigen somit noch einigen zusätzlichen Aufwand für eine zuverlässige Umsetzung einer Verbindung.

Komfortabler ist die Kommunikation mit Hilfe von RPCs möglich. Diese passen auch gut in das Konzept der Erweiterbarkeit von mosaik, da dieses vorsieht, dass neue Simulatoren in verschiedenen Programmiersprachen geschrieben werden und sich durch Implementierung der SimAPI einbinden lassen. Auf der Odysseus-Seite müsste also lediglich ein ProtocolHandler umgesetzt werden, welcher die Nachrichten der Low-Level API verarbeitet. Da mosaik dabei jedoch immer auf eine Antwort wartet und somit blockiert, würde die Anforderung **A**-6 nicht erfüllt. Zudem könnte auch **A**-3 nicht eingehalten werden, da ein die SimAPI implementierender Simulator direkt beim Start des mosaik-Szenarios gestartet sein muss und sich Odysseus somit nicht jederzeit ein- und ausklinken könnte.

Eine auf AMQP basierend Anbindung wie z. B. RabbitMQ bietet zwar wie beschrieben eine hohe Zuverlässigkeit der Kommunikation, jedoch wird diese zulasten der Geschwindigkeit erreicht. Zudem ist die Konfiguration durch den zu verwendenden Broker aufwendiger, da dieser zusätzlich installiert sein muss.

ZeroMQ bietet demgegenüber eine geringere Latenz, weil die Kommunikation direkt stattfindet und komplett auf einen zwischengeschalteten Server verzichtet wird. Dadurch ist zwar kein Austausch von Nachrichten möglich, wenn nicht beide Kommunikationspartner gleichzeitig verfügbar sind, dieser Fall spielt jedoch für die geplante Anbindung von mosaik und Odysseus keine Rolle. Zudem ist durch den nicht vorhandenen Server auch die Konfiguration deutlich weniger aufwendig. Die verschiedenen Kommunikationsschemata von ZeroMQ bieten für verschieden Anwendungsfälle angepasste Möglichkeiten.

Bei der Verwendung von ZeroMQ ist das Publish-Subscribe-Prinzip am besten geeignet, da es für Datenströme vorgesehen ist. Der mosaik-Sender kann dabei ein Socket öffnen und solange noch kein Subscriber verbunden ist, werden die Daten noch nicht versendet und die Simulation daher nicht blockiert (**A**-6). Der ZeroMQ-Handler von Odysseus kann den Datenstrom jederzeit abonnieren und erhält von diesem Moment an die Daten (**A**-3). Wie beschrieben, kann es bei Verwendung dieses Schemas zum Verlust der ersten übertragenen Pakete kommen. Dies ist jedoch für den geplanten Anwendungsfall vertretbar.

Die beiden interessantesten Kommunikationsmöglichkeiten sind demnach die Anbindung über ZeroMQ oder über RPCs unter Verwendung der SimAPI. Die Verwendung von ZeroMQ mit dem Publish-Subscribe-Prinzip bietet am ehesten einen Datenstrom, der dem Verhalten von echten Sensordaten ähnlich ist, und wird in jedem Fall umgesetzt.

Demgegenüber erfüllt die Umsetzung mit RCPs nicht alle Anforderungen eines typischen Datenstroms. Sie soll aber trotzdem zusätzlich implementiert werden, damit sie in Hinsicht auf die Geschwindigkeit und Zuverlässigkeit verglichen werden kann. Zudem lässt sich durch die Umsetzung von zwei verschiedenen Ansätzen für zukünftige Verwendung sicherstellen, dass je nach Anwendungsfall die gewünschten Eigenschaften erfüllt sind, was den Anforderungen **A**-4 und **A**-5 entspricht.

3.3 ASE-Operator in Odysseus

Der ASE-Algorithmus liegt bereits in einer Java-Implementierung vor und soll als solche in Odysseus integriert werden. An dieser Stelle liegt der Fokus zunächst auf der Verwendung der Implementierung und nicht auf der genauen Funktionsweise. Die Details der theoretischen Funktionsweise erklärte bereits Abschnitt 2.3. Damit der ASE-Algorithmus in Odysseus verwendet werden kann, ist es notwendig einen neuen Operator zu implementieren, dessen Eigenschaften in diesem Abschnitt diskutiert werden. Zunächst wird die Ansteuerung der gegebene Implementierung des ASE-Algorithmus erläutert, danach die Konfiguration des ASE-Operators, sein Datentyp und letztlich die Umsetzung in Odysseus diskutiert.

3.3.1 Ansteuerung des ASE-Algorithmus

Der in Abschnitt 2.3.3 bereits kurz beschriebene Aufbau des bereitgestellten ASE-Algorithmus ist in Abbildung A.4 auf Seite 90 in Form eines Klassendiagramm zu sehen. Nun sollen diese Ausführungen noch weiter vertieft werden, indem erläutert wird, wie die Ansteuerung geschehen kann und in welcher Form der Algorithmus Ergebnisse liefert.

```
1 public EstimationResultSet estimateState(FourWireNetwork network, Inputs
    connectorContainer, double epsilon, int maxIterations)
```

Listing 3.1: EstimateState-Methode des AdaptiveStateEstimators

Zur Ausführung des ASE-Algorithmus genügt es dessen *estimateState()* Methode mit den richtigen Parametern aufzurufen (vgl. Listing 3.1). Das beim Methodenaufruf übergebene *network* gibt die Topologie des Stromnetzes wieder und der *connectorContainer* Parameter beinhaltet die Input-Connectoren. Auf diese beiden Parameter soll später noch genauer eingegangen werden. *Epsilon* und *maxIterations* geben die maximale Veränderung als Abbruchbedingung der Iterationsschleife und die maximale Anzahl an Schritten der Iterationsschleife an.

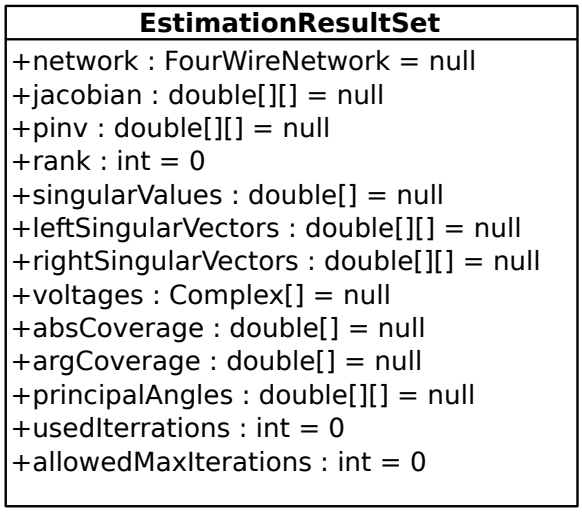
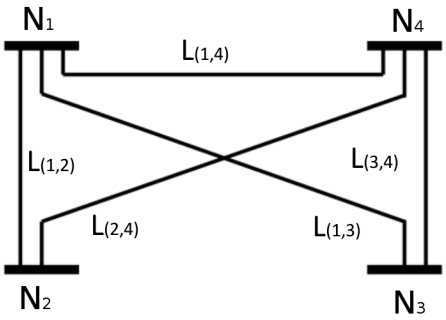


Abbildung 3.7: EstimationResultSet

Der Rückgabewert der Methode ist ein *EstimationResultSet*, welches in Abbildung 3.7 dargestellt ist. Es beinhaltet zum einen die mathematischen Artefakte der Berechnung, wie Jacobi-Matrix (*jacobian*), Pseudoinverse (*pinv*), Rang, Singulärwerte (*singularValues*), linke Singulärvektoren (*leftSingularVectors*) und rechte Singulärvektoren (*rightSingularVectors*). Zum anderen beinhaltet es als Ergebnisse der Ausführung den gefundenen Spannungsvektor (*voltages*) und die Modellabdeckung (*Coverage*) und als Statistiken zur Ausführung die benötigten Iterationen (*usedIterations*).

3.3.1.1 Netztopologie

Die Topologie des Netzes wird als Graph bestehend aus Knoten und Kanten angegeben. Dabei steht die Menge N für alle Knoten und die Menge L für alle Kanten (vgl. Abb. 3.8). Für die Verwendung der Netztopologie im ASE-Operator muss ein *FourWireNetwork* erzeugt werden, welches genau diese Topologie abbildet. Dieses liegt im Paket *com.smartdist.modelling.network* der Implementierung, welches auch im Klassendiagramm in Abbildung A.3 auf Seite 89 zu sehen ist.



$$N = \{N_1, N_2, N_3, N_4\}$$
$$L = \{L_{(1,2)}, L_{(1,3)}, L_{(1,4)}, L_{(2,4)}, L_{(3,4)}\}$$

Abbildung 3.8: Modellierung der Netztopologie

Die Knoten werden durch die Klasse *Node* umgesetzt und besitzen folgende Parameter:

- Eindeutige Identifizierungsnummer (ID)
- Nennspannung

Sie lassen sich also durch eine eindeutige ID zuordnen und beinhalten ihre Nominalspannung. Dies ist die Spannung, welche im normalen Betrieb jeweils zwischen den drei Außenleitern und dem Neutralleiter anliegen sollte. Im Niederspannungsnetz in Deutschland wären dies also 230 V.

Die Leitungen zwischen den Knoten werden durch deren IDs ebenfalls eindeutig bezeichnet, da sich die Bezeichnung $L_{(x,y)}$ aus den IDs der beiden verbundenen Knoten $x, y \in N$ ergibt. Zudem sind sie ungerichtet ($L_{(x,y)} = L_{(y,x)}$) und es gilt $x \neq y$. Jede Leitung besitzt eine Menge an Parametern, über welche sie definiert wird:

- IDs der verbundenen Knoten
- Länge
- Wirkwiderstände (Resistanzen)
- Blindwiderstände (Reaktanzen)
- Ableitwiderstände (Konduktanzen) zwischen den Leitern und zur Erde

- Kapazitive Blindwiderstände (Suszeptanzen) zwischen den Leitern und zur Erde

Die Leitungen werden durch Implementierungen des *EightTerminalElement* Interfaces abgebildet, die jeweils für einen bestimmten Leitungstyp stehen. Die verschiedenen implementierten Typen von Leitungen lauten wie folgt:

Line: Standardleitung mit vorgegebenen üblichen Parametern.

SCLine: Parametrisierbare Version der Standardleitung, bei der als zusätzliche Parameter zur Instanziierung Widerstand, Reaktanz, Nullwiderstand und Nullreaktanz angegeben werden können.

PVC_Orange_Circular_X: Umsetzung der Standardleitung für ein PVC Orange Circular Erdkabel, wobei X für verschiedene Querschnittsflächen von 120, 150 oder 240mm^2 steht und die Länge bei der Instanziierung angegeben wird.

XLPE_SDI_X: Umsetzung der Standardleitung für ein XLPE SDI Erdkabel, wobei X für verschiedene Querschnittsflächen von 70, 120 oder 185 mm^2 steht und die Länge bei der Instanziierung angegeben wird.

Anhand des so angegebenen Netzmodells, kann das FourWireNetwork eine Admittanzmatrix (vgl. Abschnitt 2.2.3) bereitstellen. Diese dient als Repräsentation des Netzes bei der ASE-Berechnung.

3.3.1.2 Connectoren

Beim Methodenaufruf der ASE werden zudem auch die Input-Connectoren übergeben. Diese können von unterschiedlichem Typ sein, die nun genauer erläutert werden sollen. Alle Typen sind unabhängig davon ob sie als Input- oder Output-Connector verwendet werden. Die einzelnen Connectoren beziehen sich jeweils auf eine Phase und müssen somit immer auch die Informationen beinhalten, auf welche Phase sie sich beziehen. Die wichtigsten Arten sind im Klassendiagramm in Abbildung A.5 auf Seite 91 zu sehen und werden in Tabelle 3.2 kurz zusammengefasst.

AbsoluteNodalPhaseCurrent	Dieser Connector steht für die Stromstärke einer Phase.
NodalPhaseCurrentSquared	Dieser Connector steht für die quadrierte Stromstärke einer Phase.
PhaseToEarthVoltage	Dieser Connector steht für die Spannung einer bestimmten Phase gegenüber der Erde.
PhaseToPhaseVoltage	Dieser Connector steht für die Spannung einer bestimmten Phase gegenüber einer anderen Phase. Daher muss zusätzlich zu der eigenen Phase noch die Referenzphase angegeben werden.
PMU	Dieser Connector steht für den Phasenwinkel einer Phase.
SinglePhaseActivePower	Dieser Connector steht für die Leistung einer Phase.
SinglePhaseReactivePower	Dieser Connector steht für die Blindleistung einer Phase.

Tabelle 3.2: Die wichtigsten Arten von Connectoren für ASE

Zur Definition eines Connectors sind somit also immer das Netz, der zugehörige Knoten des Netzes, die Phase und der eigentliche Messwert nötig. Zusätzlich wird für den PhaseToPhaseVoltage-Connector auch die Referenzphase benötigt. Für Input-Connectoren kann zudem die Genauigkeit des Messwerts in Form der Standardabweichung angegeben werden.

3.3.2 Konfiguration

Die Umsetzung der ASE-Berechnung in Odysseus soll durch eine schnelle Änder- und Austauschbarkeit die Verwendung flexibler machen. Wie genau der Operator dabei konfigurierbar sein soll, wird nun erläutert, indem die Anforderungen herausgestellt, diese mit den Gegebenheiten verglichen, zusätzliche Lösungsmöglichkeiten aufgezeigt und abschließend diskutiert werden.

3.3.2.1 Anforderungen an die Konfiguration

Die für die Konfiguration der ASE-Berechnung benötigten Parameter wurden bereits beschrieben und sind in Abbildung 3.9 nochmals zusammengefasst. Darin sind zudem auch die beiden erst später genauer erläuterte Attribute *KVAttribute* (vgl. Abschnitt 3.3.3.2) und *VisualisationType* (vgl. Abschnitt 3.4.1) dargestellt. Die Parameter der ASE-Berechnung lassen sich aufteilen in die allgemeingültige Topologie und die ASE-spezifischen In- und Output-Connectoren. Gemäß der Anforderung **O**-6 soll die Konfiguration möglichst einfach gehalten werden. Daher sollten bekannte und verbreitete Möglichkeiten der Konfiguration verwendet werden.

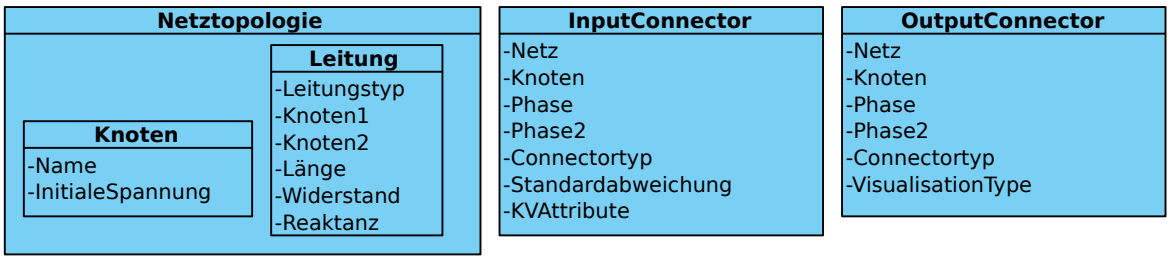


Abbildung 3.9: Parameter des ASE-Algorithmus

Die ASE-spezifischen Daten – also die In- und Output-Connectoren – werden erst für die Anwendung des ASE-Operators erstellt und sind somit in jedem Fall manuell zu definieren. Da die Topologie des Netzes von außen vorgegeben ist, muss sie zum einen ebenfalls manuell konfiguriert werden können. Zudem wäre es interessant, wenn sie sich auch auf möglichst einfache Weise von bestimmten Quellen übernehmen ließe. Dieser Aspekt soll für mosaik in Abschnitt 3.3.2.2 genauer betrachtet werden.

Eine weitere zu diskutierende Frage ist, ob die Netztopologie veränderlich sein soll. Eine Veränderung der Topologie zu ermöglichen, würde die Umsetzung deutlich komplizierter machen, da sie über den Datenstrom konfigurierbar sein müsste. Dies wäre in Odysseus umsetzbar, indem z. B. ein zweiter Port des Operators verwendet wird, welcher Konfigurationsinformationen entgegen nimmt. Jedoch stellt sich die Frage, wo die Konfigurationsbefehle generiert werden sollten. Da in der Realität grundlegende Änderungen an der Netztopologie auch nur äußerst selten vorkommen, soll auf weitere Überlegungen zu diesem Thema verzichtet und die Topologie als unveränderliche Komponente angesehen werden.

Die Output-Connectoren können ebenso als unveränderliche Komponente umgesetzt werden, da theoretisch Output-Connectoren für alle möglichen Ausgaben definiert und anschließend nur die wirklich benötigten Daten herausgefiltert werden können. Durch das Berechnen von später wieder gefilterten Ergebnissen wird zwar zusätzliche Zeit benötigt, jedoch ist diese im Vergleich zu der Dau-

er der ASE-Berechnung zu vernachlässigen. Zudem sei auf die Diskussion des Datentyps in Abschnitt 3.3.3.2 verwiesen, da die Ausgabe als Tupel auch ein festes Schema voraussetzt.

Die einzige Komponente des ASE-Operators, deren Veränderlichkeit wichtig ist, sind die Input-Connectoren. Durch Änderung an ihrer Konfiguration soll es möglich sein z. B. Ausfälle von Messstellen oder Konfigurationsänderungen der Messstellenanordnung zu simulieren. Dazu müssen jedoch bei der initialen Konfiguration des Operators bereits die maximal verwendbaren Input-Connectoren angegeben werden, da jeder zu verwendende Input-Connector konfiguriert werden muss. Diese sollen sich in der Verarbeitung dadurch deaktivieren lassen, dass das entsprechende Datenobjekt im ankommenden Datenstromobjekt *null* ist.

Für den Ablauf des ASE-Operators steht somit fest, dass Netztopologie und Output-Connectoren lediglich bei der Initialisierung des Operators eingelesen werden müssen. Bei der Verarbeitung eines neuen Datenobjekts kann auf die bereits vorliegenden Daten zurückgegriffen werden. Es müssen lediglich bei jeder Verarbeitung aus den ankommenden Messdaten die passenden Input-Connectoren erzeugt werden. Anschließend wird der ASE-Algorithmus aufgerufen und das Ergebnis weitergeleitet.

3.3.2.2 Topologie in mosaik

Wie bereits in Abschnitt 2.5.1.2 beschrieben, beinhaltete die erste Version von mosaik die DSL MoSL zum Beschreiben von Szenarien. Derart austauschbare Szenarien wären auch für die Konfiguration des ASE-Operators sehr interessant. Zum einen würde ein direkter Austausch der kompletten Szenariodefinition zwischen mosaik und Odysseus die Konfiguration des ASE-Operators stark vereinfachen und ggf. doppelt anfallende Konfigurationen überflüssig machen. Zum anderen könnte die Netztopologie für den ASE-Operator – selbst wenn kein direkter Austausch stattfinden sollte – auch in dieser bereits verwendeten oder einer sonstigen DSL beschrieben werden.

Doch die in [Sch14] beschriebene Version von MoSL wird in der zweiten Version von mosaik nicht mehr unterstützt. Stattdessen werden Szenarien zum aktuellen Zeitpunkt in Python definiert. Es läuft zwar die Entwicklung eines Nachfolgers, ein funktionsfähiger Stand ist jedoch noch nicht absehbar. Somit wäre es zu untersuchen, ob ggf. ein Nachfolger von MoSL – nach dessen Fertigstellung – direkt in den ASE-Operator eingebunden werden könnte.

3.3.2.3 Konfiguration in Odysseus

Kontinuierliche Anfragen bestehen in Odysseus generell aus Operatoren. Diese verarbeiten den Datenstrom auf die gewünschte Weise und lassen sich dazu meist konfigurieren. Dafür bietet Odysseus zum einen die Möglichkeit Key-Value-Paare über den *options*-Parameter anzugeben. Zum anderen lassen sich spezielle Parameter verwenden, bei denen der Datentyp vorgegeben ist und die Parameterwerte validiert oder vorverarbeitet werden können. Verfügbare Parameter sind z. B. StringParameter oder IntegerParameter die primitive Datentypen aufnehmen. Beispiele für komplexere Parameter sind der FileParameter, über den Dateien angegeben werden können, oder der ResolvedSDFAttributeParameter, über welchen sich Attribute eines relationalen Eingabetupels angeben lassen, die schon beim Anlegen der Anfrage direkt validiert werden.

Die Parameter können über Annotations ohne großen Aufwand in die logischen Operatoren eingebunden werden. Dabei können zudem Informationen wie der Name, ob der Parameter optional ist oder

ob es sich um eine Liste handelt – also das Angeben von mehreren Werten möglich ist – festgelegt werden.

3.3.2.4 Diskussion

Wie bereits beschrieben, lässt sich die Konfiguration des ASE-Operators aufteilen in die allgemeine Topologie und die ASE-spezifischen In- und Output-Connectoren. Die ASE-spezifischen Parameter sollen dabei auf jeden Fall manuell konfiguriert werden können, da sie nicht aus anderen Daten abgeleitet werden könnten.

Die Topologie des untersuchten Netzes hängt jedoch mit der Datenquelle zusammen. Bei der Verwendung von Daten aus mosaik zur ASE-Berechnung wäre die automatische Verwendung der Netztopologie aus mosaik eine elegante Lösung. Dies wäre z. B. über eine DSL zur Beschreibung der Szenarien realisierbar. Jedoch wird MoSL von der aktuellen mosaik Version nicht mehr unterstützt und Szenarien werden in Python definiert. Zudem muss die Konfiguration der Topologie in jedem Fall auch manuell möglich sein, da die Daten aus mosaik auch durch die Daten einer realen Quelle ersetzbar sein sollen. Daher muss der ASE-Operator manuell konfigurierbar sein.

Dafür wäre es auch möglich eine spezielle DSL zu entwerfen, die vergleichbar zu MoSL ist. Dies würde den Vorteil mit sich bringen, dass sich die Eingaben besser validieren ließen. Gerade da Knoten über ihren Namen referenziert werden, könnten so z. B. Tippfehler leicht gefunden werden und die Validität der Topologie direkt überprüft werden. Ohne den Zusammenhang mit der mosaik Szenariodefinition wäre es jedoch im Vergleich zum Nutzen sehr viel Aufwand. Daher wird davon Abstand genommen und darauf gewartet, dass eine neue DSL für mosaik bereitsteht.

Die Konfiguration könnte durch Listen-Parameter umgesetzt werden, jedoch machen diese bei größeren Netzen die PQL-Anfrage schnell unübersichtlich. Daher wird es vorgezogen statt der Definition direkt in der PQL-Anfrage auf eine Datei zu referenzieren. Als Datentyp dafür bietet sich JSON an, da dieses allgemein häufig zur Konfiguration und auch im Umfeld von Odysseus und mosaik verwendetet wird. Der folgende Abschnitt beschreibt den Aufbau der JSON-Konfigurationsdatei.

3.3.2.5 JSON-Konfigurationsdatei

In der Konfigurationsdatei sollen die in Abbildung 3.9 beschriebenen Parameter definierbar sein. Dazu ist eine JSON-Datei mit folgenden vier Hauptattributen und ihren Unterattributen zu erstellen:

Edges: Zunächst wird der **type** der Kante definiert, wobei sich die in Abschnitt 3.3.1.1 beschriebenen Typen verwenden lassen. Zur richtigen Zuordnung müssen die beiden durch die Kante verbundenen Knoten als **node1** und **node2** angegeben werden. Zudem werden physikalische Eigenschaften in Form der Parameter **length**, **resistance** und **reactance** angegeben.

Nodes: Jeder Knoten muss die Parameter **name** und **initialVoltage** beinhalten. Über den Namen wird der Knoten in den anderen Attributen referenziert.

InputConnectors: Als **type** können die in Abschnitt 3.3.1.2 vorgestellten Typen verwendet werden. Über den Parameter **node** wird der Connector einem Knoten zugeordnet. Zudem muss zumindest die Phase des Connectors angegeben werden (**ownPhase**) und bei einigen Typen auch die Referenzphase (**otherPhase**). Die Qualität des Messwertes an dem InputConnector kann über den

Parameter **stdDev** angegeben werden. Für die Verarbeitung von Key-Value-Objekten wird zudem der Parameter **attribute** benötigt, damit der entsprechende Wert aus dem Element verwendet wird.

OutputConnectors: Als **type** können die in Abschnitt 3.3.1.2 vorgestellten Typen verwendet werden. Über den Parameter **node** wird der Connector einem Knoten zugeordnet. Zudem muss zumindest die Phase des Connectors angegeben werden (**ownPhase**) und bei einigen Typen auch die Referenzphase (**otherPhase**). Optional kann auch über den Parameter **visualisationType** angegeben werden, ob die Ergebnisse des Connectors visualisiert werden sollen.

3.3.3 Datentyp des ASE-Operators

Für den Transport und die Verarbeitung bietet Odysseus verschiedene Datentypen an, die nun in Hinblick auf die Verwendung für den ASE-Operator erläutert und diskutiert werden.

3.3.3.1 Datentypen in Odysseus

Der vorherrschende Datentyp in Odysseus sind relationale Tupel, die aus der Datenbankdomäne stammen. Dort ist ein Tupel eine einzelne Reihe der Antwort auf eine Datenbankanfrage. In einem DSMS besteht ein Datenstrom nun aus einzelnen Tupeln, die nacheinander übertragen werden und dabei alle einem fest definierten Schema folgen, wie in Abbildung 3.10 dargestellt. Die meisten vorhandenen Operationen in Odysseus unterstützen Tupel und ermöglichen daher vielfältige Möglichkeiten zur Verarbeitung.

Node1_V	Node1_Va	Node2_P	Node2_Q	Node3_P	Node3_Q
230,1	0,02	1784,0	0,1	110,1	0,2
230,3	0,03	2734,4	0,3	100,9	0,1

Abbildung 3.10: Relationales Tuple-Schema

Odysseus unterstützt zudem Key-Value-Objekte. Diese bieten eine höhere Flexibilität als relationale Tupel, da sie kein festes Schema besitzen. Die Key-Value-Objekte können z. B. aus JSON Objekten erstellt werden. JSON wird auch von mosaik verwendet, wie im Anhang in Abbildung A.3 auf Seite 95 in Form einer Ausgabe von mosaik zu sehen ist. Zum aktuellen Zeitpunkt unterstützen jedoch nur relativ wenige Operatoren in Odysseus Key-Value-Objekte.

Um flexibel mit beiden Typen arbeiten zu können, bietet Odysseus auch Operatoren zur Umwandlung von Tupeln in Key-Value-Objekte und entgegengesetzt. Durch die Umwandlung gehen jedoch auch Vorteile der Datentypen verloren. So besitzen Key-Value-Objekte, die von dem selben Operator aus Tupeln erzeugt wurden, quasi das für die Tupel definierte feste Schema. Bei der Umwandlung von Key-Value-Objekten in Tupel muss zudem ein festes Schema für die Umwandlung angegeben werden und es können somit ggf. Daten verloren gehen.

3.3.3.2 Diskussion

Die Daten aus mosaik kommen als JSON Objekte an und liegen somit in jedem Fall zunächst als Key-Value-Objekte vor. Es soll nun diskutiert werden, ob der ASE-Operator direkt auf diesen arbeiten soll, oder die Umwandlung in Tupel zu präferieren ist.

Für die Dateneingabe in den ASE-Operator weist ein fest vorgegebenes Schema Vorteile auf. Insbesondere die für Tupel verfügbaren Operatoren in Odysseus bieten sehr viele Möglichkeiten der Datenaufbereitung vor der Verarbeitung, wie z. B. Projektionen, Umbenennung oder Aggregationen. Somit wird für die optimale Einbettung in die vorhandenen Funktionen von Odysseus auch für die ankommenden Daten des ASE-Operators das relationale Tupel verwendet. Jedoch hat auch ein Key-Value-ASE-Operator ohne festes Schema seine Vorteile. Zum einen müssen die Daten nicht erst in Tupel umgewandelt werden. Zum anderen sind Key-Value-Objekte beim Fehlen von Messwerten für einzelne Knoten flexibler.

Da sich die grundsätzliche Funktionalität des Operators für Tupel und Key-Value-Objekte nicht stark unterscheidet, bietet sich die Umsetzung beider Varianten an, um für verschiedene Anwendungsfälle die optimale Variante zur Verfügung zu stellen. Da Odysseus an den meisten Stellen möglichst generisch programmiert ist, sind solch flexible Operatoren häufig vorhanden. Damit die Key-Value-Variante die Werte dem richtigen Input-Connectoren zuordnen kann, muss der entsprechende Key in der Konfigurationsdatei des ASE-Operators angegeben werden (vgl. Abschnitt 3.3.2.5).

Die nächste Frage, die sich stellt, ist die nach den Ausgabedaten des Operators. Wie in Abschnitt 3.3.2.1 bereits erläutert, sollen die Netztopologie und die Output-Connectoren unveränderlich sein. Die Ausgabe des ASE-Operators besitzt somit ein festes Schema und es bietet sich an als Ausgabe des ASE-Operators relationale Tupel zu verwenden. Dies ist auch praktisch für die Visualisierung der Daten in Odysseus, da die Visualisierungskomponenten bisher ausschließlich Tupel unterstützen.

3.3.4 Umsetzung des Operators

Wie bereits in Abschnitt 2.4.1 eingeleitet wurde, werden Anfragen in Odysseus zunächst aus logischen Operatoren gebildet. Diese werden anschließend über Transformationsregeln in physische Operatoren umgewandelt. Um also einen Operator in Odysseus bereitzustellen, muss sich dieser in die gegebene Struktur eingliedern.

Dadurch dass Odysseus auf OSGi aufbaut, können Funktionen sehr flexibel eingebunden werden. Somit muss der Anwender jeweils nur die Plug-Ins laden, welche er auch wirklich verwenden möchte. Um dieses Prinzip zu unterstützen, soll auch die ASE-Berechnung in dem eigenem Plug-In *de.uniol.inf.is.odysseus.energy.ase* bereitgestellt werden. Für die Verwendung muss der Anwender anschließend das Plug-In einbinden und aktivieren. Eine Übersicht über das beschriebene Paket wird im Klassendiagramm in Abbildung 3.13 gegeben.

Der logische Operator *ASEAO* liest entweder über den *FileParameter* eine lokale oder über den *HTTPStringParameter* eine per URL verfügbare JSON-Konfigurationsdatei, deren Struktur im vorangehenden Abschnitt beschrieben wurde, für die ASE-Berechnung ein. Im Sequenzdiagramm in Abbildung 3.11 wird dieser Vorgang als Erzeugen des ASEAO durch den Benutzer und dem Übertragen der Konfiguration dargestellt. Mit Hilfe des *KeyValueObjectDataHandler* wird aus der JSON-Datei ein Key-Value-Objekt erzeugt aus welchem sich anschließend die definierte Netztopologie in ein *FourWireNetwork* umwandeln lässt (*initNetwork()*). Zudem werden die Input- und Output-Connectoren

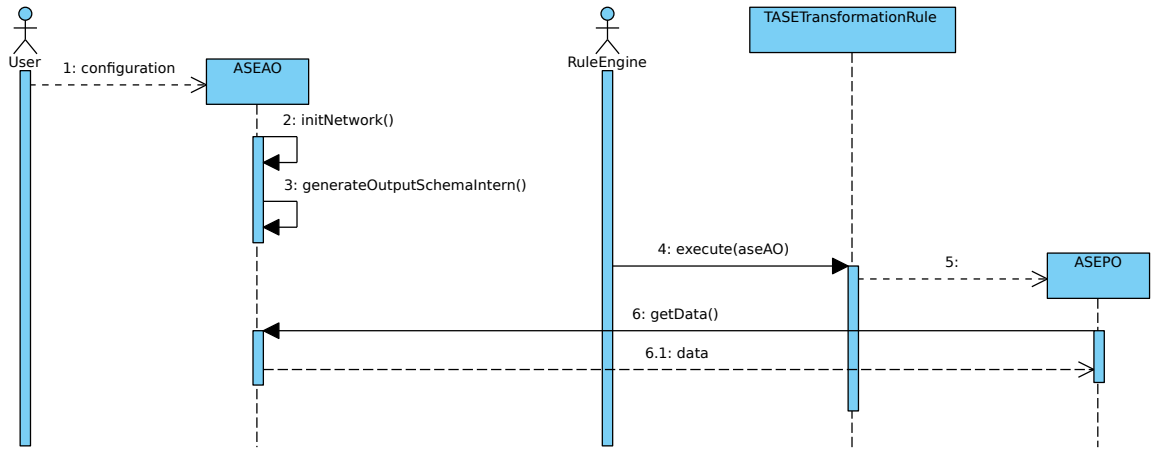


Abbildung 3.11: Sequenzdiagramm der Transformation des logischen zum physischen Operator

aus der Konfigurationsdatei heraus erstellt und anhand der Output-Connectoren das Output-Schema des Operators generiert.

Aus dem logischen Operator wird schließlich durch die *TASETransformationRule* der physische Operator *ASEPO* instanziiert. Dabei wird auch überprüft, ob es sich bei den Eingangsdaten des Operators um KeyValue Objekte handelt, da in diesem Fall ein *KeyValueASEPO* erstellt werden würde. Diese beiden Operatoren leiten sich vom *AbstractASEPO* ab, welcher die allgemeine Funktionalität beinhaltet und wiederum von Odysseus *AbstractPipe* abgeleitet ist. Der Ablauf des Operators wurde umgesetzt wie in Abschnitt 3.3.2.1 bereits beschrieben und in Abbildung 3.12 in Form eines Sequenzdiagramms zusammengefasst.

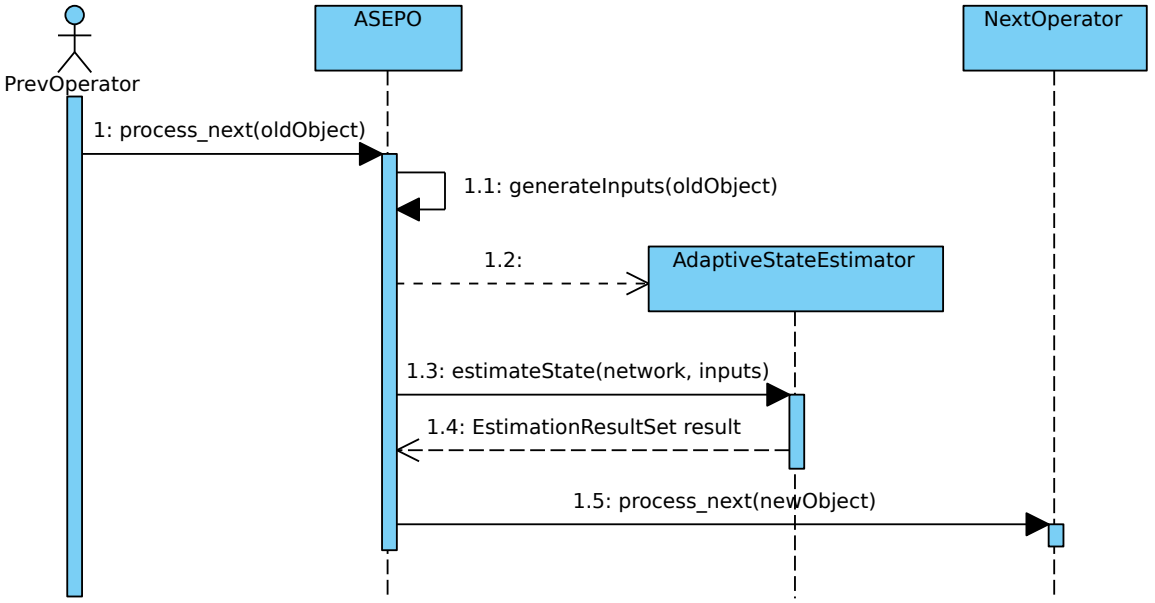


Abbildung 3.12: Sequenzdiagramm des ASE-Operators

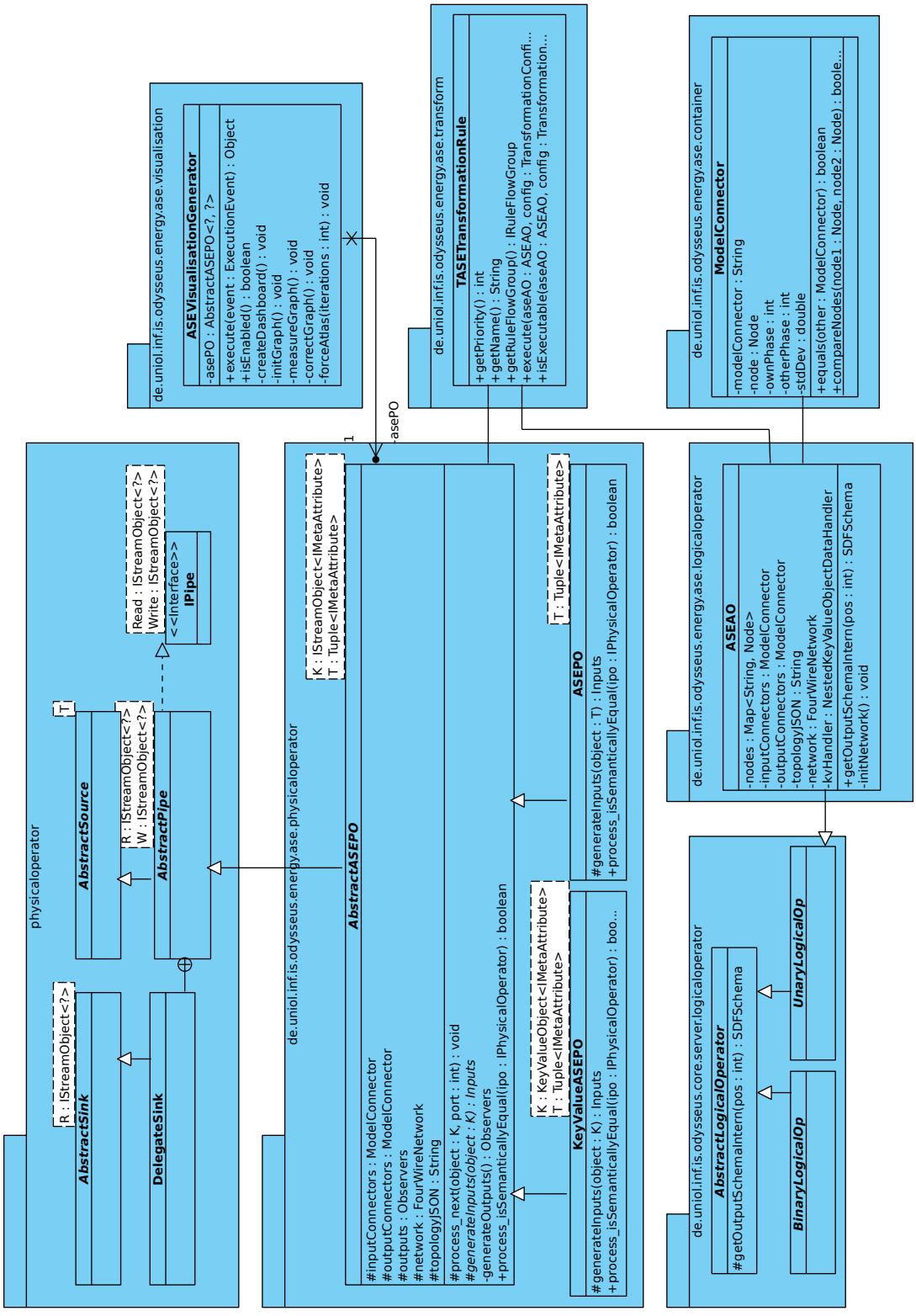


Abbildung 3.13: ASE-Package – de.uniol.inf.is.odysseus.energy.ase

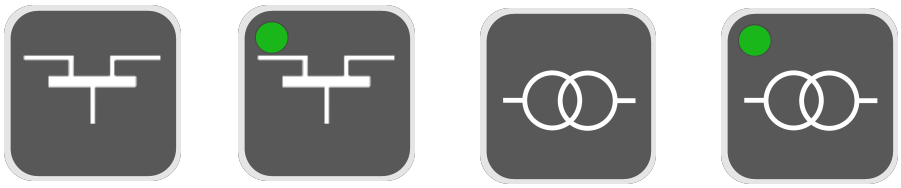
Um mehrfach vorhandene Operatoren zu entdecken, welche auch genau gleich konfiguriert sind und somit nicht mehrfach angelegt werden müssen, bieten Operatoren in Odysseus die Methode *isSemanticallyEqual* an. Dieser wird ein Operator übergeben und anschließend überprüft, ob die beiden identische konfiguriert sind. Im Falle des ASEPO ist der Vergleich über das FourWireNetwork dabei sehr aufwendig, da alle Knoten und Kanten miteinander verglichen werden müssen. Diese sind jedoch nicht sortiert und können teilweise unterschiedliche IDs besitzen, selbst wenn sie aus der gleichen Konfigurationsdatei generiert wurden. Daher ist der Vergleich des JSON-String, der zur Konfiguration des Netzes eingelesen wurde, die deutlich performantere Lösung. Auch wenn durch die Verwendung dieses Ansatzes nicht erkannt wird, wenn Elemente in der Definition in einer anderen Reihenfolge beschrieben werden, soll sie zum Einsatz kommen.

3.4 Visualisierung

Die Ergebnisse der ASE-Berechnungen sollen nach Anforderungen **O**-3 und **O**-4 auch in Odysseus visualisiert werden können. Odysseus bietet als Möglichkeit zur Visualisierung der Datenströme bereits sogenannte Dashboards an. Diese lassen sich durch XML-Dateien definieren und können z. B. Diagramme, Datenwerte oder auch von den Datenwerten abhängige Bilder beinhalten. Für die Darstellung des dem ASE-Operator zugrunde liegende Netzes und die Ergebnisse seiner Berechnungen sollen die Möglichkeiten der Dashboards verwendet werden. Im Kommenden wird darauf eingegangen welche Informationen visualisiert werden, wie die Generierung der Dashboards durchgeführt werden kann und wie das Energienetz darin dargestellt werden soll.

3.4.1 Inhalt

In der Visualisierung soll das komplette Netz bestehend aus Knoten und Leitungen dargestellt werden. Die relevanten Netzelemente für die ASE sind hauptsächlich die normalen Netzanschlussknoten. Da sich das betrachtete Netz lediglich in einer Spannungsebene befindet, kann als einziges Element der Transformator als Referenzknoten und Übergang in eine höhere Spannungsebene hervorgehoben werden. Somit wird bei der Visualisierung lediglich durch verschiedene Icons zwischen Transformatoren und Netzanschlussknoten unterschieden (vgl. Abb. 3.14).



(a) Netzanschlussknoten ohne Input | *(b) Netzanschlussknoten mit Input* | *(c) Transformator ohne Input* | *(d) Transformator mit Input*

Abbildung 3.14: Icons für die Visualisierung

Zusätzlich zu der Netztopologie soll ersichtlich sein zu welchen Knoten aktuell Input-Connectoren aktiv sind. Dazu wird ein grüner Kreis in das ansonsten graue Icon eingefügt (vgl. Abb. 3.14). Eine Anzeige der genauen Werte am Input-Connector wären zwar auch interessant, jedoch werden diese aus Gründen der Übersichtlichkeit nicht realisiert. Denn gerade in Fällen, in denen mehrere

Input-Connectoren an einem Knoten vorhanden sind, ist eine übersichtliche Darstellung nicht mehr möglich.

Wichtig ist auch die Visualisierung der Output-Connectoren, deren Werte in einem Diagramm über die Zeit dargestellt werden. In dieser Darstellung wird der berechnete Wert zudem gegen den bereits von mosaik gesendeten Wert aufgetragen. Auf diese Weise lässt sich schnell die Qualität der ASE-Ergebnisse erkennen. Für welche Output-Connectoren welche Art von Diagramm angezeigt werden soll, kann in der Konfigurationsdatei über den Parameter *visualisationType* angegeben werden (vgl. Abschnitt 3.3.2.5). Dabei können *voltage* oder *coverage* angegeben werden, um die berechnete Spannung mit der tatsächlichen zu vergleichen oder die Modellabdeckung über die Zeit zu visualisieren.

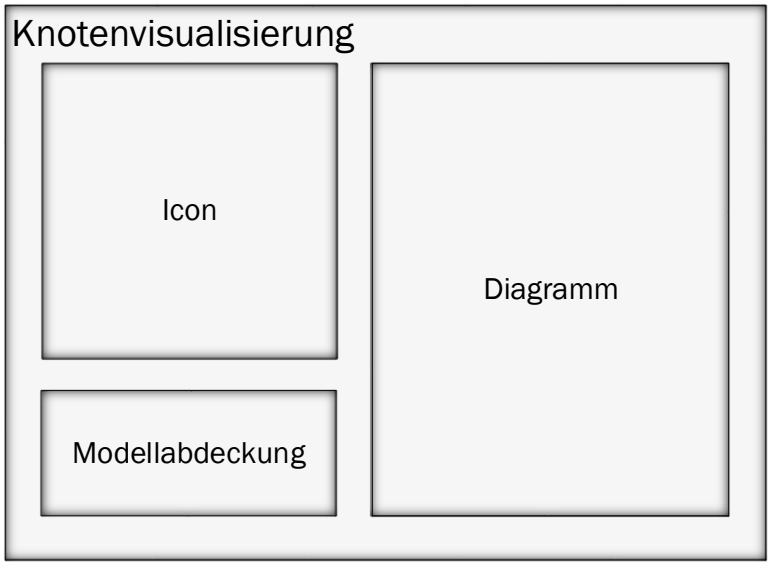


Abbildung 3.15: Komponenten der Knotenvisualisierung

Als letztes Element der Visualisierung wird die Modellabdeckung dargestellt. Zum einen ist dabei der exakte Wert zu sehen und zum anderen wird ein farbiges Icon angezeigt, welches in vier Stufen die Modellabdeckung visualisiert (vgl. Tab. 3.3). Dadurch lässt sich schnell ablesen an welchen Knoten die Werte nicht mehr genau bestimmt sind. Alle beschriebenen Elemente der Knotenvisualisierung werden in Abbildung 3.15 noch einmal zusammengefasst.

Modellabdeckung	Farbe
0 - 5	Grün
5 - 20	Grau
20 - 40	Orange
40 - 90	Rot

Tabelle 3.3: Farbliche Visualisierung der Modellabdeckung

3.4.2 Generierung

Da das manuelle Erstellen von Dashboards über den Editor oder die XML-Definition selbst für Netze mit wenigen Knoten bereits sehr aufwendig ist, soll die Generierung der Dashboards automatisch ausgeführt werden. Dashboards können in Odysseus aus mehreren Teilen (*DashboardParts*) zusammengesetzt werden. Die Struktur der gegebenen Implementierung ist im Klassendiagramm in Abbildung 3.16 dargestellt. Dabei ist zunächst vom Dashboard ausgehend zu beachten, dass dieses aus verschiedenen *DashboardParts* besteht. Diese werden über *DashboardPartPlaycements* und die dazugehörigen Container mit dem Dashboard verbunden. Zudem gibt es zwei *XMLHandler*, die Dashboards und DashboardParts in XML-Dateien abspeichern und auch aus diesen auslesen und instanziieren können. Für die Generierung der ASE-Visualisierung müssen somit zunächst die DashboardParts erzeugt und anschließend zu einem Dashboard zusammengefügt werden, damit schließlich unter Verwendung der XMLHandler die nötigen XML-Definitionen gespeichert werden können.

Eine automatische Durchführung bei jedem Erzeugen des Operators wäre zwar möglich, würde diese jedoch unter Umständen stark verzögern (vgl. Abschnitt 3.4.3) und würde zudem dem eigentlichen Sinn eines Operators widersprechen, der für die kontinuierliche Verarbeitung eines Datenstroms vorgesehen ist und nicht für das einmalige Ausführen einer Aufgabe. Daher soll ein Dashboard nur bei Bedarf auf expliziten Wunsch des Anwenders erstellt werden.

Da Informationen, wie z. B. das Schema und die Attributnamen, benötigt werden, die im Operator verwaltet werden, wäre eine Generierung aus dem Operator heraus wünschenswert. Somit bietet sich die grafische Darstellung des Anfrageplans in Odysseus als Ausgangspunkt an. Dieser beinhaltet die in einer Anfrage enthaltenen Operatoren und zeigt die Datenflüsse zwischen diesen an (siehe z. B. Abb. A.7 auf Seite 99). Da die Oberfläche von Odysseus auf Eclipse PDE[2] aufbaut, lassen sich an dieser Stelle auch sehr flexibel Anpassungen vornehmen.

Zusätzlich zu dieser Generierung direkt in Odysseus könnte auch über eine Umsetzung ohne Odysseus nachgedacht werden, bei welcher die ASE-Konfigurationsdatei verwendet wird. Jedoch müssten zu dieser noch zusätzliche Informationen angegeben werden, die z. B. das Schema des Operators betreffen. Daher soll im Rahmen dieser Arbeit der Fokus auf der Generierung aus dem Anfrageplan liegen.

3.4.3 Layouting

Das automatische Generieren der Visualisierung erfordert auch eine automatische Ausrichtung (Layouting) der Knoten und Kanten der Netztopologie. Im Zusammenhang mit mosaik wurden bereits verschiedene Möglichkeiten des automatischen Layoutings diskutiert. Für eine aktuell laufende Entwicklung eines graphischen Editors für mosaik Szenarien wird letztlich der sogenannte ForceAtlas2 Algorithmus [JVHB14] verwendet. Dieser soll auch für die Visualisierung in Odysseus verwendet werden. Da das Layouting in dieser Arbeit nur eine Nebenrolle spielt, soll diese Auswahl im Folgenden nur kurz erläutert werden. Für genauere Informationen sei auf die Diskussion der verschiedenen Möglichkeiten in [Gü14] und eine detailliertere Beschreibung des ForceAtlas2 Algorithmus in [JVHB14] verwiesen.

[2] `https://eclipse.org/pde/`

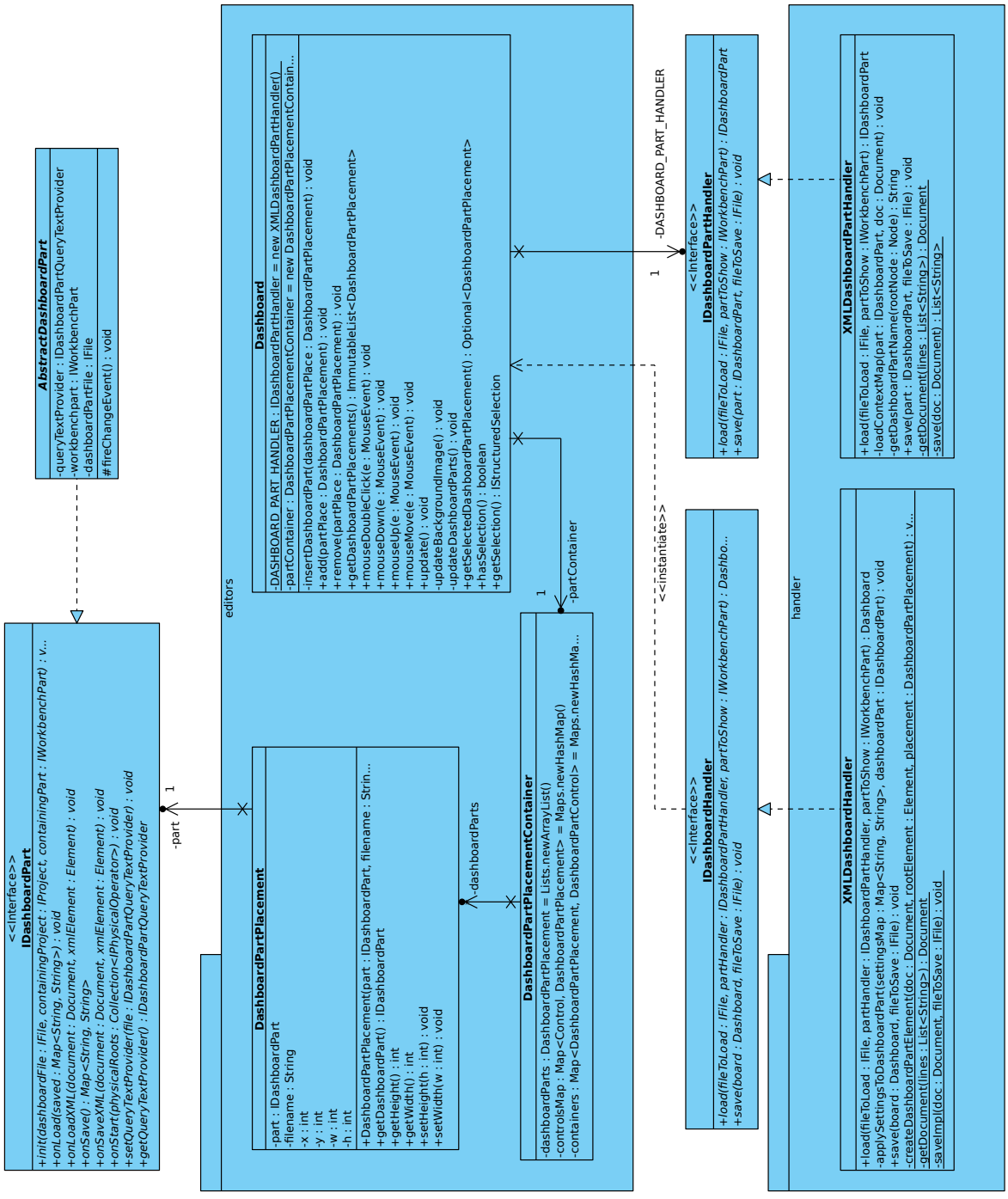


Abbildung 3.16: Klassendiagramm der Dashboards in Odysseus

Der ForceAtlas2 Algorithmus arbeitet kräftebasierend auf einem Graph der Netztopologie. Dies bedeutet, dass ihm ein physikalisches Modell von anziehenden und abstoßenden Kräften zugrunde liegt. Dabei stoßen sich die Knoten voneinander ab und jede Kante hält jeweils zwei Knoten durch ihre Anziehungskräfte beieinander. Er beinhaltet dabei keine komplett neuen Konzepte, sondern verbindet viele bereits vorhandene zu einem leicht anwendbaren Paket [Gü14], [JVHB14].

Das Verhalten des Algorithmus kann über die Angabe einiger Parameter beeinflusst werden. Die wichtigsten sollen nun vorgestellt werden und auf die Rolle eingegangen werden, welche sie allgemein und für die Umsetzung der Visualisierung in Odysseus spielen. Genauere Angaben zu ihrer Funktion sind in [JVHB14, S. 3ff.] zu finden.

LinLog mode: Dies ist ein spezieller Modus des ForceAtlas2 Algorithmus, der zu einem stärkeren Clustering des Graphen führt und somit gerade für die Darstellung von Energienetzen interessant ist und aktiviert wird.

Gravity: Dieser Parameter beschreibt eine Kraft, welche die Knoten zur Mitte des Graphen zieht, damit sie durch die Abstoßung nicht zu weit nach außen getrieben werden. Um ein übersichtliches Ergebnis zu erhalten wird ein Wert von 0,7 gewählt.

Strong gravity mode: Dieser Modus führt eine weitere Kraft ein, welche besonders weit vom Zentrum entfernte Knoten anzieht und somit das Ergebnis stark beeinflusst, indem der Graph im Zentrum komprimiert wird. Für die Verwendung in der ASE-Visualisierung wird dieser Modus deaktiviert.

Edge weight: Das Gewichtung der Kanten steht für die Anziehungskräfte, die diese auf die anliegenden Knoten ausüben. Für die Umsetzung der Visualisierung wird ein Wert von 0 verwendet und somit die Anziehungskraft der Kanten vernachlässigt. Dadurch werden die Abstoßungskräfte der Knoten betont und ein ausgeglicheneres Ergebnis erzielt.

Adjust size: Durch die Aktivierung dieser Option wird dem Algorithmus erlaubt die Größe der Knoten zu ändern. Da die Knoten in der Visualisierung in Odysseus durch Bilder repräsentiert werden, die eine bestimmte Größe besitzen, muss diese Option deaktiviert sein. Ansonsten überschneiden sich die Bilder der verschiedenen Knoten.

Iterationen: Die Anzahl an Durchläufen des Algorithmus, bis das Ergebnis für die Visualisierung verwendet wird. Je häufiger der Algorithmus ausgeführt wird, desto besser ist das Resultat. Um Überschneidungen der Kanten zu vermeiden, sollte er je nach Größe des Netzes etwa 25.000 bis 100.000 mal ausgeführt werden.

Der ForceAtlas2 Algorithmus ist ausgelegt für das kontinuierliche Layouting von bis zu 100.000 Knoten. Das Ergebnis ist dabei abhängig vom initialen Zustand. Zudem ist der Prozess nicht deterministisch und kann somit in lokalen Minima stecken bleiben, was zu einem nicht optimalen Ergebnis führen kann. Der Algorithmus eignet sich gut für die Visualisierung des Energienetzes, da sogenannte Communities nah beieinander liegend dargestellt werden. Er ist Teil der Visualisierungssoftware Gephi, die in Java geschrieben und frei verfügbar ist [Gü14], [JVHB14].

3.5 Zusammenfassung

In diesem Kapitel wurden zunächst die Anforderungen an die Umsetzung der ASE-Berechnung in Odysseus erläutert. Diese wurden erst über die Vision motiviert, tabellarisch zusammengefasst und zudem wurde das zu entwickelnde System in den Kontext eingeordnet.

Anschließend wurde die Anbindung von mosaik an Odysseus entworfen. Dazu wurden als mögliche Kommunikationstechnologien Sockets, RCPs, RabbitMQ und ZeroMQ vorgestellt. Für die Umsetzung wurden sowohl ZeroMQ als auch die Implementierung der SimAPI von mosaik mit Anbin-

dung über RCPs und Sockets ausgewählt, um sowohl eine sehr offene datenstromorientierte, als auch eine engere und evtl. zuverlässigere Anbindung zu haben.

Der ASE-Operator wurde beschrieben, indem die Ansteuerung des ASE-Algorithmus vorgestellt und die dabei zu konfigurierenden Parameter herausgearbeitet wurden. Darauf aufbauend wurde diskutiert, auf welche Art und Weise der Operator konfiguriert werden soll, mit dem Ergebnis, dass sowohl für die allgemeine Topologie als auch für die ASE-spezifischen Connectoren eine JSON-Datei eingelesen wird. Für den Datentyp des Operators wurde entschieden, dass er seine Ausgaben als relationale Tupel erzeugen wird und als Eingabe mit relationalen Tupeln und Key-Value-Objekten umgehen können soll. Der Fokus liegt hierbei jedoch aufgrund der besseren Unterstützung in anderen Operatoren auf den relationalen Tupeln. Schließlich wurde auch beschrieben, wie der Operator in Odysseus eingebunden wird.

Als letzter Punkt wurde auf die Visualisierung der Ergebnisse in Odysseus eingegangen. Für diese sollen die sogenannten Dashboards und DashboardParts in Odysseus verwendet werden, die sich über XML-Dateien definieren lassen. Sie sollen sich dabei programmatisch erzeugen lassen und die komplette Topologie des Netzes anzeigen. Zudem sollen die wichtigsten Ergebnisse der ASE-Berechnung und auch der Vergleich zwischen berechneten und von mosaik gelieferten Werten visualisiert werden können. Zum Layouting der Netzknoten wird der ForceAtlas2 Algorithmus verwendet.

4 Implementierung

Nach der Beschreibung des Entwurfs im vorangehenden Kapitel, folgt nun die Beschreibung der Umsetzung. Dabei geht es um die Anbindung von mosaik an Odysseus, den ASE-Operator in Odysseus und zuletzt die Visualisierung.

4.1 Anbindung von mosaik an Odysseus

Als Grundlage für die Umsetzung des ASE-Algorithmus in Odysseus war zunächst die Anbindung von mosaik an Odysseus zu implementieren, da die Daten aus mosaik für die Berechnungen nötig sind. Die Umsetzung wurde dabei entsprechend der in Abschnitt 3.2 beschriebenen Entwurfsentscheidungen durchgeführt. Die Ausführungen sind unterteilt in Beschreibungen der in mosaik vorliegenden Simulationsdaten, der aktuellen Version der SimAPI, der beiden Implementierungen in Python sowie Java und schließlich des Einbindens der Simulatoren in mosaik und Odysseus.

4.1.1 Simulationsdaten

Die in mosaik anfallenden und zu versendenden Simulationsdaten liegen dort als JSON-Objekte vor. Für ein Beispielszenario ist in Listing A.3 auf Seite 95 im Anhang zu sehen, wie diese Daten aussehen. Der Inhalt ist dabei teilweise abhängig von der Art des Simulators. Die folgenden Erläuterungen beziehen sich auf den PyPower[1]-Simulator, der im Beispielszenario zur Simulation des Energienetzes Verwendung findet.

Zunächst werden die Daten über die ID des Simulators definiert. Darauf folgt eines der unten aufgelisteten Abkürzungen für die Parameter des Netzes, der Name des Simulators und zuletzt der Name des Netzknotens. Die beschriebenen Elemente sind dabei durch Punkte unterteilt. Ein Key-Value-Paar kann dabei bspw. folgendermaßen aussehen:
mosaik.Vm.PyPower-0.0-node1=229.877

Folgende Typen von Daten kommen vor:

P: Die Leistung in Watt [W]. Kommt auch mit den Endungen *"_out"*, *"_in"*, *"_to"* oder *"_from"* zur genaueren Definition vor.

Q: Die Blindleistung in Var [VAr]. Kommt auch mit den Endungen *"_out"*, *"_in"*, *"_to"* oder *"_from"* zur genaueren Definition vor.

Vm: Die nominelle Spannung in Volt [V].

Vl: Die tatsächliche momentane Spannung in Volt [V].

Va: Der Spannungswinkel (*voltage angle*) in Grad [°].

[1] `https://pypi.python.org/pypi/PYPOWER`

4.1.2 Mosaik API

Um an die zuvor beschriebenen Daten in mosaik zu gelangen, sind Simulatoren umzusetzen, welche die mosaik API implementieren. Diese wurde bereits in Abschnitt 2.5.1.1 kurz beschrieben und soll nun etwas detaillierter anhand der aktuellen Version der API für mosaik 2 erläutert werden. Folgende Methoden und Attribute sind darin für Simulatoren beschrieben [mos14]:

meta: Das Meta-Attribut beinhaltet allgemeine Informationen zum Simulator, wie in Listing 4.1 dargestellt ist. Dabei ist die zu implementierende API-Version anzugeben. Falls die Hauptversion *(x)* nicht mit der verwendeten mosaik-Version übereinstimmt, wird die Simulation nicht gestartet. Die Liste *models* kann verschiedene Modelle beinhalten. Diese beinhalten jeweils wiederum eine Liste von *params*, die bei der Instanziierung des Modells übergeben werden und eine Liste von *attrs*, auf welche sowohl schreibend als auch lesend zugegriffen werden kann.

init(sid, sim_params): Diese Methode initialisiert den Simulator mit einer ID. Zudem lassen sich zusätzliche Parameter übergeben. Als Antwort wird das *meta* Attribut zurückgegeben.

create(num, model, model_params): Erzeugt Instanzen des dem Simulator zugrunde liegenden Modells. Übergeben werden dabei *num* als Anzahl der zu erstellenden Modelle, *model* als der Name des zu erzeugenden Modells, welches in *meta['models']* existieren muss, und schließlich *model_params*, das die Werte der im *meta* Attribut definierten Parameter enthält.

Eine Liste der erzeugten Modellinstanzen bildet den Rückgabewert. Diese enthält eindeutige IDs für jede Instanz und optional auch vorhandene Beziehungen zwischen unterschiedlichen Instanzen desselben Simulators.

step(time, inputs): Führt den nächsten Simulationsschritt durch. Übergeben wird dabei die aktuelle Zeit *time* und die Werte aller verbundenen Simulatoren im *inputs* Parameter. Die neue Simulationszeit, die $time + step\_size$ entspricht, wird zurückgegeben.

get_data(outputs): Gibt die Daten der gewünschten *outputs* zurück.

```
1  {   "api_version": "x.y.z",
2      "models": {
3          "ModelName": {
4              "public": true|false,
5              "params": ["param_1", ...],
6              "attrs": ["attr_1", ...] },
7          ...
8      },
9      "extra_methods": ["do_something", ...]
10 }
```

Listing 4.1: Aufbau des meta Attributs

Wie bereits in Abschnitt 3.2.2.2 beschrieben, lässt sich die mosaik API als High-Level oder Low-Level API verwenden. Erstere ist bisher für Python und Java verfügbar und ermöglicht das direkte Aufrufen der beschriebenen Methoden, während bei der zweiten Variante die Methodenaufrufe als JSON Objekte über Sockets versendet werden.

4.1.3 Implementierung in Python (mosaik-zmq)

Als erste Methode der Anbindung von mosaik und Odysseus wird die Implementierung eines Simulators in Python beschrieben. Da mosaik ebenfalls in Python programmiert ist, lässt sich dieser Simulator direkt einbinden. Er versendet alle Daten über ZeroMQ, damit sie in Odysseus – oder auch in beliebigen anderen Anwendungen – empfangen werden können. Daher ist sein Name *mosaik-zmq*.

Die Implementierung des Simulators ist in Abbildung A.1 auf Seite 92 zu sehen. Als meta Parameter dienen zunächst die Host-Adresse, der Port und der Socket-Typ, damit diese beim Instanziieren des Simulators angegeben werden können. Ein Aufruf der init-Methode setzt standardmäßig die Schrittgröße der Simulation und die Dauer der Ausführung. Beim Erstellen des Simulators in der create-Methode werden die Parameter auf ihre Korrektheit überprüft. Der Socket-Typ muss dabei entweder *PUSH* oder *PUB* sein, da entsprechend der Erläuterung in Abschnitt 3.2.2.4 das Push-Pull- oder das Publish-Subscribe-Verfahren verwendet werden soll. Das ZeroMQ-Socket kann dabei, wie in Listing 4.2 gezeigt, geöffnet werden. Dazu muss das *pyzmq*[2] Paket installiert sein und *zmq* importiert werden.

```
1 self.context = zmq.Context()
2 self.sender = self.context.socket(zmq.PUB)
3 self.sender.bind(host + str(port))
```

Listing 4.2: Aufbau der ZeroMQ-Verbindung in mosaik-zmq

Bei jedem Simulationsschritt ruft mosaik die Step-Methode des Simulators auf. Diese bekommt dabei die aktuelle Zeit und die Daten der zuvor in der Szenariodefinition verbundenen anderen Simulatoren übermittelt. Die aktuellen Daten liegen als JSON-Objekt vor und ihnen wird vor dem Senden an das ZeroMQ-Socket der aktuellen Zeitpunkt hinzugefügt, da dieser für die spätere Verarbeitung in Odysseus wichtig ist. Abschließend gibt die Methode die Zeit des nächsten Simulationsschrittes zurück.

4.1.4 Implementierung in Java (MosaikProtocolHandler)

Wie im Entwurf beschrieben, wird als zusätzliche Variante der Anbindung eine Implementierung des Simulators direkt in Odysseus, also in Java, umgesetzt. In Abschnitt 4.1.2 wurde bereits beschrieben, dass eine High-Level API für Java verfügbar ist. Jedoch setzt diese beim Verbinden voraus, dass das ausführbare Java-Programm aus mosaik heraus gestartet wird. Da dies für Odysseus nur unter hohem Aufwand umzusetzen ist und um eine höhere Flexibilität der Verbindung zu ermöglichen, wird daher die Low-Level API verwendet, deren Beschreibung in Abschnitt 3.2.2.2 zu finden ist.

Für das Empfangen von Daten stellt Odysseus das in Abbildung 4.1 dargestellte Access Framework bereit. Die Daten kommen dabei von einem externen *Sensor*, der in diesem Fall das externe Programm mosaik ist, und werden zunächst vom *TransportHandler* empfangen. Dieser kümmert sich dabei lediglich um die Kommunikation und leitet die empfangenen Daten weiter an den *ProtocolHandler*, welcher die Semantik der Daten kennt und sie entsprechend verarbeitet. Um die typischen in Odysseus verwendeten Datenelemente zu erstellen, wird dabei auf den gewünschten *DataHandler* zurückgegriffen. Abschließend werden die nun erzeugten Datenelemente an einen *Source* Operator gesendet, welcher den Beginn des physischen Anfrageplans darstellt.

[2] https://zeromq.github.io/pyzmq/

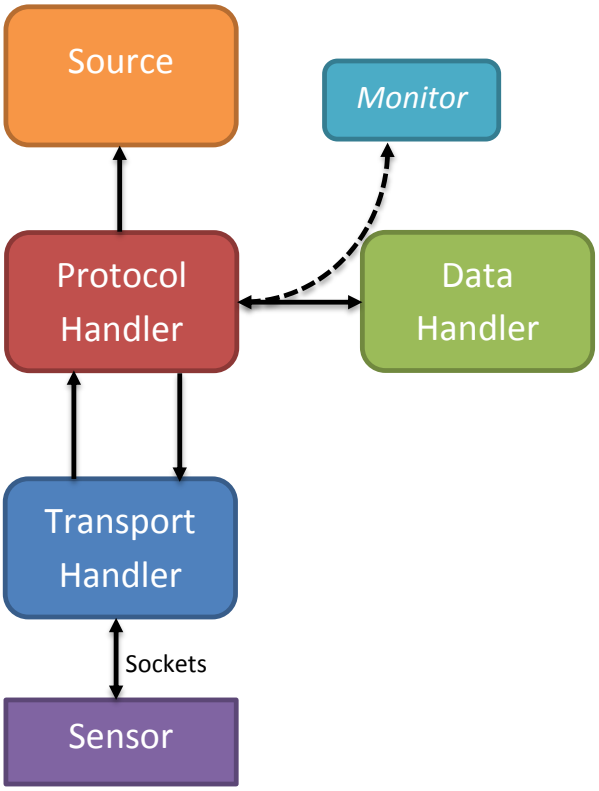


Abbildung 4.1: Access Framework in Odysseus

Da die Kommunikation der Low-Level API über TCP Sockets abläuft, lässt sich als Transport-Handler der bereits vorhandene *TCPServerTransportHandler* verwenden, der zum Empfangen einen *SocketServer* öffnet. Beim Transfer von der API dient JSON als Datentyp, für dessen Verarbeitung der bereits vorhandene *KeyValueObjectHandler* als DataHandler verwendet werden kann. Neu umzusetzen ist somit lediglich ein *ProtocolHandler*.

Die Umsetzung wird im Plug-In *de.uniol.inf.is.odysseus.wrapper.mosaik* bereitgestellt und ist in Abbildung A.2 auf Seite 88 dargestellt. Durch die Trennung des Plug-Ins von der ASE-Berechnung, können beide unabhängig voneinander eingebunden und verwendet werden. Da die Anbindung push-basiert abläuft, also die Nachrichten von mosaik versendet und nicht von Odysseus abgeholt werden, liegt die Hauptfunktionalität in der *process* Methode. Diese wird vom TransportHandler bei Erhalt eines neuen Datenpakets aufgerufen und die Daten in Form eines ByteBuffers übergeben. Da unter Umständen bei der Übertragung über TCP größere Pakete in mehrere Pakete aufgeteilt werden, ist zunächst sicherzustellen, dass bereits eine komplette Nachricht vorliegt. Dazu wird der Header verwendet, der die Länge der Nachricht angibt. Solange sie noch nicht komplett ist, wird der bisher empfangene Teil daher gepuffert. Sobald schließlich eine komplette Nachricht im MosaikProtocol-Handler vorliegt, wird der Typ ausgelesen und die dementsprechende Methode im *OdysseusSimulator* aufgerufen. Dieser stellt die Implementierung der mosaik API dar, stellt jedoch keine weiteren Funktionalitäten bereit, außer, dass er die Methoden der API beantwortet und die empfangenen Daten weiterleitet.

4.1.5 Einbinden in mosaik-Szenario

Für das Einbinden von Simulatoren in mosaik ist der Simulation-Manager zuständig. Dieser bietet grundsätzlich drei verschiedene Möglichkeiten, die in Abbildung 4.2 dargestellt werden.

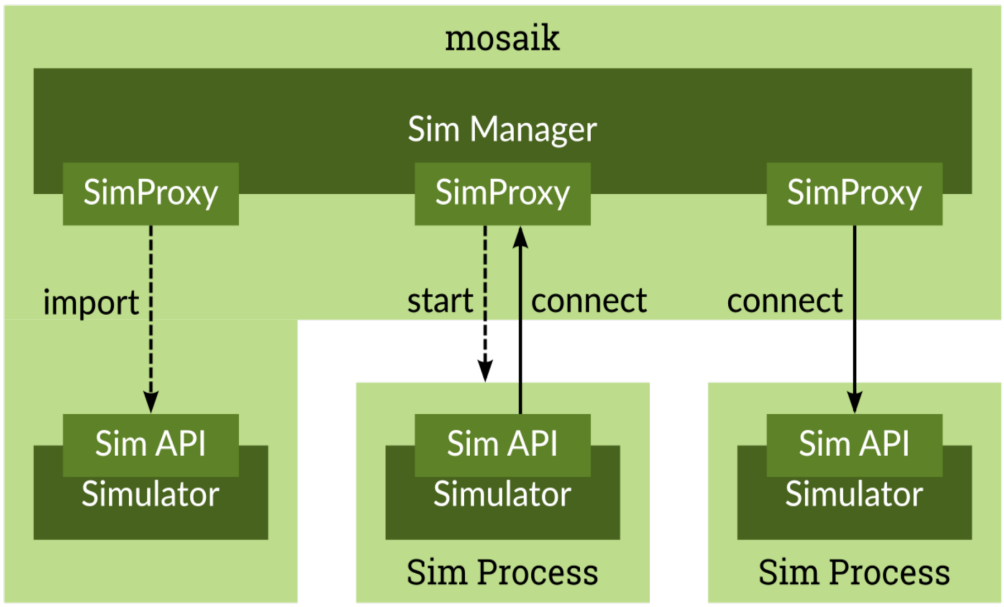


Abbildung 4.2: mosaik Simulation-Manager [mos14]

python (import): Das direkte Einbinden erfolgt mit dem Befehl *python* und funktioniert dementsprechend nur mit in Python 3 implementierten Simulatoren. Dabei wird der Simulator im gleichen Prozess wie mosaik ausgeführt.

cmd (start/connect): Der *cmd* Befehl erlaubt es ein angegebenes Kommando auszuführen, um den Simulator in einem Unterprozess zu starten. Mosaik ersetzt dabei automatisch die Variable *%(addr)s* durch die eigene Netzwerk-Adresse, sodass der gestartete Prozess anschließend eine Verbindung aufbauen kann. Durch den Befehl *cwd* kann zudem ein Arbeitsverzeichnis angegebene werden, zu welchem mosaik wechselt, bevor es den Befehl ausführt.

connect: Zu einem bereits laufenden Simulator kann über *connect* verbunden werden. Dazu ist lediglich die Adresse anzugeben, unter welcher mosaik dann nach dem Simulator sucht. Für den Aufbau der Verbindung muss der Simulator einen SocketServer bereitstellen, der auf mosaiks Nachrichten antwortet.

Damit die beschriebenen Verbindungsarten in mosaik angewandt werden können, müssen sie wie in Listing 4.3 in die Variable *sim_config* eingefügt werden.

```
sim_config = {
    'SimA': {
        'python': 'package.module:SimClass', },
    'SimB': {
        'cmd': 'java -jar simB.jar %(addr)s',
        'cwd': 'simB/dist/', },
    'SimC': {
        'connect': 'localhost:5678', },
}
```

Listing 4.3: Konfigurationsmöglichkeiten zum Einbinden von Simulatoren in mosaik

Da der mosaik-zmq Simulator auf Python basiert, ließe er sich zwar auch direkt einbinden, damit er jedoch in einem eigenen Prozess ablaufen kann, soll er über ein Kommando gestartet werden (vgl. Listing 4.4). Der Java Simulator wird wie beschrieben über den connect Befehl eingebunden (vgl. Listing 4.5). Nachdem jeweils in der ersten Zeile der Eintrag des jeweiligen Simulators in die *sim_config* vorgenommen wird, werden sie in der zweiten Zeile gestartet. In der dritten Zeile wird schließlich das jeweilige Modell erzeugt.

```
sim_config = {'ZMQ': {'cmd': 'mosaik-zmq %(addr)s' } }
zmqModel = world.start('ZMQ', step_size=60, duration=END)
zmq = zmqModel.Socket(host='tcp://*:', port=5558, socket_type='PUB')
```

Listing 4.4: Einbinden des mosaik-zmq-Simulators in ein Szenario

```
sim_config = {'Odysseus': {'connect': '127.0.0.1:5554' } }
odysseusModel = world.start('Odysseus', step_size=60)
odysseus = odysseusModel.Odysseus.create(1)
```

Listing 4.5: Einbinden des Java Simulators in ein mosaik Szenario

4.1.6 Einbinden in Odysseus

Für die Anbindung von Datenquellen verwendet Odysseus den Access-Operator, welcher die Konfiguration von Transport- und DataHandlern für jeden Anwendungsfall ermöglicht. Das zugrunde liegende Access Framework und die Verwendung für den MosaikProtocolHandler wurde bereits in Abschnitt 4.1.4 beschrieben. Für die Verwendung des mosaik-zmq Simulators können die bereits vorhandenen ZeroMQTransportHandler, JSONProtocolHandler und KeyValueObjectHandler Verwendung finden. Tabelle 4.1 bietet eine Übersicht über die Konfiguration der beiden Anbindungen in Odysseus.

Damit die Verwendung der Anbindung möglichst leicht zu bedienen ist, wird ein *MosaikAccessAO* Operator erstellt, welcher die Standardkonfiguration beinhaltet. Über den Parameter *type* kann ausgewählt werden welche der Anbindungen zu verwenden ist. Somit ist die Anbindung von mosaik in Odysseus in einer Zeile möglich, wie in Listing 4.6 für beide Varianten zu sehen ist.

```
mosaikZMQ := MOSAIK({source = "mosaikZMQ", type = "zeromq"})
mosaikAPI := MOSAIK({source = "mosaikAPI", type = "simapi"})
```

Listing 4.6: Vereinfachtes Einbinden von mosaik in Odysseus

Version	mosaik-zmq	MosaikProtocolHandler
Wrapper	GenerishPush	GenericPush
TransportHandler	ZeroMQ	TCPServer
ProtocolHandler	JSON	Mosaik
DataHandler	KeyValueObject	KeyValueObject
Options	host=127.0.0.1, readport=5558, writeport=5559, byteorder=LittleEndian	port=5554, byteorder=LittleEndian

Tabelle 4.1: Konfiguration des MosaikAccessAO

Um andere als die Standardoptionen zu verwenden, können die Einträge im *options* Parameter überschrieben werden. Ebenfalls kann weiterhin der normale Access-Operator verwendet werden, in dem alle Parameter manuell anzugeben sind.

4.2 ASE-Operator in Odysseus

Der ASE-Operator wurde wie im Entwurf in Abschnitt 3.3.4 beschrieben umgesetzt. Da der verwendete ASE-Algorithmus nicht frei verfügbar ist, liegt das ASE-Plug-In in einem zugriffsbeschränkten Bereich in der Versionsverwaltung SVN von Odysseus. Der MosaikProtocolHandler hingegen ist im SVN-Repository frei verfügbar und auch für die weitere Verwendung vorgesehen.

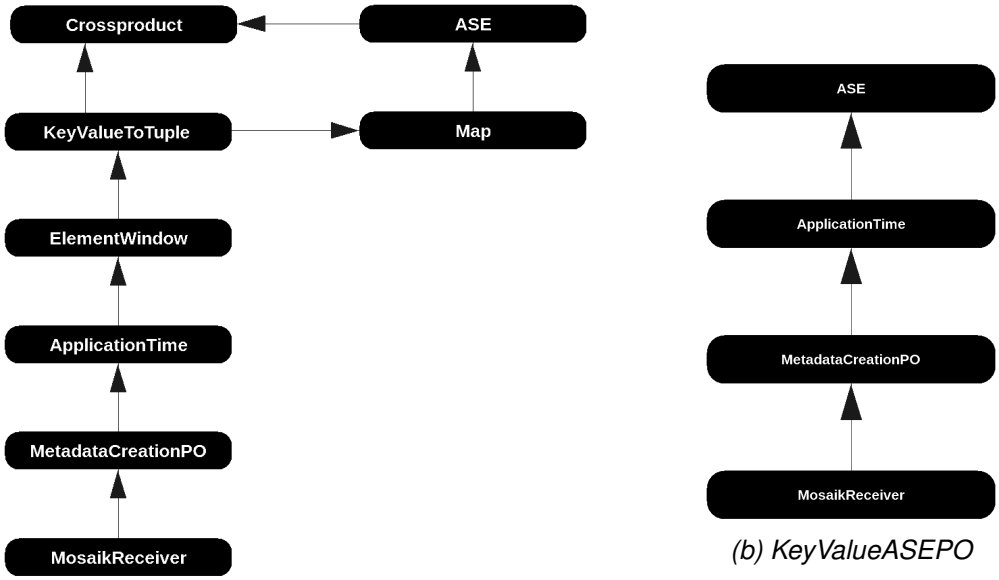
In Odysseus lässt sich der ASE-Operator in der Anfragesprache PQL unter der Bezeichnung ASE einbinden (vgl. Listing 4.7). Die JSON-Konfigurationsdatei für den ASE-Operator folgt dem Entwurf aus Abschnitt 3.3.2.5. Ein Beispiel ist in Abbildung A.4 auf Seite 96 zu sehen. Darin ist die Konfiguration für ein kleines Szenario definiert, welches zum Testen und zur Evaluation verwendet wurde.

```
1 mosaikInput = MOSAIK({source = 'MosaikReceiver', type = 'zeromq'})
2 aseResult = ASE({
3    topologyFile = '${WORKSPACEPROJECT}/Scenario1/Scenario1KV.json'
4    }, mosaikInput)
```

Listing 4.7: Verwendung des Key-Value-ASE-Operators in Odysseus

Bei der Transformation des logischen in einen physischen ASE-Operator wird automatisch, je nach dem Typ der Eingangsdaten, die Variante für Tupel (*ASEPO*) oder Key-Value-Objekte (*KeyValueASEPO*) verwendet, wie im Entwurf beschrieben (vgl. Abschnitt 3.3.3). Während der KeyValueASEPO quasi direkt auf den aus mosaik kommenden Daten arbeiten kann, benötigen die Daten für den ASEPO noch eine Vorverarbeitung. Zwei typische Anfragepläne für die beiden Varianten sind in Abbildung 4.3 dargestellt. Der rechte entspricht dabei der PQL Anfrage aus Listing 4.7.

Der Beginn ist bei beiden Anfragepläne identisch. Der *MosaikReceiver* ist für das Empfangen der Daten zuständig und die darauf folgenden *MetadataCreationPO* und *ApplicationTime* sorgen für das korrekte Setzen der Metadaten. Dabei werden die Metadaten der Datenstromelemente gesetzt. Der KeyValueASEPO kann nun direkt auf diesen Daten arbeiten, da die Zuordnung von Input-Connectoren zu Key-Value-Attributen in der Konfigurationsdatei definiert ist.



(a) ASEPO *(b) KeyValueASEPO*

Abbildung 4.3: Vorverarbeitung der Daten vor dem ASE-Operator

Für den ASEPO müssen die Daten noch im *KeyValueToTuple*-Operator in Tupel umgewandelt und anschließend durch den *Map*-Operator verarbeitet werden. Dabei werden die Daten auch an das Eingabeschema des ASEPO angepasst und Attribute umbenannt, sodass sie bei der Auswertung gut ausgewertet werden können.

Durch die bessere Unterstützung von Tupeln in Odysseus lässt sich die Anfrage bei Verwendung des ASEPO sehr gut erweitern. So kann durch das *ElementWindow* und das *Crossproduct* das Ergebnis der ASE-Berechnung mit den Daten aus mosaik verglichen werden. Dazu sorgt das ElementWindow dafür, dass der Endzeitstempel der Elemente richtig gesetzt wird. In diesem Fall handelt es sich um ein Elementfenster mit der Größe 1, sodass jedes Objekt im Strom nur bis zum darauf folgenden gültig ist. Mit Hilfe der Zeitstempel lassen sich die Datenstromelemente so im Crossproduct wieder zuordnen und somit anreichern, dass sowohl die berechneten als auch die ursprünglichen Daten in einem Element vorhanden sind.

4.3 Visualisierung

Den Entwurf aus Abschnitt 3.4.2 umsetzend, wird ein Eclipse-Plugin erstellt, welches über sogenannte Extensions das Kontextmenü in der Visualisierung des Anfrageplans erweitert (vgl. Abb. 4.4). Dazu ist eine *plugin.xml* Datei und ein vom *org.eclipse.ui.plugin.AbstractUIPlugin* abgeleiteter *Activator* zu erstellen. Der Activator ist im Kontext von OSGi eine Klasse, welche immer beim Starten des Plug-Ins ausgeführt wird. In der plugin.xml kann über ein *org.eclipse.ui.command* der *ASEVisualisationGenerator* mit dem Menüpunkt verknüpft werden [Ecl15]. Da dieser vom in Odysseus bereitgestellten *AbstractCommand* abgeleitet ist, kann durch überschreiben der *isEnabled()* Methode eine Überprüfung hinzugefügt werden, ob der ausgewählte Operator ein ASE-Operator ist. Ansonsten wird der Menüpunkt deaktiviert.

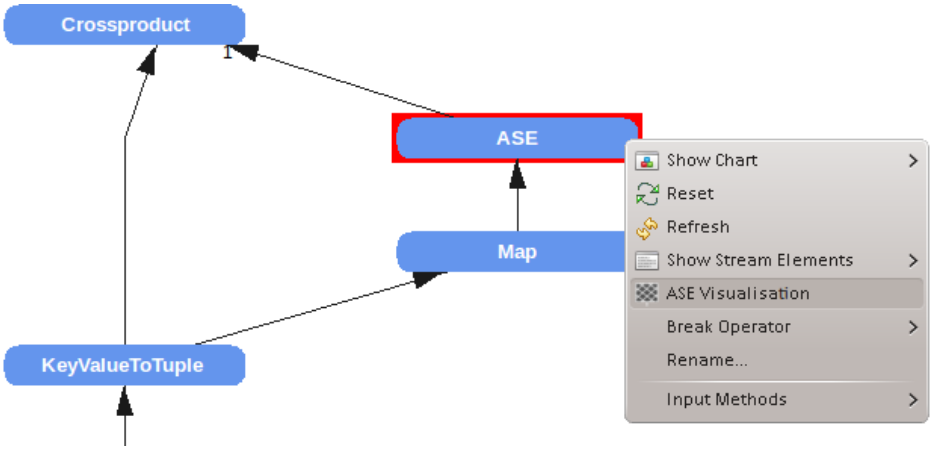


Abbildung 4.4: Menüpunkt zum Generieren der Visualisierung

Der ASEVisualisationGenerator fragt den Anwender zunächst welche Anfragedatei für die Visualisierung verwendet werden soll. Hier ist die Datei des gerade offenen Anfrageplans auszuwählen. Der Dateiname der ausgewählten Datei wird für die kommenden Schritte jeweils als Name der Anfrage verwendet, damit sich die zusammengehörigen Dateien und Anfragen identifizieren lassen.

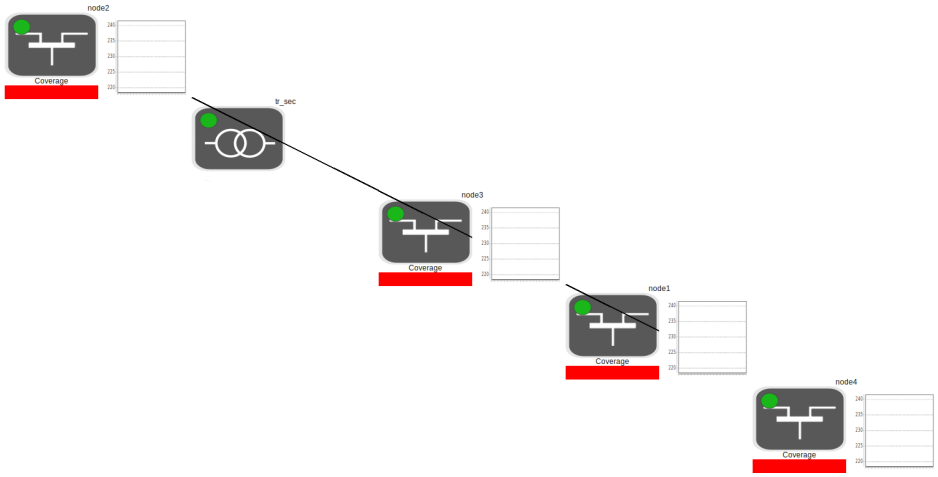


Abbildung 4.5: Graph vor dem Layouting

Für das Layouting verwendet der ASEVisualisationGenerator, wie in Abschnitt 3.4 beschrieben, den durch das Gephi-Toolkit bereitgestellte ForceAtlas2-Algorithmus. Für dessen Verwendung werden zunächst die aus dem *FourWireNetwork* des Operators bekannten Kanten und Knoten in die für das Toolkit nötigen Datentypen umgewandelt. Dazu werden anhand der Informationen des Knotens jeweils Elemente vom Typ *org.gephi.graph.api.Node* erzeugt. Dabei wird der Name des Knoten übernommen und es sind die initialen Koordinaten festzulegen. Da später die automatische Ausrichtung abläuft, werden an dieser Stelle alle Knoten nur in einer Reihe liegend und unsortiert definiert (vgl. Abb. 4.5). Die Kanten lassen sich direkt über die definierenden Knoten im Graphen erstellen.

Nach der Umwandlung der Topologie in die passende Struktur lässt sich der ForceAtlas2-Algorithmus ausführen. Weil die Koordinaten nach Fertigstellung des Layoutings teilweise auch negativ sein kön-

nen und es dadurch zu Problemen in den DashboardParts kommen kann, werden anschließend noch die Minima und Maxima der Koordinaten gesucht und der Graph dementsprechend verschoben, sodass er lediglich positive Koordinaten beinhaltet.

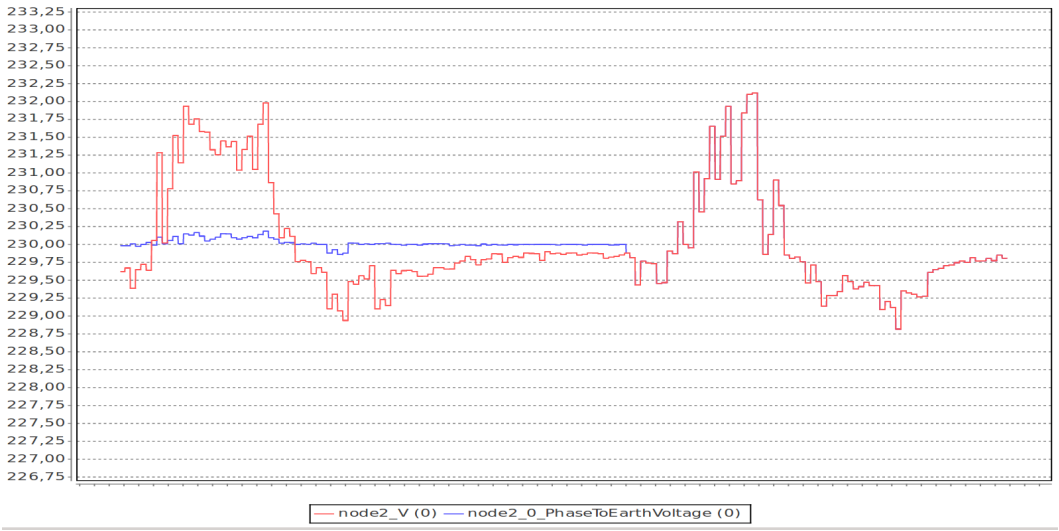


Abbildung 4.6: DashboardPart mit Vergleich der berechneten Spannung mit dem Original aus mosaik

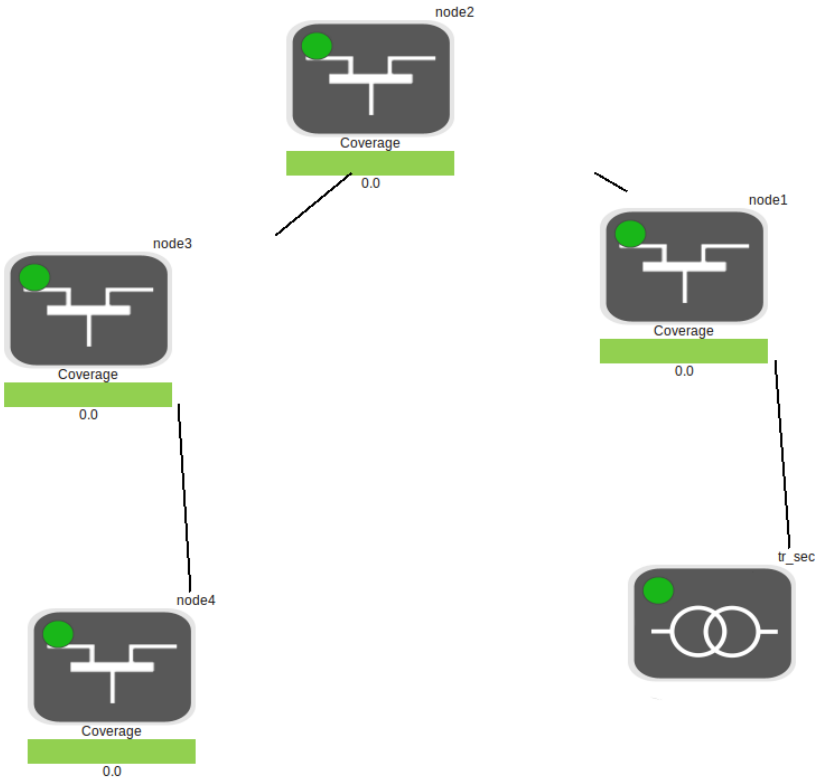


Abbildung 4.7: DashboardPart der Netztopologie

Nach dem Layouting werden die DashboardParts erzeugt. Erst werden Elemente vom Typ *XYLineChart* für alle gewünschten Diagramme erstellt. Für die Auswahl an welchen Knoten die Daten durch Diagramme visualisiert werden sollen, muss in der Konfigurationsdatei des ASE-Operators der entsprechende *visualisationType* angegeben werden. Damit die berechnete Spannung mit der Spannung aus mosaik verglichen werden kann, muss sich zudem hinter dem ASE-Operator ein Join-Operator im Anfrageplan befinden, wie z. B. in Abb. A.7 auf Seite 99 zu sehen. XYLineCharts basieren auf *JFreeChart*[3] und stellen die gewünschten Werte über die Zeit dar, wie in Abbildung 4.6 zu sehen. Die dort abgebildete Situation gehört zu einem simulierten Messstellenausfall, was sich darin zeigt, dass sich zu Beginn die verglichenen Spannungen unterscheiden und am Ende, wenn die Messstelle wieder ordnungsgemäß funktioniert, quasi identisch sind.

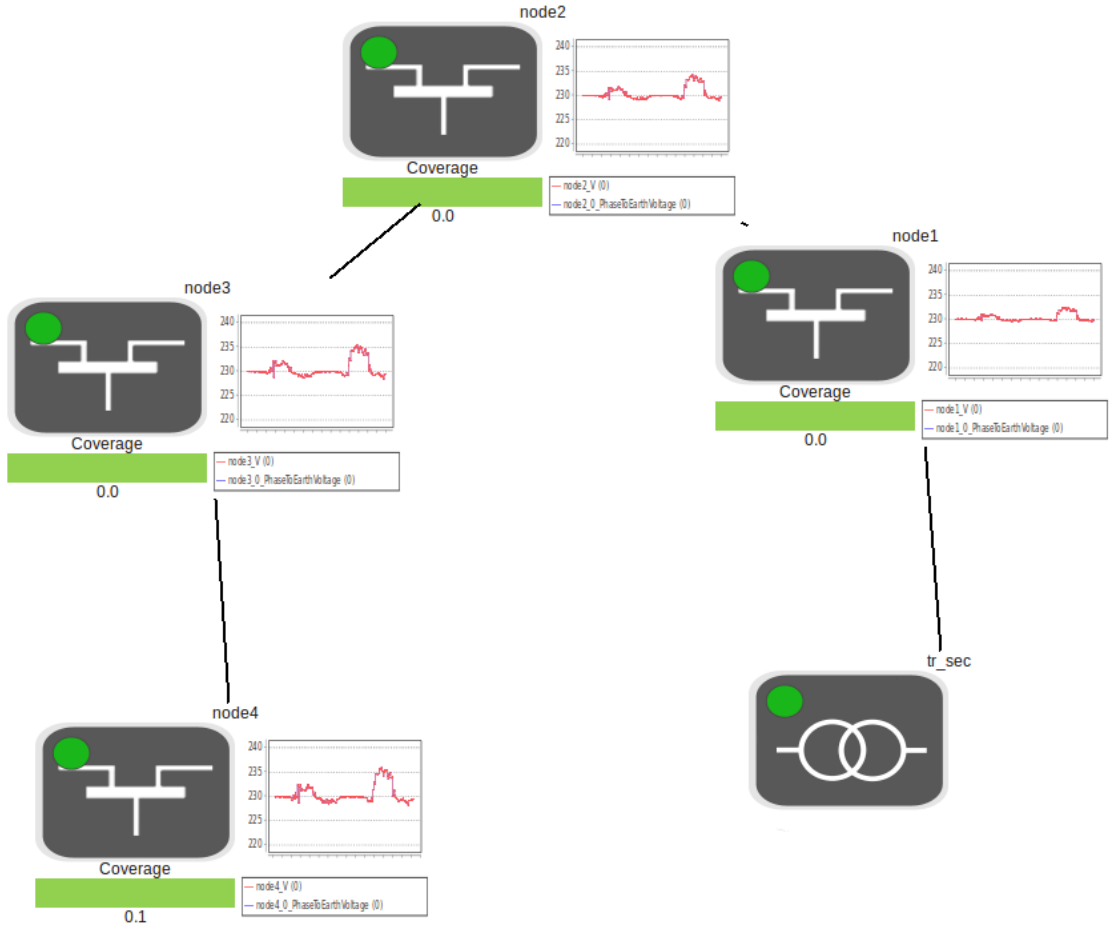


Abbildung 4.8: Komplettes Dashboard

Anschließend wird ein DashboardPart für die Visualisierung der Netztopologie erstellt. Dieser basiert auf dem Typ *DashboardGraphicsPart* und ist in Abbildung 4.7 dargestellt. Die Knoten werden als konstante Icons dargestellt, während für die Visualisierung der Modellabdeckung (*coverage*) sogenannte *MultiImagePictograms* verwendet werden. Diese zeigen je nach Wert der Modellabdeckung verschiedene Bilder. In diesem Fall sind dies einfache farbliche Flächen, wie im Entwurf beschrieben.

[3] `http://www.jfree.org/jfreechart/`

Zuletzt wird ein Dashboard angelegt, dass alle bisher beschriebenen DashboardParts zu einer Visualisierung zusammenfügt und dabei die Diagramme bei den dazugehörigen Knoten anzeigt (siehe Abb. 4.8).

4.4 Zusammenfassung

In diesem Kapitel wurde die Umsetzung des im vorangehenden Kapitel beschriebenen Entwurfs vorgestellt. Zunächst erfolgte dabei die Beschreibung der Anbindung von mosaik und Odysseus. Die von mosaik gelieferten Simulationsdaten wurden dargestellt und eine Übersicht über die API von mosaik 2 gegeben. Anschließend wurden die beiden verschiedenen Umsetzungen in Python und in Java beschrieben und wie diese jeweils in mosaik und Odysseus in ein Szenario bzw. eine Anfrage einzubinden sind.

Dann schloss sich die Beschreibung des ASE-Operators und seiner Integration in Odysseus an. Dabei wurde zudem auf die Vorverarbeitung der Simulationsdaten in Odysseus eingegangen. Darüber hinaus folgte die Beschreibung, wie die automatische Generierung des Dashboards zur Visualisierung umgesetzt werden konnten. Unterpunkte waren dabei das Einfügen der ASE-Visualisierung in das Kontextmenü des Anfrageplans, die Erstellung der DashboardParts vom Typ XYLineChart für das Diagramm und DashboardGraphicsPart für die Visualisierung der Netztopologie. Abschließend wurde gezeigt, wie diese zu einem Dashboard zusammengefügt werden, welches alle gewünschten Visualisierungen beinhaltet.

5 Evaluation

Dieses Kapitel stellt die Evaluation der zuvor beschriebenen Implementierungen vor. Als Grundlage dafür werden zunächst die verwendeten mosaik-Szenarien beschrieben, sowie die Art und Weise, auf welche in Odysseus Messungen durchgeführt werden können. Die Evaluationen beziehen sich auf drei unterschiedliche Aspekte. Der erste Teil dreht sich um die Anbindung von mosaik und Odysseus. Danach wird im zweiten Teil der ASE-Operator evaluiert, bevor schließlich als drittes einige Fragestellungen betrachtet werden, die sich auf die allgemeine Funktionalität des ASE-Algorithmus beziehen.

5.1 Mosaikszenarien

Zur Evaluation dienen zwei Szenarien verschiedener Größen in mosaik, die so verschiedene Möglichkeiten für die Evaluation bieten. Einleitend wird zuvor noch die allgemeine Szenariodefinition in mosaik erläutert.

5.1.1 Allgemeine Szenariendefinition in mosaik

Als Grundlage beider Szenarien dient das mosaik-demo[1] Paket, welches für den Einstieg in das Arbeiten mit mosaik zur Verfügung steht. Es beinhaltet Simulatoren zur Simulation von Haushalten und PV-Anlagen. Die Simulatoren werden dabei, wie in Listing 5.1 zu sehen, in den Python-Code eingefügt.

```
1 sim_config = { 'CSV': {'python': 'mosaik_csv:CSV'},
2     'HouseholdSim': {'python': 'householdsim.mosaik:HouseholdSim'},
3     'PyPower': {'python': 'mosaik_pypower.mosaik:PyPower'}}
```

Listing 5.1: Konfiguration der Simulatoren

Die mosaik-demo beinhaltet insgesamt drei verschiedene Simulatoren. *PyPower* ist dabei für die Simulation des Stromnetzes und die Durchführung von Leistungsflussrechnungen zuständig. Zur Instanziierung ist eine Konfiugrationsdatei nötig, welche im JSON-Format die Struktur des Netzes beinhaltet. Dabei müssen Busse, Transformatoren und Leitungen angegeben werden. Der Simulator *HouseholdSim* simuliert Haushalte als Verbraucher und wird ebenfalls über eine Konfigurationsdatei angepasst. Diese beinhaltet hinterlegte Standardlastgänge für ein Kalenderjahr in 15-minütigem Intervall. Mit Hilfe des *CSV*-Simulators werden zudem übliche Lastgänge einer Photovoltaik-Anlage eingebunden.

Zusätzlich zu den bereits gegebenen Simulatoren werden, je nach dem zu messenden Szenario, der *mosaik-zmq* Simulator (vgl. Abschnitt 4.1.3) oder der auf der Low-Level API basierende Odysseus-Simulator (vgl. Abschnitt 4.1.4) eingebunden, um die Verbindung zu Odysseus herzustellen. Anschließend werden alle Knoten mit der Instanz des verwendeten Simulators verbunden, damit deren Daten gesendet werden. Dieses Szenario simuliert lediglich ein einphasiges Netz, weshalb auch nur die Daten dieser einen Phase übertragen werden. Da sich die evaluierten Szenarien ausschließlich im

[1] https://bitbucket.org/mosaik/mosaik-demo

Niederspannungsnetz abspielen, wird lediglich das dort vorherrschende Strahlennetz für die Evaluation betrachtet (vgl. Abschnitt 2.2.1). Als Referenzknoten dient die Sekundärspule des Transformators.

Die mosaik-demo ist auf eine maximale Simulationsdauer von einem Kalenderjahr ausgelegt. Für die Evaluation soll jedoch nur ein Monat verwendet werden oder für spezielle Messungen auch andere Zeitintervalle, welche dann entsprechend erwähnt werden. Die Schrittgröße der Simulation soll eine Minute betragen. Somit ergeben sich für einen Monat 44.640 zu verarbeitende Simulationsschritte.

5.1.2 Testszenarien

Für die ersten Tests der Funktionalität der Umsetzung wurde ein kleines Netz verwendet, welches sich besonders durch seine Übersichtlichkeit für einige Evaluationen anbot. Dieses besteht aus einem Transformator und vier in einem Strahl angeschlossene Knoten, mit jeweils einem Haushalt und einer Solaranlage, wobei jedoch für die Berechnungen nur die Gesamtleistungsaufnahme des Netzknotens von Belang ist. Die Knoten sind dabei jeweils durch Leitungen mit einer Länge von 100 m mit üblichen Parametern verbunden. Die Visualisierung des Szenarios ist in Abbildung 5.1 zu sehen und die Definitionsdatei ist in Abbildung A.2 auf Seite 94 angegeben.

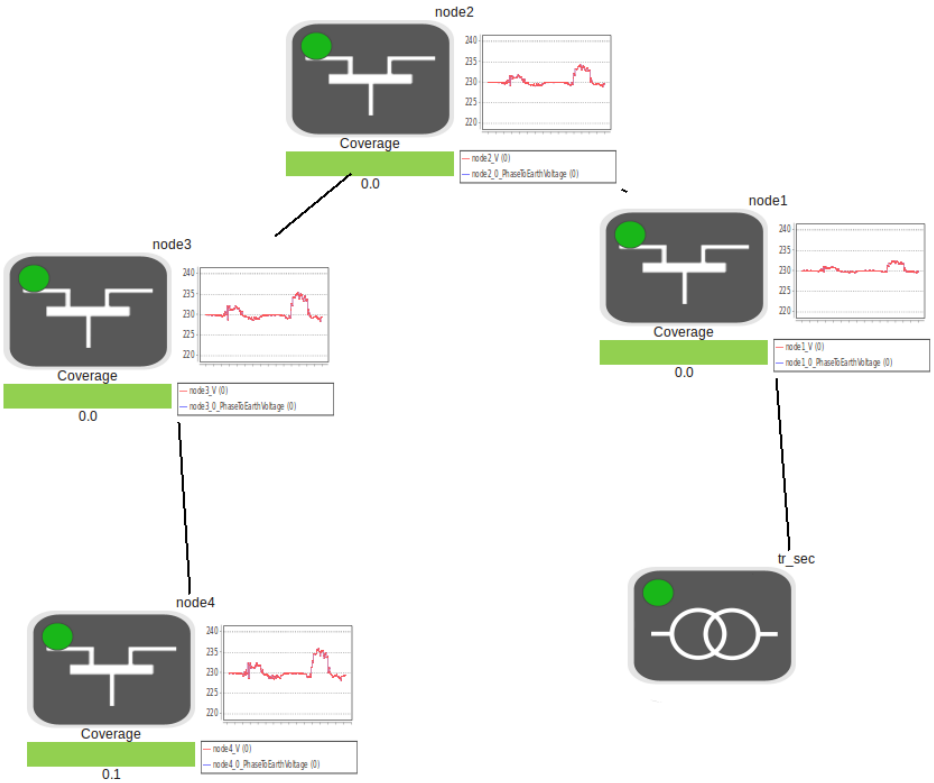


Abbildung 5.1: Aufbau des kleinen Szenarios mit Diagrammen

Als größeres Szenario wird das in der mosaik-Demo bereitgestellte Szenario verwendet. Dieses beinhaltet ein Strahlennetz aus 37 Knoten, die in vier Strängen von einem Transformator ausgehen und teilweise verzweigt sind. Die Leitungen besitzen Längen zwischen 6 m und knapp 300 m. Der exakte Aufbau des Szenarios ist in Abbildung 5.2 zu sehen.

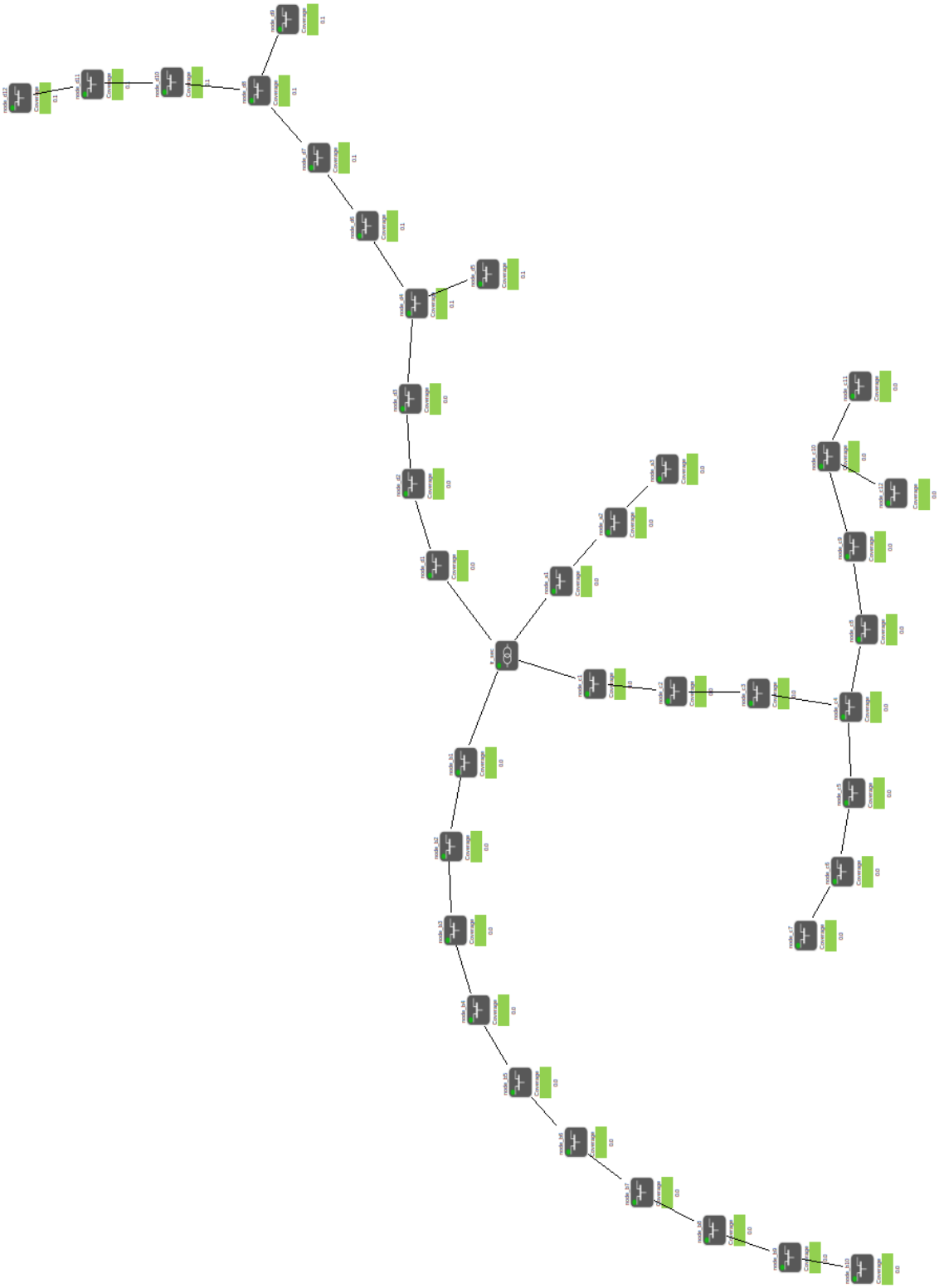


Abbildung 5.2: Aufbau des großen Szenarios ohne Diagramme

5.2 Messungen in Odysseus

Um die Geschwindigkeit und Latenzen der Verarbeitung in Odysseus vergleichen zu können, wurden einige Messungen durchgeführt, die in den folgenden Abschnitten beschrieben werden. Da derartige Evaluationen häufig durchzuführen sind, stellt Odysseus auch bereits einige Operatoren dafür bereit. Zu den wichtigsten zählen dabei *CalcLatency* und *MeasureThroughput* zur Messung von Latenz und Datendurchsatz, die sich an beliebigen Stellen in Anfragen einfügen lassen. CalcLatency berechnet dabei die Zeit, die vergangen ist von dem Zeitpunkt der Erstellung des aktuellen Datenstromelements bis zum Erreichen des Operators. MeasureThroughput gibt Auskunft darüber, wie lange es dauert bis eine bestimmte Anzahl an Elementen angekommen sind. Als Beispiel einer Anfrage zum Messen der Latenz und des Datendurchsatzes sei auf die PQL-Anfrage in Listing A.5 auf Seite 97 verwiesen. Der dazugehörige Anfrageplan ist in Abbildung A.6 auf Seite 99 zu sehen.

Bei der Messung von Latenzen wurde, wenn möglich, auf Fenster verzichtet, da diese die Latenzmessung stark beeinträchtigen. Dies liegt daran, dass die Elemente in einem einelementigen Fenster immer erst weitergesendet werden können, sobald das folgende Element angekommen ist. Somit würde die Simulationszeit von mosaik mit in die Messung einfließen und nicht nur die Zeit der Verarbeitung in Odysseus gemessen werden. Alle im folgenden beschriebenen Ergebnisse wurden auf einem Lenovo X230i mit Intel Core i3-3110M Prozessor mit zwei realen bzw. vier virtuellen Kernen, 16 GB RAM auf einer SSD gemessen. Als Betriebssystem kam dabei Linux Mint 17 zum Einsatz. Für die Darstellung der Messergebnisse werden häufig Boxplots verwendet. Diese zeigen besonders gut die Verteilung der Ergebnisse. Dabei werden für die Darstellung das 2,5%-, 25%-, 75%- und 97,5%-Quantil sowie der Median verwendet.

5.3 Anbindung von mosaik an Odysseus

Die Evaluation der Anbindung beinhaltet zunächst eine allgemeine Beschreibung des Verhaltens der beiden realisierten Verbindungen von mosaik und Odysseus. Zudem wird ihr Verhalten in Bezug auf die Anforderungen des Ein- und Ausklinkens (**A**-3) und Blockierens (**A**-6) betrachtet (vgl. Abschnitt 3.1).

Wie bereits in Abschnitt 3.2.2.4 beschrieben, können beim Aufbau einer Verbindung über das Publish-Subscribe-Prinzip von ZeroMQ die ersten Nachrichten verloren gehen. Dies bestätigt sich auch für die umgesetzte Anbindung. Teilweise kommen alle Pakete an, jedoch passiert es sporadisch, dass zu Beginn bis zu 20 Nachrichten verloren gehen. Da dieses Phänomen jedoch nur zu Beginn der Verbindung auftreten kann, stellt es kein größeres Problem dar. In der Anforderungserhebung war erhoben worden, dass die Anbindung das Ein- und Ausklinken von Odysseus in die mosaik Simulation ermöglichen soll, was mit der Implementierung problemlos möglich ist. Wenn Odysseus die Daten des ZeroMQ-Sockets abonniert, erhält es alle Daten, ansonsten werden diese gar nicht erst versendet, aber die Simulation läuft trotzdem normal weiter.

Um zu Beginn der Evaluation auftretende kleine Probleme beim erneuten Verbinden zu beheben, wurden an dem bereits vor dieser Arbeit bereitstehenden ZeroMQTransportHandler einige Optimierungen zur Fehlerbehandlung durchgeführt. Als weiterer Fehler fiel bei der Evaluation auf, dass die Verbindung zunächst im Vergleich zur RCP-Anbindung deutlich langsamer war und nach etwa 120.000 Tupeln auch Nachrichten verloren gingen. Dieses Problem ließ sich jedoch durch eine Änderung an der *ReceiveBufferSize* des ZeroMQ-Sockets im TransportHandler beheben. Dieser Buffer ist

dafür zuständig, dass die Nachrichten, welche nicht sofort verarbeitet werden können, zwischengespeichert werden. Eine falsche Dimensionierung kann daher zu großen Problemen bei der Verbindung führen.

Die Verbindung über die RCP-Variante ist, wie bereits an früherer Stelle beschrieben, nur möglich, wenn zunächst die Anfrage mit dem MosaikProtocolHandler in Odysseus gestartet wird, da der TransportHandler dabei einen TCP-Socket-Server öffnet. Beim anschließenden Start von mosaik baut dies eine Verbindung zu dem Socket auf und die Kommunikation kann ablaufen. Konzeptionell bedingt, blockiert dabei die Simulation in mosaik. Dadurch ist die Analogie zu einem realen Sensornetz nicht gegeben, die in den Anforderungen beschrieben wird. Jedoch entsteht dadurch im Vergleich zur ZeroMQ-Anbindung der Vorteil, dass die Verarbeitung synchron zur Simulation abläuft. Zudem ist auch die automatische Beendigung der Anfrage beim Ende der mosaik-Simulation möglich.

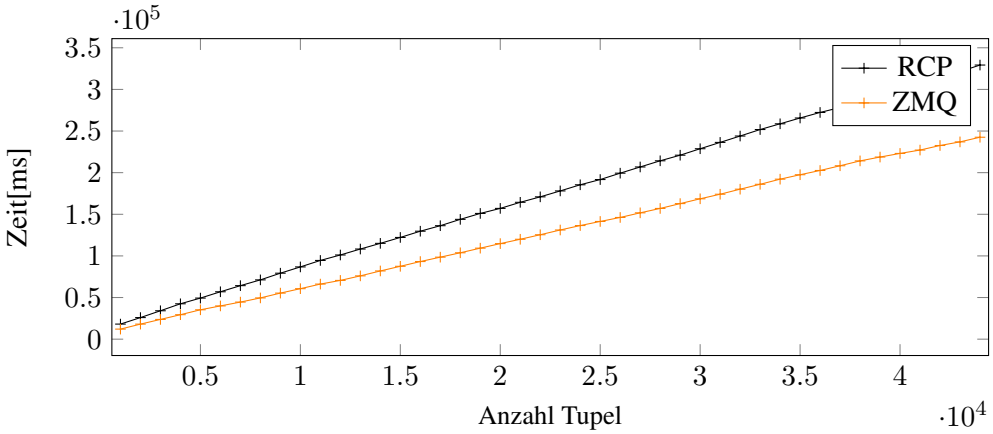


Abbildung 5.3: Durchsatz der ASE-Berechnung im kleinen Szenario

Um die Geschwindigkeit der beiden verschiedenen Anbindungen vergleichen zu können, wurden der Datendurchsatz gemessen. Bei der in Abbildung 5.3 dargestellten Messung, wurde Szenario 1 und eine Berechnung, wie in Abschnitt 5.2 beschrieben, verwendet. Die Ergebnisse zeigen, dass Empfangen und Verarbeiten von 46.000 Elementen mit der ZeroMQ-Anbindung etwa 4:03 Minuten (242.552 ms) und mit der RCP-Anbindung etwa 5:33 Minuten (332.988 ms) dauert. Dies zeigt, dass der Transfer über den MosaikProtocolHandler 37,29% länger braucht, was sich wahrscheinlich dadurch erklären lässt, dass die Simulation in mosaik während der Verarbeitung in Odysseus blockiert. Bei der ZeroMQ-Anbindung kann in dieser Zeit bereits die Simulation weiterlaufen und die nächsten Daten werden versendet.

5.4 ASE-Operator in Odysseus

Nach der Evaluation der Anbindung von mosaik an Odysseus, soll nun der ASE-Operator in Odysseus in Bezug auf Korrektheit und Latenz in verschiedenen Situationen betrachtet werden.

5.4.1 Korrektheit

Da mosaik bereits alle Zustandswerte der Knoten und Leitungen des Netzes liefert, lässt sich die Korrektheit der Ergebnisse des ASE-Operators gut überprüfen. Dazu wird zunächst nur der für die definierten Input-Connectoren benötigte Teil der Daten aus dem Datenstrom herausgefiltert und in den ASE-Operator eingespeist. Nach der Berechnung kann der Ergebnisdatenstrom wieder mit Hilfe des Join-Operators mit den zuvor ausgefilterten Werten angereichert werden. Dies ist problemlos möglich, da jedes Datenobjekt über ein Elementfenster mit einem Zeitintervall seiner Gültigkeit versehen werden kann. Somit lassen sich die einander zugehörigen Daten ohne Weiteres finden. So wird erreicht, dass in jedem Tupel sowohl die von mosaik gelieferten als auch die vom ASE-Operator berechneten Werte bereitstehen. Durch den Map-Operator kann so mit Hilfe von Ausdrücken direkt die Abweichung der Berechnung von den tatsächlichen Werten zu jedem Zeitpunkt ausgegeben werden (vgl. Listing 5.2). Dabei steht $node\$\{i\}\_V$ für die von mosaik gelieferte Spannung, $node\$\{i\}\_0\_PhaseToEarthVoltage$ für die berechnete Spannung und $dif\$\{i\}$ ist das neu berechnete Attribut. Eine Übersicht über den Anfrageplan kann in Abbildung A.7 auf Seite 99 betrachtet werden.

```
1 mapp = MAP({EXPRESSIONS = [
2    #LOOP i 1 UPTO 4
3       ['node${i}_V - node${i}_0_PhaseToEarthVoltage', 'dif${i}'],
4    #ENDLOOP
5    'timestamp' ]}, joined)
```

Listing 5.2: Berechnen der Abweichung in Odysseus

Zum Vergleich wurden die Anfragen für das kleine Szenario vollständig bestimmt, über- und unterbestimmt ausgeführt. Bei dem unterbestimmten Versuch wurde die Blindleistung eines Knotens nicht verwendet und beim überbestimmten besitzt der ASE-Operator Input-Connectoren für Spannung, Spannungswinkel, Wirk- und Blindleistung für alle Knoten. Zudem wurde das große Szenario vollständig bestimmt gemessen.

Attribut	Szenario 1	Szenario 1 unterb.	Szenario 1 überb.	Szenario 2
Maximale Abweichung	$2,27 \cdot 10^{-5}$	$1,99 \cdot 10^{-2}$	$3,16 \cdot 10^{-6}$	$5,25 \cdot 10^{-4}$
Mittelwert	$2,55 \cdot 10^{-7}$	$4,88 \cdot 10^{-4}$	$1,28 \cdot 10^{-8}$	$1,15 \cdot 10^{-5}$
Standardabweichung	$1,38 \cdot 10^{-6}$	$1,60 \cdot 10^{-3}$	$9,06 \cdot 10^{-8}$	$1,31 \cdot 10^{-9}$
Varianz	$1,92 \cdot 10^{-12}$	$2,72 \cdot 10^{-6}$	$8,20 \cdot 10^{-15}$	$3,62 \cdot 10^{-5}$

Tabelle 5.1: Abweichung der durch den ASE-Operator von den in mosaik berechneten Spannungen

An den Ergebnissen in Tabelle 5.2 ist zu sehen, dass die Abweichung sehr gering ist. Im Vergleich der drei verschieden bestimmten Varianten des kleinen Szenarios lässt sich erkennen, dass der Fehler, wie zu erwarten, beim überbestimmten Netz deutlich geringer ist als beim unterbestimmten.

5.4.2 Geschwindigkeit

Nach der Korrektheit der berechneten Werte soll nun auch die Geschwindigkeit und Latenz der Implementierung analysiert werden. Dafür wurde zunächst der normale Durchlauf einer vollständig bestimmten Leistungsflussrechnung mit der RCP-Anbindung gemessen, wie in Abbildung A.6 dargestellt.

	Szenario 1	Szenario 2
Durchschnittliche Latenz	$2,35$ ms	$277,87$ ms
Durchschnittliche Latenz / Anzahl Knoten	$0,47$ ms	$7,31$ ms

Tabelle 5.2: Latenz bei verschiedenen Szenarien

Abbildung 5.2 zeigt die dabei gemessene durchschnittliche Latenz und zudem die Latenz dividiert durch die Anzahl an Knoten des Szenarios. Die Ergebnisse zeigen, dass die Latenz deutlich stärker als linear ansteigt. Außerdem wird deutlich, dass die ASE-Berechnung bei der verwendeten Netzgröße ohne Probleme die Zustandsüberwachung in Echtzeit durchführen kann. Die Evaluationsszenarien waren jedoch relativ klein im Vergleich zu realen Netzen mit mehreren Tausenden von Knoten.

Doch nicht nur die Gesamtlatenz der Anfrage ist interessant, sondern es wurden ebenso die Latenzen an den einzelnen Operatoren gemessen. Aus den an den verschiedenen Positionen der Anfrage gemessenen Latenzen wurden die Differenzen gebildet, sodass sich die Verarbeitungszeiten der einzelnen Operatoren in Abbildung 5.4 darstellen lassen.

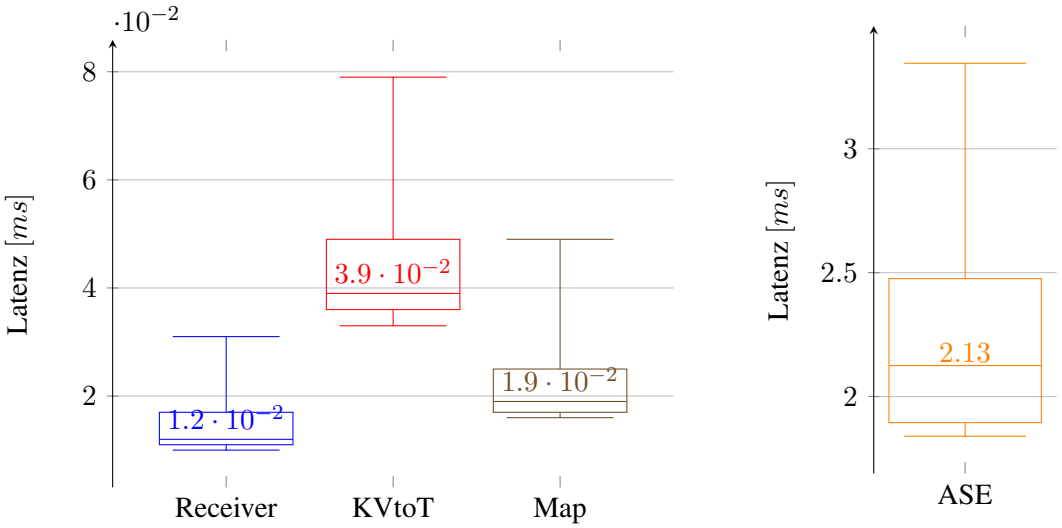


Abbildung 5.4: Latenz an verschiedenen Operatoren in Szenario 1

Wie zu erwarten, ist die Verarbeitungszeit des ASE-Operators deutlich größer als bei den anderen Operatoren. Trotzdem soll für eine bestmögliche Optimierung zusätzlich untersucht werden, wie sich die Latenz verhält, wenn nicht benötigte Daten bereits früher gefiltert werden. Denn bei der Realisierung der Anbindung werden standardmäßig alle Daten des Szenarios aus mosaik versendet. Dadurch kann es zu großen Datenmengen kommen und ggf. ist nur ein kleiner Teil davon wirklich für die ASE-Berechnung nötig. Drei verschiedene Orte des Filterns sind zu vergleichen:

Filtern direkt vor dem ASE-Operator: Mosaik sendet alle Daten an Odysseus. Dort werden alle Daten der verwendeten Netzknoten in Tupel umgewandelt und erst im Map-Operator, direkt vor der ASE-Berechnung, die nicht benötigten Daten herausgefiltert. Diese Variante bietet in Odysseus die komfortabelste Verwendung der Daten. So lässt sich die Anbindung an mosaik inklusive der Umwandlung in Tupel als Quelle definieren und anschließend bequem für verschiedene Weiterverarbeitungen verwenden, ohne dass in die Simulation in mosaik eingegriffen werden muss. Es bietet sich also gerade dann an, wenn verschiedene Berechnungen auf den Daten durchgeführt werden sollen.

Filtern direkt nach dem Empfang in Odysseus: Mosaik sendet alle Daten an Odysseus. Bereits der KeyValueToTuple-Operator wandelt nur die tatsächlich benötigten Werte in Tupel um.

Filtern in mosaik: Bereits in der Szenariodefinition in mosaik werden nur die wirklich benötigten Daten versendet. Das Filtern in mosaik ist dabei unter Umständen etwas komplizierter als in Odysseus und setzt gewisse Erfahrungen mit mosaik und Python voraus. Diese Variante bietet sich hauptsächlich an, wenn eine bestimmte Berechnung in Odysseus durchzuführen ist und nicht erwartet wird, dass weitere Daten benötigt werden könnten.

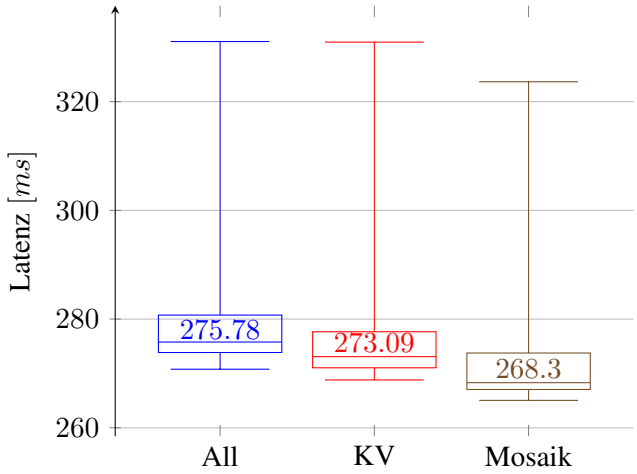


Abbildung 5.5: Filtern an verschiedenen Positionen in der Anfrage in Szenario 2

Um die tatsächlichen Unterschiede der beschriebenen Möglichkeiten zu bestimmten, wurden einige Messungen der Latenz durchgeführt. Dazu fand das große Szenario Verwendung, auf dessen Daten eine vollständig bestimmte Leistungsflussrechnung durchgeführt wurde. Die Messungen beziehen sich dabei auf die Simulation von einem Tag in mosaik, also 1.440 Simulationsschritte. Die in Abbildung 5.5 dargestellten Ergebnisse der Messungen zeigen, dass sich der Ort des Filterns auf die Latenz der Verarbeitung auswirkt. Jedoch ist der Unterschied im Vergleich zu der Verarbeitungszeit im ASE-Operator, wie gezeigt, eher zu vernachlässigen. Daher lässt sich sagen, dass das Filtern in mosaik in Szenarien Sinn ergeben kann, deren Geschwindigkeit sehr wichtig ist und zusätzlich keine weitere Anpassbarkeit gefordert ist. Für eine bessere Wiederverwendbarkeit reicht es jedoch die Daten in mosaik entsprechend der gewünschten Verwendung zu filtern. Der Ort des Filterns kann zudem vom Vorwissen des Anwenders abhängig gemacht werden, sodass ein erfahrener mosaik-Nutzer leicht das mosaik-Szenario entsprechend anpassen kann, während einem erfahrener Odysseus-Nutzer das Filtern in Odysseus einfacher erscheint.

5.4.3 Visualisierung

Mit Hilfe der in Odysseus existierenden Dashboards lassen sich das Netz sowie die ASE-Ergebnisse übersichtlich visualisieren. Allerdings wurden die Dashboards bisher noch nicht in dieser automatisch generierten und zusammengesetzten Form verwendet. So führt es z. B. zu Problemen, dass verschiedene DashboardParts darin übereinander angeordnet sind. Dadurch können sie teilweise nicht markiert werden, um Änderungen vorzunehmen. Noch elementarer sind jedoch die Geschwindigkeitsprobleme. Für das kleine Szenario ist das Visualisierungsdashboard gut nutzbar, jedoch kommt es bei dem großen Szenario bereits zu starken Verzögerungen. Aus diesem Grund wurde nachträglich noch der Parameter *visualisationType* für die Output-Connectoren eingeführt. Auf diese Weise lässt sich die Visualisierung auf bestimmte Knoten beschränken und infolgedessen die Geschwindigkeit erhöhen.

Eine weitere Möglichkeit zur Verbesserung wäre die Entwicklung von spezialisierten Visualisierungskomponenten. Die bisher verwendeten Elemente sind auf eine sehr allgemeine Verwendbarkeit ausgelegt und bieten tatsächlich wesentlich mehr Anpassungsmöglichkeiten, als für den speziellen Fall der Visualisierung der Ergebnisse des ASE-Operators benötigt.

5.5 ASE-spezifische Evaluationsfragen

Nachdem zuvor die Umsetzung evaluiert wurde, folgen nun Fragestellungen, welche das ASE-Verfahren als solches behandeln. Durch die Implementierung der Berechnung in Odysseus lassen sich ohne großen Aufwand interessante Szenarien modellieren und somit Fragestellungen beantworten, die sich für das Verhalten der ASE-Berechnung in verschiedenen Situationen ergeben.

5.5.1 Robustheit bei Ausfall von Input-Connectoren

Der große Vorteil des ASE-Verfahrens gegenüber herkömmlichem NR-Verfahren und State Estimation ist die Robustheit bei fehlenden Informationen zu einzelnen Knoten des Netzes. Diese Robustheit ist mit Hilfe der im Rahmen dieser Arbeit durchgeführten Umsetzung in Odysseus sehr gut zu evaluieren. Dabei soll in Odysseus der Ausfall von Sensoren simuliert werden, um anschließend die Auswirkungen auf die Ergebnisse auszuwerten.

```
1 tuples = MAP({EXPRESSIONS = ['tr_sec_V', 'tr_sec_Vang',
2    'node1_P', 'node1_Q', 'node2_P', 'node2_Q', 'node3_P',
3    ['eif((timestamp < 25000 || timestamp > 200000), node3_Q, null)', '
        node3_Q'],
4    'node4_P', 'node4_Q'],
5    ALLOWNULL = true, SUPPRESSERRORS = true
6    }, mosaikTuples)
```

Listing 5.3: Simulation eines Input-Connector-Ausfalls

Ausfälle von Input-Connectoren lassen sich mit Hilfe des Map-Operators, wie in Listing 5.3 gezeigt, simulieren. In Abhängigkeit vom aktuellen Zeitstempel wird dabei entweder der tatsächliche Wert oder *null* gesendet. Die Ergebnisse des gezeigten Ausfalls sind in Abbildung 5.6 dargestellt. Sie zeigt, dass sich dieser Ausfall der Blindleistung ohne Probleme kompensieren lässt und die Ergebnisse trotzdem bis auf minimale Abweichungen der tatsächlichen Spannung entsprechen. In Abbildung

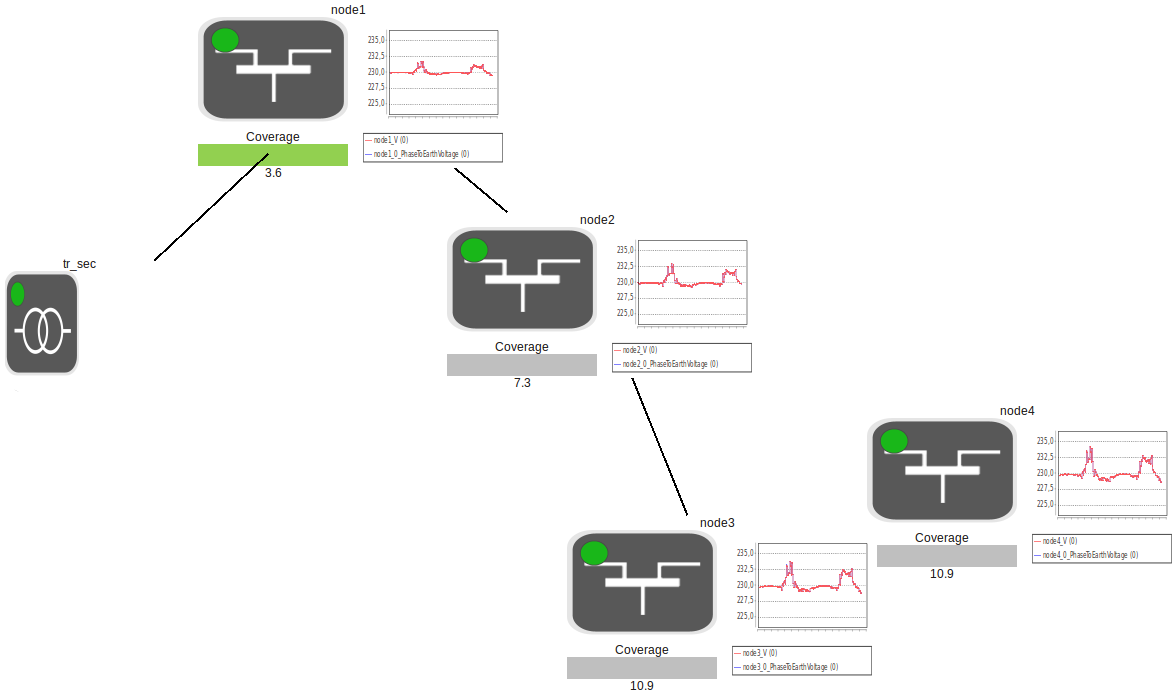


Abbildung 5.6: Verhalten der geschätzten Spannung bei Ausfall des Sensors für die Blindleistung an Knoten 3

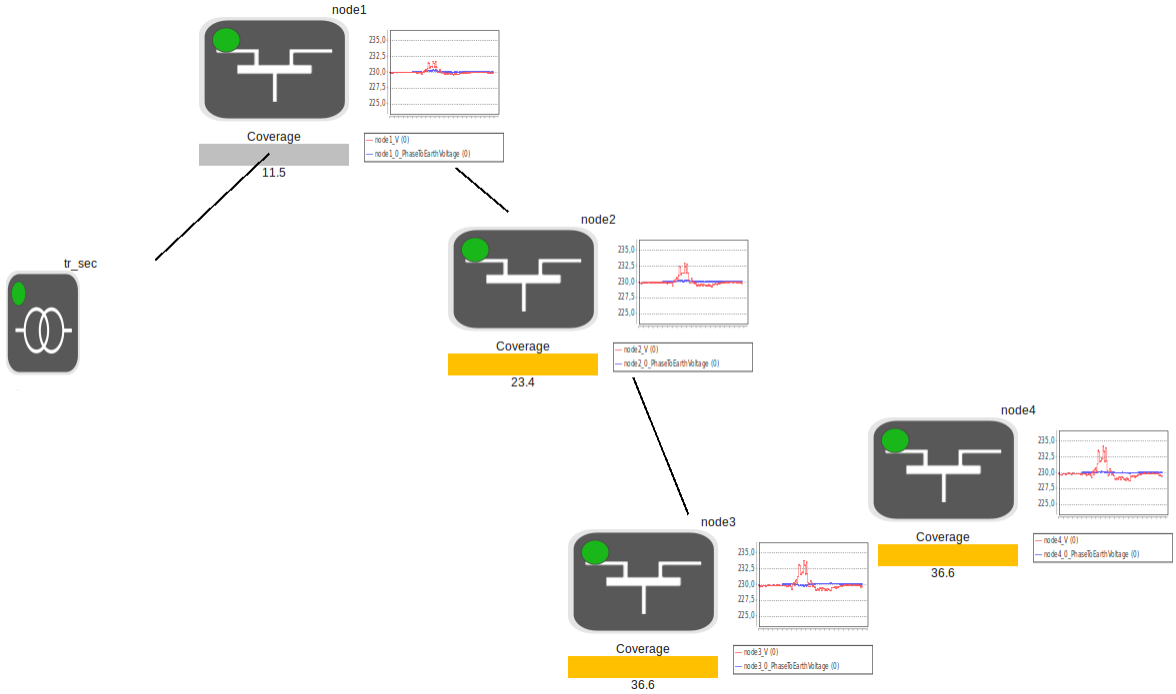


Abbildung 5.7: Verhalten der geschätzten Spannung bei Ausfall des Sensors für die Wirkleistung an Knoten 3

5.7 hingegen sind die Ergebnisse für den Ausfall des Wirkleistungssensors an Knoten 3 zu sehen. Dies führt zu größeren Problemen in den Ergebnissen, wie am höheren Wert der Modellabdeckung und den deutlichen Abweichungen der berechneten zu den tatsächlichen Werten in den Diagrammen sichtbar wird.

5.5.2 Modellabdeckung bei verschiedenen Lastsituationen

Da das ASE-Verfahren auf dem NR-Verfahren beruht, ist das Ergebnis abhängig vom Iterationsstart und dem Betriebspunkt. Daher müsste sich die Modellabdeckung bei verschiedenen Lastsituationen unterscheiden. Zur Überprüfung dieser Hypothese diente das kleine Szenario in einer unterbestimmten Konfiguration mit einem fehlenden Input-Connector der Wirkleistung an Knoten 3. Abbildung 5.8 zeigt, dass sich die Modellabdeckung über die Zeit leicht ändert, obwohl keine Unterschiede in der Konfiguration vorlagen.

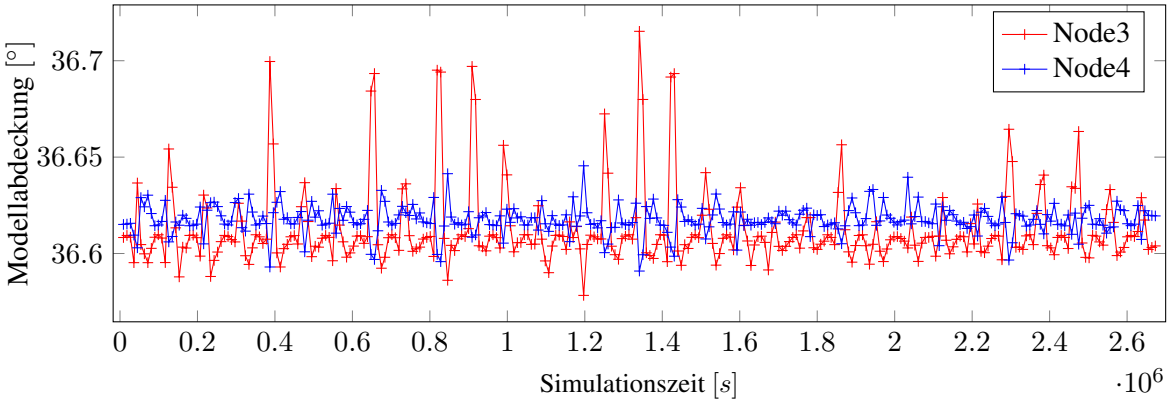


Abbildung 5.8: Modellabdeckung in unterbestimmtem Szenario

5.6 Zusammenfassung

Dieses Kapitel beinhaltet die Evaluation der Umsetzung. Zu Beginn erfolgte dabei die Erläuterung der Evaluationsumgebung, die von zwei unterschiedlich großen mosaik-Szenarien und verschiedenen Mess-Szenarien mit den Evaluationsoperatoren zur Messung der Latenz und des Durchsatzes in Odysseus gebildet wird. Darauf aufbauend zeigte die Evaluation der beiden Realisierungen der Anbindung von mosaik und Odysseus deren Unterschiede in Bezug auf die Verwendung und Geschwindigkeit. Dabei ist die auf ZeroMQ basierende Lösung etwas schneller im Datendurchsatz, da sie die Simulation in mosaik nicht blockiert. Ihr Verhalten ist somit eher datenstromtypisch im Vergleich zum MosaikProtocolHandlers, dessen Bindung an mosaik deutlich enger ist.

Der ASE-Operator wurde in Hinblick auf die Korrektheit der Ergebnisse untersucht, wobei die ASE-Ergebnisse mit den in mosaik errechneten Werten in verschiedenen Szenarien verglichen wurde. Des Weiteren wurde die Geschwindigkeit verschiedener Szenarien, Konfigurationen, Filterpositionen und an verschiedenen Positionen im Datenstrom gemessen und verglichen. Zum anderen konnten mit Hilfe der Umsetzung Fragestellungen, welche die allgemeine Funktionalität der ASE betreffen, betrachtet werden. So wurde mit Hilfe der Umsetzung in Odysseus gut die Robustheit der

ASE-Berechnung gegenüber Sensorausfällen dargestellt werden. Zudem ließ sich die sich ändernde Modellabdeckung bei gleich bleibender Konfiguration aber veränderter Lastsituation zeigen.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit konnten zwei verschiedene Varianten der Online-Anbindung von mosaik an Odysseus und ein Operator für die ASE-Berechnung in Odysseus umgesetzt werden.

Als Grundlage der Implementierungen diente dabei die Anforderungserhebung. Diese definierte die Umsetzung der ASE-Berechnung in Odysseus als Hauptaufgabe der Arbeit. Zudem war es ein Ziel die Ergebnisse der Berechnung in Odysseus visualisieren zu können. Als Datenquelle für die Berechnungen sollte mosaik verwendet werden. Dabei sollte bei der Anbindung von mosaik an Odysseus besonderes Augenmerk auf der Unabhängigkeit liegen, damit sie auch für weitere Anwendungsfälle einsetzbar ist. Abschließend wurde die Überprüfung der Ergebnisse gefordert. Die Umsetzung dieser Anforderungen erfolgte so in den vier Arbeitspaketen Anbindung, Operator, Visualisierung und Evaluation.

Zunächst wurden zwei verschiedene Anbindungen von mosaik an Odysseus implementiert, welche erstmalig die Online-Verarbeitung der Daten in Odysseus ermöglichen. Die beiden Varianten besitzen verschiedene Vor- und Nachteile und bieten sich daher für den Einsatz in verschiedenen Situationen an. Die Implementierung über ZeroMQ liefert ein eher datenstromtypisches Verhalten. Alle anfallenden Daten in mosaik werden dabei sofort versendet und Odysseus kann sich jederzeit Ein- und Ausklinken. Beim Versenden wird also nicht beachtet, ob die Verarbeitung schnell genug erfolgt, oder ob es sonstige Probleme geben kann. Bei zu hohem Datenaufkommen ist daher Odysseus für die Aufrechterhaltung einer ordnungsgemäßen Verarbeitung zuständig. So müssen z. B. Daten verworfen werden. Dies spiegelt die Gegebenheiten eines realen Sensornetzes wider. Demgegenüber bietet die Umsetzung über die Low-Level API von mosaik eine engere Bindung der beiden Systeme. Dabei ist sie geringfügig langsamer und die Abhängigkeit beider Systeme ist größer. Deshalb muss beim Start einer Simulation in mosaik bereits die Anfrage in Odysseus gestartet sein, damit der Verbindungsaufbau korrekt funktioniert. Zudem wird die Simulation beendet, sobald die Verbindung getrennt ist und der Simulator wird durch die Anbindung blockiert, da er immer auf eine Antwort wartet. Dies hat jedoch wiederum den Vorteil, dass eine Simulation in mosaik und die entsprechende Verarbeitung in Odysseus immer synchron ablaufen.

Aufbauend auf den aus mosaik stammenden Daten wurde anschließend die Realisierung der ASE-Berechnung in Odysseus umgesetzt. Dafür konnte die bereits gegebene Implementierung des ASE-Algorithmus verwendet werden, welche in Form eines neuen ASE-Operators in Odysseus integriert wurde. Für die Konfiguration des Operators sind allgemeine Informationen zur Netztopologie und ASE-spezifische zu den gewünschten Input- und Output-Connectoren über eine JSON-Datei anzugeben. Der Operator kann als Eingabe sowohl direkt von mosaik empfangene Key-Value-Objekte als auch in Odysseus vorverarbeiteten relationale Tupel nutzen. Dies ermöglicht die von Odysseus bereitgestellten Operatoren für Anfragen zu verwenden. Die Ergebnisse des ASE-Operators werden hingegen ausschließlich in Form von relationalen Tupeln erstellt.

Die ASE-Ergebnisse lassen sich zudem mit Hilfe von automatisch generierten Dashboards visualisieren. Dabei wird zum einen die Netztopologie als Graph der einzelnen Netzknoten und Verbindungen dargestellt. Für die Ausrichtung der Knoten findet der ForceAtlas2-Algorithmus Anwendung. Zum anderen erfolgt auch eine Visualisierung der Ergebnisse der Output-Connectoren, welche sich sowohl direkt als konkreter Wert mit einer farblichen Hervorhebung anzeigen lassen, als auch als Diagramm über die Zeit. Da die Dashboards ursprünglich nicht für die automatische Generierung entwickelt wurden, erwies sich die Umsetzung teilweise als etwas kompliziert und im Nachhinein

hätte wahrscheinlich besser eine spezialisierte Visualisierungskomponente entwickelt werden sollen. Zudem ließe sich durch diese auch die Geschwindigkeit und Skalierbarkeit der Visualisierung optimieren. So wurden bisher nur bereits vorhandene Visualisierungskomponenten verwendet, welche sehr allgemein gehalten und möglichst stark konfigurierbar sind, um ein weites Anwendungsspektrum zu bedienen. Doch dadurch gibt es auch einen deutlichen Overhead, welcher zu Geschwindigkeitsproblemen bei Visualisierungen mit vielen Elementen führt und durch spezialisierte Komponenten vermindert werden könnte.

Die Evaluation des ASE-Operators wurde mit Hilfe von zwei Szenarien verschiedener Größen durchgeführt. Dabei konnte die Korrektheit der Ergebnisse bestätigt und die Geschwindigkeit auf verschiedenen Daten gemessen werden. So konnte gezeigt werden, dass die ASE-Berechnung für die getesteten Szenarien ohne Probleme eine Echtzeit-Überwachung durchführen kann. Zudem ermöglichte es die flexible Verarbeitung in Odysseus zu zeigen, dass sich der Algorithmus robust gegenüber Unter- und Überbestimmtheit des Netzes verhält.

Durch die beschriebenen Implementierungen konnte eine Online-Anbindung der Smart Grid Simulationsumgebung mosaik mit dem DSMS Odysseus umgesetzt werden. Da im Rahmen dieser Arbeit also erstmalig die Möglichkeiten dieser Verbindung betrachtet wurden, bieten sich noch viele Möglichkeiten für weitere Untersuchungen in zukünftigen Arbeiten und Projekten. Dies ist vor allem durch die allgemein gehaltene Umsetzung möglich, welche somit vollkommen unabhängig von der umgesetzten ASE-Berechnung zu betrachten ist.

Die Realisierung des MosaikProtocolHandlers könnte z. B. in der Art erweitert werden, dass Odysseus auch Daten an mosaik zurückliefert und dadurch die Verwendung von Odysseus als Simulator in mosaik möglich ist. So könnte Odysseus bspw. zur Überwachung bestimmter Werte oder Netzzustände Verwendung finden und abhängig von diesen Daten Steuerungsbefehle an die mosaik-Simulation senden.

Neben den weiteren Möglichkeiten, welche die Anbindung von mosaik und Odysseus bietet, sind auch Weiterentwicklungen bzgl. der ASE-Berechnung in Odysseus möglich. Als erstes ist in dieser Beziehung die DSL zur Beschreibung von Szenarien zu nennen. In der ersten Version von mosaik gab es noch die Beschreibungssprache MoSL, welche inzwischen nicht mehr unterstützt wird. Falls in Zukunft ein Nachfolger Einzug in mosaik halten sollte, wäre eine Integration für die ASE-Berechnung in Odysseus sehr interessant, weil dadurch doppelte Definitionen der Netztopologie entfallen könnten.

Ein zweiter interessanter Aspekt wäre die Verwendung von probabilistischen Datenmodellen. In Odysseus ist eine auf [Tra13] basierende Umsetzung gegeben, welche z. B. einen Kalman Filter und ein Gaussian Mixture Model (GMM) beinhaltet. Der Kalman Filter betrachtet Werte über eine Zeit und kann Rauschen und andere Unstimmigkeiten erkennen und Schätzungen über die wirklichen Werte berechnen. Die genaue Theorie hinter dem Filter wurde 1960 von R. E. Kalman in [Kal60] beschrieben. Das GMM wird meist zum Clustern oder zur Dichtebestimmung eingesetzt. Dabei wird die Verteilung von mehreren Variablen betrachtet und eine Wahrscheinlichkeitsverteilung basierend auf einer Kombination von mehreren Gauß-Verteilungen erzeugt [Tra13, S. 59ff.]. Diese Funktionalität kann z. B. Verwendung finden, wenn Sensoren eine bekannte spezifische Ungenauigkeit aufweisen. Ihre Werte werden dann unter Berücksichtigung von stochastischen Eigenschaften betrachtet, um Rauschen und Ausreißer aus den Daten zu entfernen. Der Einsatz von probabilistischen Methoden kann gerade in der Energiedomäne interessante Möglichkeiten bieten, da häufig auch die dort verwendeten Bauteile eben diese beschriebenen Ungenauigkeiten aufweisen, welche auch zu Ungenauigkeiten bei den weiteren Berechnungen führen können.

Der dritte potentielle Ansatzpunkt für weitere Arbeiten setzt bei der Realisierung der ASE-Berechnung an. Bei der Umsetzung des ASE-Operators wurde der dahinter stehende Algorithmus bisher nur als reine Black Box betrachtet, dessen Funktionalität keine weitere Rolle spielte. Zur Optimierung der Geschwindigkeit durch Ausnutzen der Optimierungsstrategien von Odysseus wäre auch eine Abbildung des kompletten ASE-Algorithmus in Odysseus interessant. Dazu könnten einzelne Berechnungsschritte des Algorithmus durch bereits existierende Ausdrücke im Map-Operator durchgeführt werden. Wahrscheinlich wäre hierbei auch das Umsetzen von neuen Operatoren für einige der Rechnungen nötig. Der dabei umzusetzende Datenfluss ist in Abbildung A.1 auf Seite 87 dargestellt.

A Anhang

A.1 Diagramme

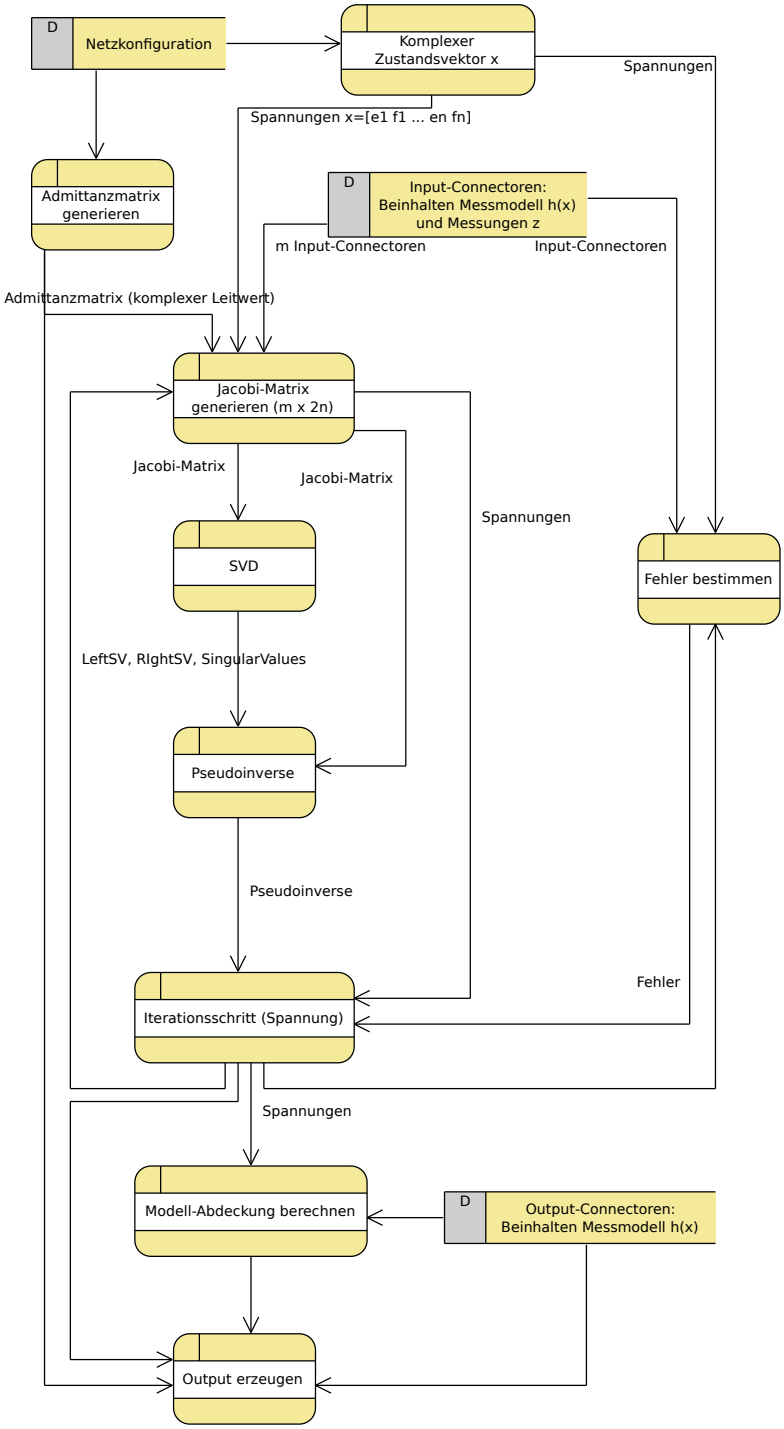


Abbildung A.1: Datenfluss des ASE-Algorithmus

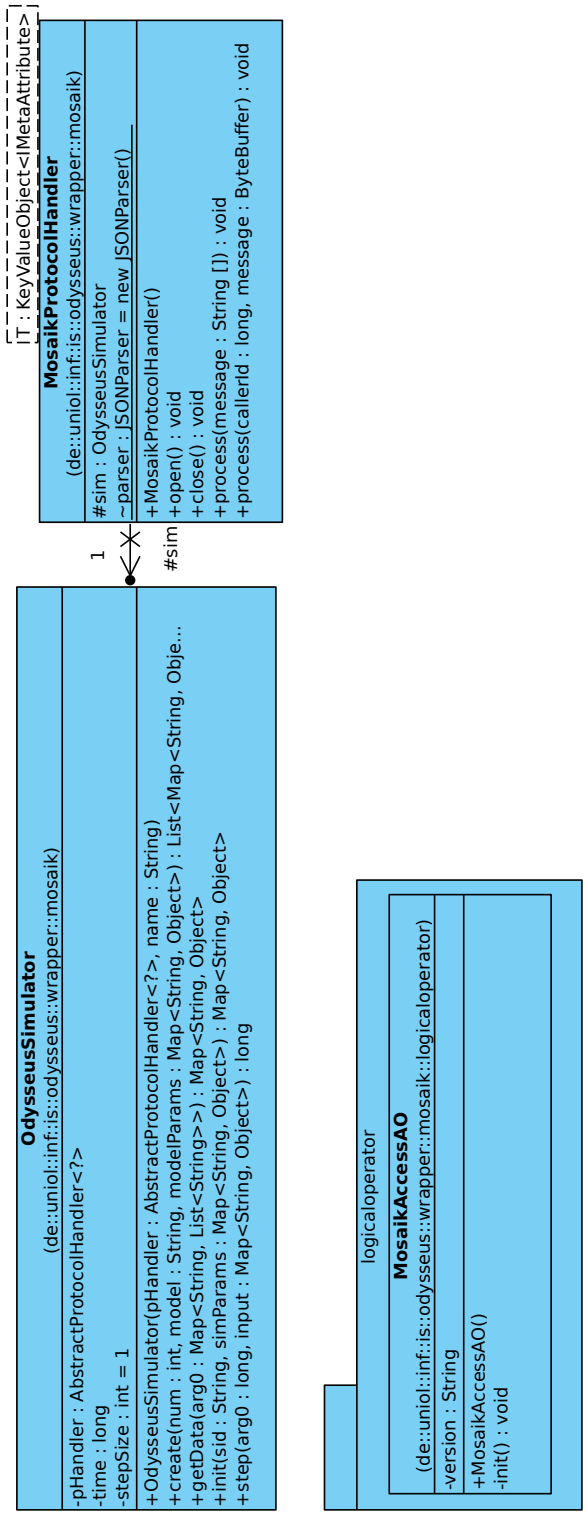


Abbildung A.2: MosaikProtocolHandler

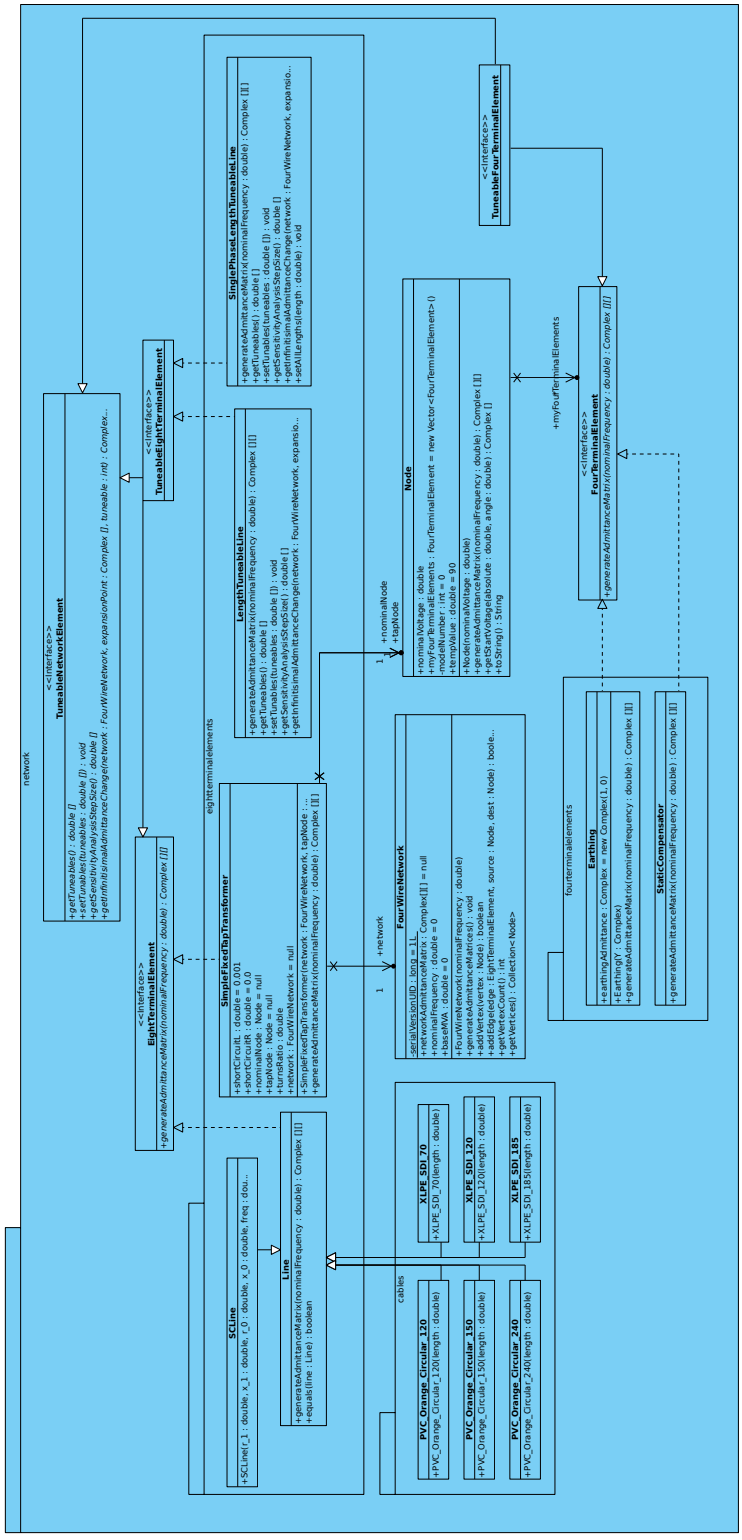


Abbildung A.3: ASE: Leitungen und Knoten – com.smartdist.modelling.network

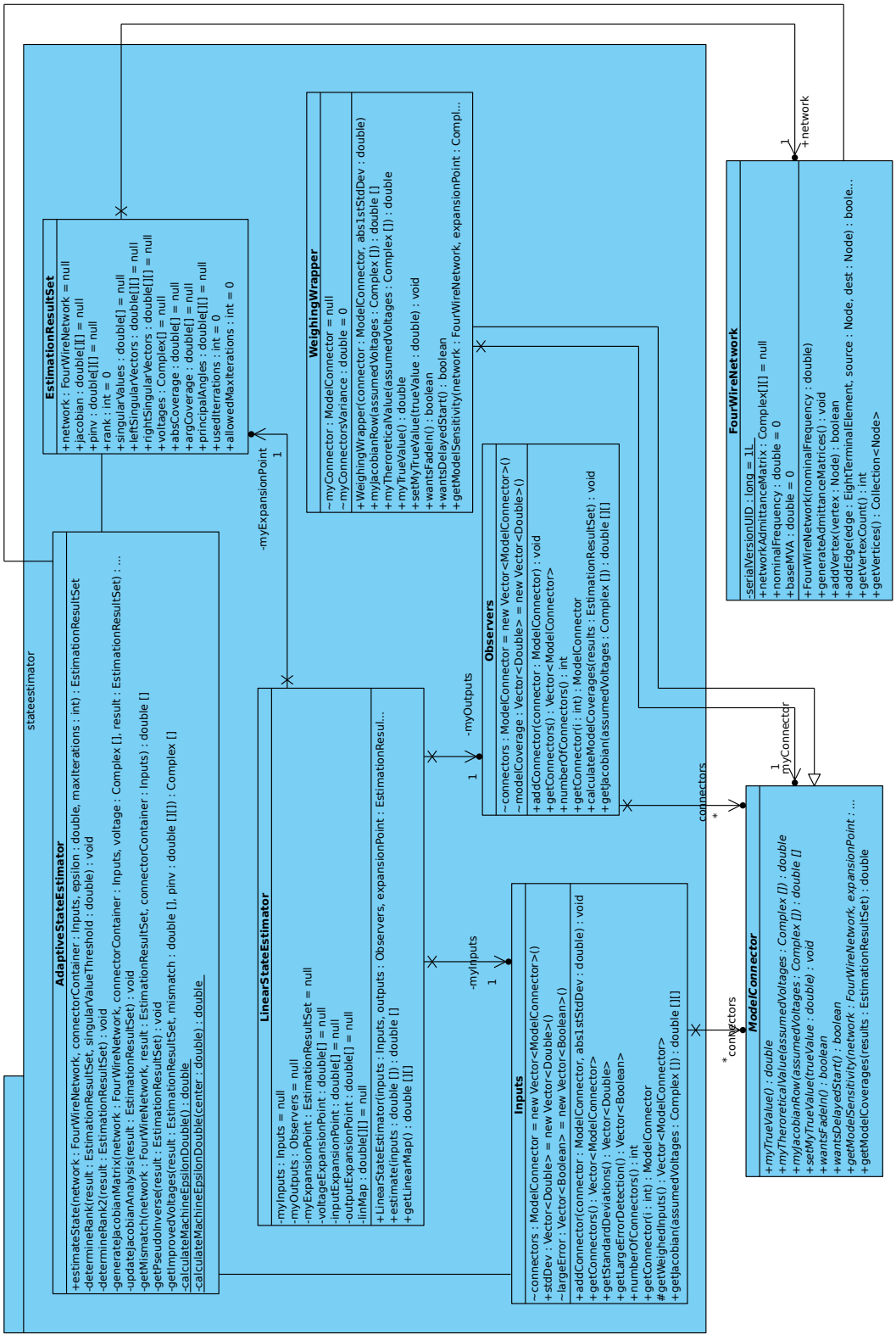


Abbildung A.4: ASE: State Estimatoren – com.smartdist.solvers

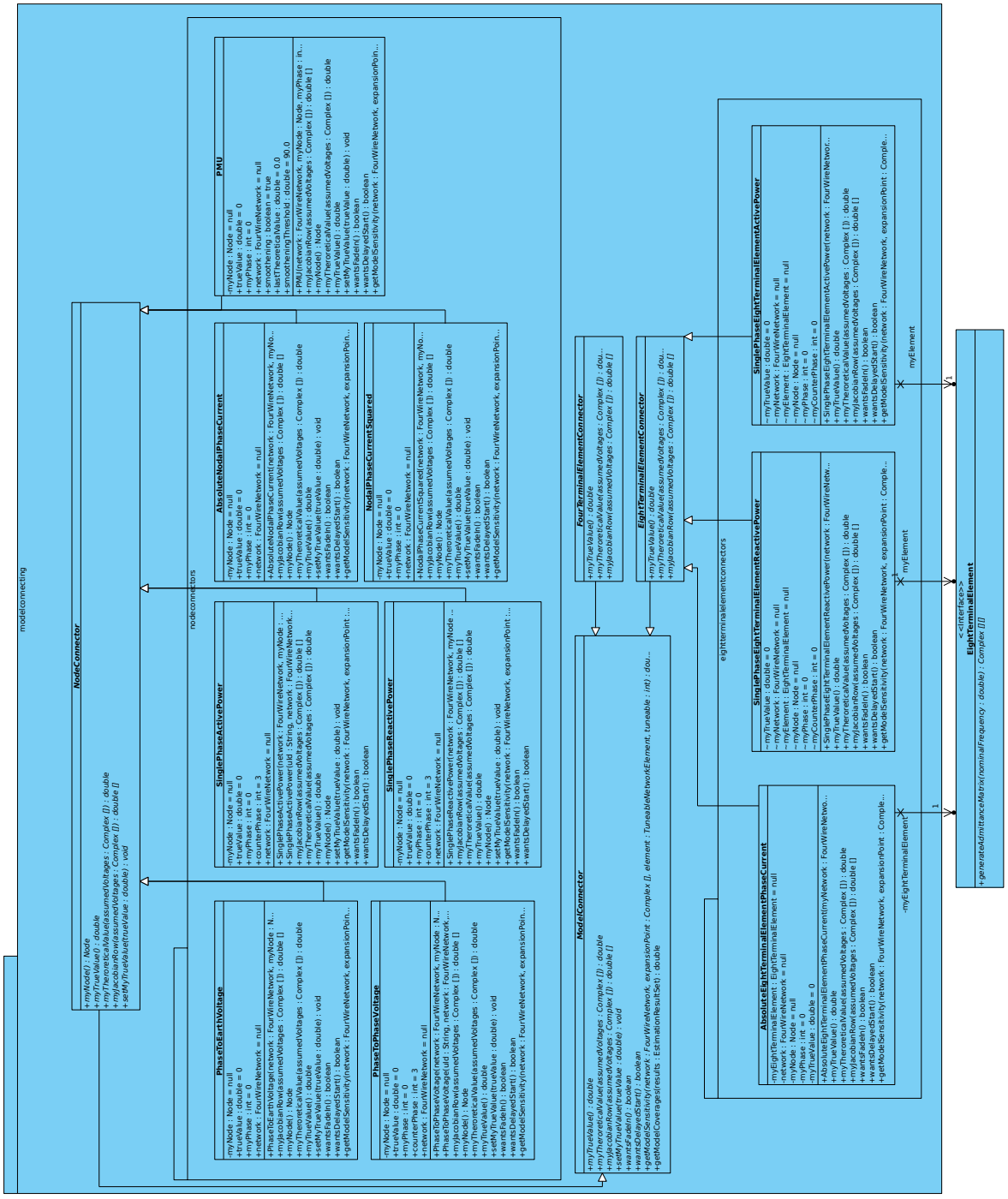


Abbildung A.5: ASE: ModelConnectoren – com.smartdist.modelconnecting

A.2 Programmcode und Beispiele

```
meta = {
    'models': {
        'Socket': {
            'public': True,
            'any_inputs': True,
            'params': ['host', 'port', 'socket_type'],
            'attrs': [],
        },
    },
}

class MosaikZMQ(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(meta)
        self.eid = 'mosaik'
        self.port = 5558
        self.host = 'tcp://*:'

    def init(self, sid, step_size, duration):
        self.sid = sid
        self.step_size = step_size
        self.duration = duration
        return self.meta

    def create(self, num, model, host, port, socket_type, buf_size=1000,
               dataset_opts=None):
        if num != 1 or self.db is not None:
            raise ValueError('Can only create one zeromq socket.')
        if model != 'Socket':
            raise ValueError('Unknown model: "%s"' % model)

        self.context = zmq.Context()
        if socket_type == 'PUSH':
            self.sender = self.context.socket(zmq.PUSH)
        elif socket_type == 'PUB':
            self.sender = self.context.socket(zmq.PUB)
        else:
            raise ValueError('Unknown socket type. Allowed are PUSH and
                PUB')

        self.sender.bind(host + str(port))
        return [{'eid': self.eid, 'type': model, 'rel': []}]

    def step(self, time, inputs):
        assert len(inputs) == 1
        inputs.update({'timestamp': time})
        self.sender.send_json(inputs)
        return time + self.step_size

    def get_data(self, outputs):
```

```
49          raise RuntimeError('mosaik_zmq does not provide any outputs.')
50
51  def main():
52      desc = __doc__.strip()
53      mosaik_api.start_simulation(MosaikZMQ(), desc)
```

Listing A.1: mosaik-zmq Simulator

```
sim_config = {
    'CSV': {'python': 'mosaik_csv:CSV'},
    'ZMQ': {'cmd': 'mosaik-zmq %(addr)s'},
    'Odysseus': {'connect': '127.0.0.1:5554'},
    'HouseholdSim': {'python': 'householdsim.mosaik:HouseholdSim'},
    'PyPower': {'python': 'mosaik_pypower.mosaik:PyPower'},
}
START = '2014-01-01 00:00:00'
END = 31 * 24 * 3600  # 1 month
PV_DATA = 'data/pv_10kw.csv'
PROFILE_ASE_FILE = 'data/profiles.ase.data.gz'
GRID_ASE_NAME = 'demo_ase_lv_grid'
GRID_ASE_FILE = 'data/%s.json' % GRID_ASE_NAME

def main():
    random.seed(23)
    world = mosaik.World(sim_config)
    create_scenario(world)
    world.run(until=END)  # As fast as possilbe

def create_scenario(world):
    pypower = world.start('PyPower', step_size=60)
    hhsim = world.start('HouseholdSim')
    pvsim = world.start('CSV', sim_start=START, datafile=PV_DATA)
    # one of the follwing models has to be commented out
    odysseusModel = world.start('ZMQ', step_size=60, duration=END)
    ody = odysseusModel.Socket(host='tcp://*:', port=5558, socket_type='
        PUB')
    odysseusModel = world.start('Odysseus', step_size=60, duration=END)
    odysseus = odysseusModel.Odysseus.create(1)
    ody = odysseus[0]
    # Instantiate models
    grid = pypower.Grid(gridfile=GRID_ASE_FILE).children
    houses = hhsim.ResidentialLoads(sim_start=START,
                                    profile_file=PROFILE_ASE_FILE,
                                    grid_name=GRID_ASE_NAME).children
    pvs = pvsim.PV.create(4)
    connect_buildings_to_grid(world, houses, grid)
    connect_randomly(world, pvs, [e for e in grid if 'node' in e.eid], 'P
        ')
    # Odysseus
    connect_many_to_one(world, houses, ody, 'P_out')
    connect_many_to_one(world, pvs, ody, 'P')
    nodes = [e for e in grid if e.type in ('RefBus, PQBus')]
    connect_many_to_one(world, nodes, ody, 'P', 'Q', 'Vl', 'Vm', 'Va')
    branches = [e for e in grid if e.type in ('Transformer', 'Branch')]
    connect_many_to_one(world, branches, ody,
                        'P_from', 'Q_from', 'P_to', 'P_from')
```

Listing A.2: Definition eines mosaik-Szenarios

```
{'mosaik': {
   'P_out': {
      'HouseholdSim-0.House_1': 39.94,
      'HouseholdSim-0.House_0': 204.96
   },
   'P_to': {
      'PyPower-0.0-branch_1': -244.86986834585113,
      'PyPower-0.0-transformer': -244.8999990667926,
      'PyPower-0.0-branch_2': -39.9340707074282
   },
   'Vm': {
      'PyPower-0.0-tr_pri': 20000.0,
      'PyPower-0.0-tr_sec': 229.9999996722332,
      'PyPower-0.0-node1': 229.97293289832015,
      'PyPower-0.0-node2': 229.96851866005926
   },
   'P_from': {
      'PyPower-0.0-branch_1': 244.89868826737285,
      'PyPower-0.0-transformer': 244.899999415793,
      'PyPower-0.0-branch_2': 39.9348372317444
   }, 'P': {
      'PyPower-0.0-tr_pri': 244.899999415793,
      'PyPower-0.0-tr_sec': 0.0,
      'PyPower-0.0-node1': 204.95999999999998,
      'PyPower-0.0-node2': 39.94
   }, 'Q': {
      'PyPower-0.0-tr_pri': 0.0,
      'PyPower-0.0-tr_sec': 0.0,
      'PyPower-0.0-node1': 0.0,
      'PyPower-0.0-node2': 0.0
   }, 'Q_from': {
      'PyPower-0.0-branch_1': 0.00870350737994637,
      'PyPower-0.0-transformer': 0.0,
      'PyPower-0.0-branch_2': 0.001583079476607352
   }, 'Vl': {
      'PyPower-0.0-tr_pri': 20000.0,
      'PyPower-0.0-tr_sec': 230.0,
      'PyPower-0.0-node1': 230.0,
      'PyPower-0.0-node2': 230.0
   }, 'Va': {
      'PyPower-0.0-tr_pri': 0.0,
      'PyPower-0.0-tr_sec': -3.117571196747344e-07,
      'PyPower-0.0-node1': -0.002133586615704701,
      'PyPower-0.0-node2': -0.002481495957688719
   }
}}
```

Listing A.3: Von mosaik gesendete Daten

```
{'edges': [
   {'type': 'SCLine', 'node1': 'tr_sec', 'node2': 'node1', 'length': 0.1,
       'resistance': 0.2542, 'reactance': 0.080425},
   {'type': 'SCLine', 'node1': 'node1', 'node2': 'node2', 'length': 0.1,
      'resistance': 0.2542, 'reactance': 0.080425},
   {'type': 'SCLine', 'node1': 'node2', 'node2': 'node3', 'length': 0.1,
      'resistance': 0.2542, 'reactance': 0.080425},
   {'type': 'SCLine', 'node1': 'node3', 'node2': 'node4', 'length': 0.1,
      'resistance': 0.2542, 'reactance': 0.080425}
   ],
'nodes': [
   {'name': 'tr_sec', 'initialVoltage': 230.0},
   {'name': 'node1', 'initialVoltage': 230.0},
   {'name': 'node2', 'initialVoltage': 230.0},
   {'name': 'node3', 'initialVoltage': 230.0},
   {'name': 'node4', 'initialVoltage': 230.0}
   ],
'inputConnectors': [
   {'type': 'PhaseToEarthVoltage', 'node': 'tr_sec', 'ownPhase': 'PhaseA'
      , 'stdDev': 1.0, 'attribute': 'odysseus_0.Vm.PyPower-0.0-tr_sec'},
   {'type': 'PMU', 'node': 'tr_sec', 'ownPhase': 'PhaseA', 'stdDev': 1.0,
       'attribute': 'odysseus_0.Va.PyPower-0.0-tr_sec'},
   {'type': 'SinglePhaseActivePower', 'node': 'node1', 'ownPhase': '
      PhaseA', 'stdDev': 1.0, 'attribute': 'odysseus_0.P.PyPower-0.0-
      node1'},
   {'type': 'SinglePhaseReactivePower', 'node': 'node1', 'ownPhase': '
      PhaseA', 'stdDev': 1.0, 'attribute': 'odysseus_0.Q.PyPower-0.0-
      node1'},
   {'type': 'SinglePhaseActivePower', 'node': 'node2', 'ownPhase': '
      PhaseA', 'stdDev': 1.0, 'attribute': 'odysseus_0.P.PyPower-0.0-
      node2'},
   {'type': 'SinglePhaseReactivePower', 'node': 'node2', 'ownPhase': '
      PhaseA', 'stdDev': 1.0, 'attribute': 'odysseus_0.Q.PyPower-0.0-
      node2'},
   {'type': 'SinglePhaseActivePower', 'node': 'node3', 'ownPhase': '
      PhaseA', 'stdDev': 1.0, 'attribute': 'odysseus_0.P.PyPower-0.0-
      node3'},
   {'type': 'SinglePhaseReactivePower', 'node': 'node3', 'ownPhase': '
      PhaseA', 'stdDev': 1.0, 'attribute': 'odysseus_0.Q.PyPower-0.0-
      node3'}
   ],
'outputConnectors': [
   {'type': 'PhaseToEarthVoltage', 'node': 'node1', 'ownPhase': 'PhaseA',
       'stdDev': 1.0, 'visualisationType': 'coverage'},
   {'type': 'PhaseToEarthVoltage', 'node': 'node2', 'ownPhase': 'PhaseA',
       'stdDev': 1.0, 'visualisationType': 'voltage'}
   ]
}
```

Listing A.4: JSON-Konfiguration von Szenario 1

```
#PARSER PQL
#METADATA Latency
#METADATA TimeInterval
#DEFINE delay 1000000 ///1 mio = 1 MB
#DEFINE measureEach 1000
#DEFINE evalPath ${WORKSPACEPROJECT}/ASETest/Evaluation
#DEFINE projectName scenario1LatenzRCPohneWindow
#RUNQUERY
mosaikInput = MOSAIK({source = 'MosaikReceiver',   type = 'simapi'})

calcLatency1 = CALCLATENCY(mosaikInput)
writeLatency1 = SENDER({transport='File', wrapper='GenericPush',
   protocol='JSON', dataHandler='KeyValueObject', sink='writeLatency1',
   options=[
      ['filename','${evalPath}/${projectName}/${NOW}latency1.json'],
      ['json.write.metadata','true'],
      ['writedelaysize', '${delay}']
   ]}, calcLatency1)

tuples1 = KEYVALUETOTUPLE({ SCHEMA = [
      ['odysseus_0.Q.PyPower-0.0-tr_sec', 'Double'],
      ['odysseus_0.P.PyPower-0.0-tr_sec', 'Double'],
      ['odysseus_0.Va.PyPower-0.0-tr_sec', 'Double'],
      ['odysseus_0.Vm.PyPower-0.0-tr_sec', 'Double'],
      ['odysseus_0.P.PyPower-0.0-node1', 'Double'],
      ['odysseus_0.Q.PyPower-0.0-node1', 'Double'],
      ['odysseus_0.Va.PyPower-0.0-node1', 'Double'],
      ['odysseus_0.Vm.PyPower-0.0-node1', 'Double'],
      ['odysseus_0.P.PyPower-0.0-node2', 'Double'],
      ['odysseus_0.Q.PyPower-0.0-node2', 'Double'],
      ['odysseus_0.Va.PyPower-0.0-node2', 'Double'],
      ['odysseus_0.Vm.PyPower-0.0-node2', 'Double'],
      ['odysseus_0.P.PyPower-0.0-node3', 'Double'],
      ['odysseus_0.Q.PyPower-0.0-node3', 'Double'],
      ['odysseus_0.Va.PyPower-0.0-node3', 'Double'],
      ['odysseus_0.Vm.PyPower-0.0-node3', 'Double'],
      ['odysseus_0.P.PyPower-0.0-node4', 'Double'],
      ['odysseus_0.Q.PyPower-0.0-node4', 'Double'],
      ['odysseus_0.Va.PyPower-0.0-node4', 'Double'],
      ['odysseus_0.Vm.PyPower-0.0-node4', 'Double'],
      ['timestamp', 'STARTTIMESTAMP']
   ],
   KEEPINPUT = false,
   TYPE = 'mosaik'},
   calcLatency1)

calcLatency2 = CALCLATENCY(tuples1)
writeLatency2 = SENDER({
   transport='File',
   wrapper='GenericPush',
   protocol='CSV',
   dataHandler='Tuple',
```

```
53    sink='writeLatency2',
54    options=[
55       ['filename','${evalPath}/${projectName}/${NOW}latency2.csv'],
56       ['csv.writeMetadata', 'true'],
57       ['writedelaysize', '${delay}']
58    ]}, calcLatency2)
59
60 mosaikTuples = RENAME({aliases = [
61    'tr_sec_Q', 'tr_sec_P', 'tr_sec_Vang',
62    'tr_sec_V',    'node1_P', 'node1_Q', 'node1_Vang', 'node1_V',
63    'node2_P',  'node2_Q', 'node2_Vang', 'node2_V', 'node3_P',
64    'node3_Q', 'node3_Vang',   'node3_V', 'node4_P', 'node4_Q',
65    'node4_Vang', 'node4_V', 'timestamp'
66    ]}, calcLatency2)
67
68 tuples = MAP({EXPRESSIONS = [
69    'tr_sec_V', 'tr_sec_Vang', 'node1_P', 'node1_Q', 'node2_P',
70    'node2_Q', 'node3_P', 'node3_Q',    'node4_P', 'node4_Q'
71    ]}, mosaikTuples)
72
73 calcLatency3 = CALCLATENCY(tuples)
74 writeLatency3 = SENDER({transport='File', wrapper='GenericPush',
75    protocol='CSV', dataHandler='Tuple', sink='writeLatency3',
76    options=[
77       ['filename','${evalPath}/${projectName}/${NOW}latency3.csv'],
78       ['csv.writeMetadata', 'true'],
79       ['writedelaysize', '${delay}']
80    ]}, calcLatency3)
81
82 aseResult = ASE({
83    topologyFile = '${WORKSPACEPROJECT}/ASETest/TestScenario1/Scenario1.
          json'
84    }, calcLatency3)
85
86 writeLatency5 = SENDER({transport='File', wrapper='GenericPush',
87    protocol='CSV', dataHandler='Tuple', sink='writeLatency5',
88    options=[
89       ['filename','${evalPath}/${projectName}/${NOW}latency5.csv'],
90       ['csv.writeMetadata', 'true'],
91       ['writedelaysize', '${delay}']
92    ]},  CALCLATENCY(aseResult))
93 throughput6 = THROUGHPUT({EACH = ${measureEach},
94    FILENAME = '${evalPath}/${projectName}/${NOW}throughput5.csv'
95    }, aseResult)
```

Listing A.5: Anfrage zur Latenzmessung in PQL

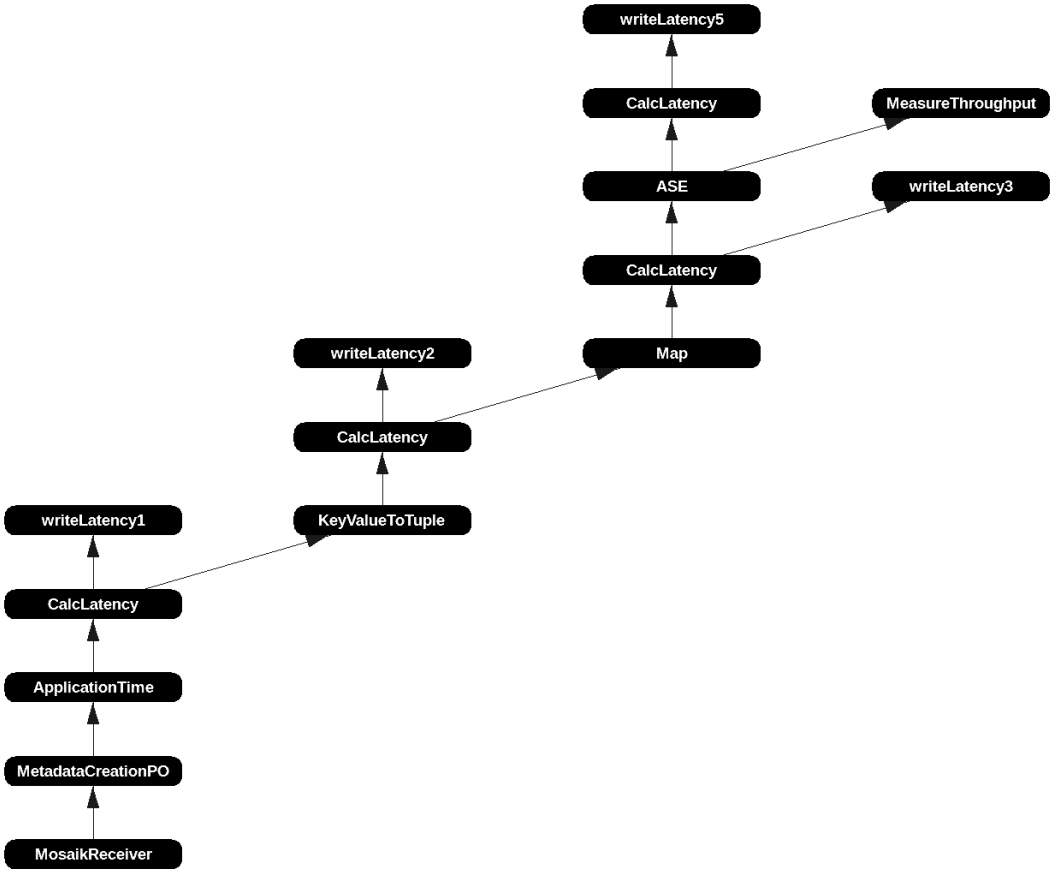


Abbildung A.6: Anfrageplan zur Evaluation der Latenz

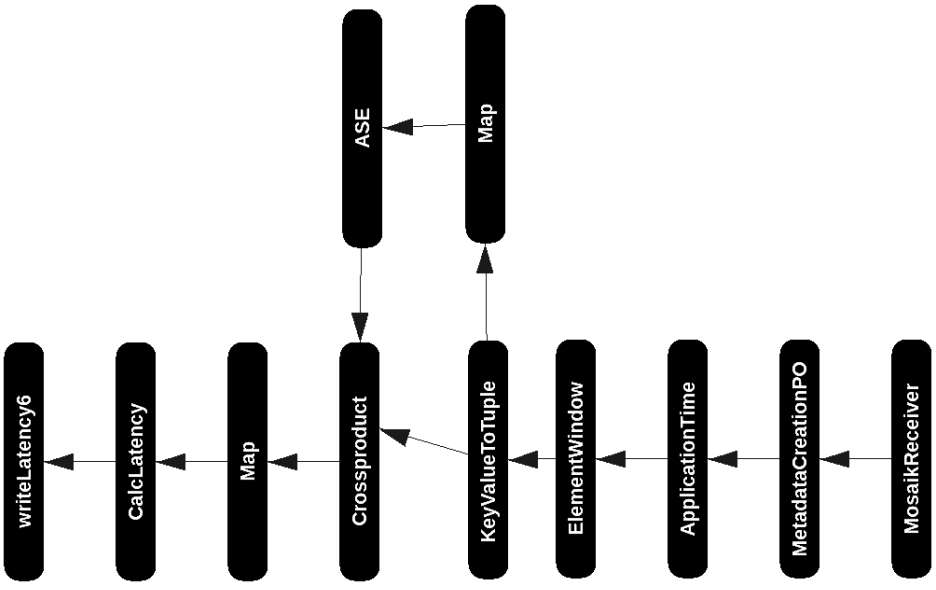


Abbildung A.7: Anfrageplan zur Evaluation der Korrektheit

A.3 Anleitung der Verwendung

Die folgende Anleitung beschreibt kurz, wie sich die im Rahmen dieser Arbeit beschriebenen Systeme verwenden lassen. Weitere Informationen bieten die ausführlichen Dokumentationen von mosaik und Odysseus.

A.3.1 mosaik

Mosaik kann über Bitbucket[1] heruntergeladen werden und entsprechend der Installationsanleitung[2] installiert werden. Zur Verwendung des mosaik-zmq Simulators ist dessen zusätzliche Installation erforderlich. Das dazu nötige Python-Package liegt im Odysseus-SVN im Verzeichnis „Odysseus/wrapper/mosaik/mosaik-zmq-simulator/“ und kann über „pip install mosaik-zmq-0.1.tar.gz“ installiert werden. Eine direkte Bereitstellung über Bitbucket ist geplant, jedoch noch nicht abgeschlossen. Der MosaikProtocolHandler kann direkt verwendet werden ohne vorherige Installationen. Das Einbinden der Simulatoren wurde bereits in Abschnitt 4.1.5 beschrieben und ein Beispiel für eine Szenariodefinition in mosaik ist in Listing A.2 zu sehen.

A.3.2 Odysseus

Odysseus kann aus dem SVN-Repository[3] unter Verwendung des Benutzernamens *lesend* mit dem Passwort *rurome48* ausgecheckt werden. Weitere Informationen bietet die Installationsanleitung[4]. Für die Verwendung des ASE-Operators wird jedoch ein Benutzerkonto mit erweiterten Rechten benötigt, da der Ordner zugriffsbeschränkt ist. Nachdem Odysseus in Eclipse geladen wurde, sollte zum Start von Odysseus das *ASE-Odysseus.product* verwendet werden, welches alle benötigten Abhängigkeiten beinhaltet. Anschließend können in Odysseus Anfragen erstellt werden, wie z. B. in Listing A.5 zu sehen.

[1] `https://bitbucket.org/mosaik`
[2] `https://mosaik.offis.de/install/`
[3] `http://isdb1.offis.uni-oldenburg.de/repos/odysseus/trunk/`
[4] `http://odysseus.informatik.uni-oldenburg.de:8090/display/ODYSSEUS/Development+with+Odysseus`

Glossar

Nachfolgend sind wesentliche Begriffe dieser Arbeit kurz zusammengefasst und erläutert. Eine ausführliche Erklärung findet sich jeweils in den einführenden Abschnitten sowie der darin angegebenen Literatur. Das im Rahmen der Erläuterung verwendete Symbol $\sim$ bezieht sich jeweils auf den im Einzelnen vorgestellten Begriff, das Symbol ↑ verweist auf einen ebenfalls innerhalb dieses Glossars erklärten Begriff.

Ableitwiderstand Der $\sim$ stellt Verlustströme, wie z.B. durch Koronaentladungen, Kriechströme und mangelhafte Isolation, dar, die vor allem bei einem Betrieb mit hoher Spannung auftreten können.

Adaptive State Estimation Verfahren zur Bestimmung des Zustands eines Energienetzes, welches auf dem ↑Newton-Raphson-Verfahren und der ↑State Estimation basiert. Im Vergleich zu diesen kann es auch mit unter- und überbestimmten Netzen arbeiten.

Admittanz Bei der $\sim$ handelt es sich um den Scheinleitwert, der als komplexe Zahl aus der ↑Konduktanz als Real- und der ↑Suszeptanz als Imaginärteil besteht.

Admittanzmatrix Die $\sim$ ist eine Matrix, welche die ↑Admittanzen der Leitungen zwischen verschiedenen Knoten des Energienetzes beinhaltet.

Anfrageplan In ↑Odysseus wird zunächst ein logischer $\sim$ erstellt, der noch keine Algorithmen enthält. Durch die ↑Transformationskomponente wird dieser dann in einen physischen $\sim$ überführt, der in Form der ↑physischen Operatoren auch die Algorithmen enthält.

ASE-Algorithmus Der in Java implementierte Algorithmus der ↑Adaptive State Estimation, welcher für diese Arbeit bereitgestellt wurde.

ASE-Berechnung Mit $\sim$ ist allgemein der Vorgang der Berechnung des Netzes durch die ↑Adaptive State Estimation gemeint.

ASE-Operator Der $\sim$ ist ein ↑Physischer Operator, welcher den ↑ASE-Algorithmus in ↑Odysseus einbindet.

Ausführungskomponente Die $\sim$ setzt eine einheitliche Ausführung der ↑Anfragepläne in ↑Odysseus um. Dabei werden die ↑physischen Operatoren verknüpft und die Unabhängigkeit von bestimmten Datenmodellen wird durch Generics und das Strategie-Muster eingehalten.

Blindleistung Nicht nutzbare elektrische Leistung, welche durch induktive und kapazitive Effekte der Leitungen entsteht.

DataHandler Ein $\sim$ in ↑Odysseus erstellt aus einem primitiven Objekt, wie z. B. einem String oder ByteBuffer, ein bestimmtes Datenstromobjekt wie z.B. ↑Tupel oder ↑Key-Value-Objekt.

Datenstrom Als $\sim$ wird eine Abfolge von kontinuierlich erzeugten Daten bezeichnet, deren Menge im Vornherein nicht bekannt ist und daher als unendlich angesehen wird.

Datenstrommanagementsystem Ein $\sim$ ist ein Softwaresystem, das zur Verwaltung von ↑Datenströmen eingesetzt wird. ↑Anfragen werden dabei kontinuierlich ausgeführt und liefern einen Datenstrom als Ergebnis zurück.

ForceAtlas2-Algorithmus Der $\sim$ ist ein kräftebasierter Layouting-Algorithmus zur Anordnung von Knoten in einem Graph, der im ↑Gephi-Toolkit enthalten ist.

Gephi-Toolkit Eine frei verfügbare Sammlung von Werkzeugen zum Umgang mit Graphen in der Programmiersprache Java.

HDF5 Ein Datenformat zur Speicherung von großen Datenmengen, welches gerade in der Energiedomäne häufig Verwendung findet.

Induktiver Blindwiderstand Der induktive Blindwiderstand bildet die Verluste durch Magnetfelder anderer Leiter und Selbstinduktion ab.

Jacobi-Matrix Die ∼ einer differenzierbaren Funktion beinhaltet ihre sämtlichen partiellen Ableitungen.

Kapazitiver Blindwiderstand Der ∼ beschreibt Lade- und Entlade-Vorgänge der Leitungen, die darauf beruhen, dass die Leiter sich wie Kondensatoren verhalten.

Key-Value-Objekt Datenstromelement in ↑Odysseus, welches die Daten in Form von Key-Value-Paaren beinhaltet.

Komplexer Leitwert ↑Admittanz

Konduktanz ↑Ableitwiderstand

Kontinuierliche Anfrage Eine ∼ wird an ein ↑Datenstrommanagementsystem gestellt und liefert als Ergebnis eine Teilmenge des zugrundeliegenden Informationsbestandes. Sie wird durchgehend ausgeführt und liefert einen ↑Datenstrom als Ergebnis.

Leistungsflussrechnung Die ∼ berechnet aus den bekannten Knotenleistungen eines Energienetzes deren Knotenspannungen.

Logischer Anfrageplan siehe ↑Anfrageplan

Logischer Operator Ein ∼ ist ein Operator, der im logischen ↑Anfrageplan von ↑Odysseus verwendet wird und noch keinen Algorithmus zur Durchführung der gewünschten Operation beinhaltet.

Newton-Raphson-Verfahren Effizientes iteratives Verfahren der ↑Leistungsflussrechnung

Odysseus ∼ ist ein an der Universität Oldenburg entwickeltes ↑Datenstrommanagementsystem.

OSGi ∼ ist eine Spezifkation für dynamische Softwareplattformen, die das System in verschiedene Plug-Ins bzw. Bundles unterteilt. Das Hinzufügen, Aktualisieren und Entfernen von Bundles ist auch im laufenden System möglich.

Physischer Anfrageplan siehe ↑Anfrageplan

Physischer Operator Ein ∼ ist ein Operator, der im physischen ↑Anfrageplan verwendet wird und den Algorithmus zur Durchführung der gewünschten Operation enthält.

PQL ∼ ist eine für ↑Odysseus entwickelte Anfragesprache (vgl. Abschnitt 2.4.4).

Pseudoinverse Die Pseudoinverse ist eine verallgemeinerte Inverse, die selbst für nichtquadratische und singuläre Matrizen existiert, welche nicht direkt invertierbar sind.

ProtocolHandler Der ∼ wird in ↑Odysseus verwendet, um verschiedene Protokolle zum Senden und Empfangen von Daten umzusetzen.

Query-Sharing Beim ∼ teilen sich mehrere ↑Anfragen Operatoren, die in ihren ↑Anfrageplänen auf gleiche Art und Weise verwendet werden.

Reaktanz ↑Induktiver Blindwiderstand

Resistenz ↑Wirkwiderstand

Restrukturierungskomponente Die ∼ wendet Restrukturierungsregeln zur Optimierung der logischen ↑Anfragepläne an.

Scheinleitwert ↑Admittanz

State Estimation Die ∼ versucht anhand aller zur Verfügung stehender Daten den Zustand des Energienetzes möglichst exakt zu bestimmen. Dabei versucht sie fehlerhafte Messdaten zu identifizieren und soweit möglich zu korrigieren.

Suszeptanz ↑Kapazitiver Blindwiderstand

Transformationskomponente In der ∼ werden mit Hilfe von ↑Transformationsregeln ↑logische Operatoren in ↑physische Operatoren umgewandelt.

Transformationsregeln ∼ werden in der ↑Transformationskomponente angewendet, um aus ↑logischen Operatoren ↑physische Operatoren zu erstellen.

TransportHandler Der ∼ ist für den Empfang der Daten in ↑Odysseus zuständig und kann dabei mit verschiedenen Typen umgehen, wie z.B. TCP, RS232 oder HTTP.

Tupel Datenstromelement in ↑Odysseus, welches einem festen Schema entsprechende Daten beinhaltet.

Übersetzungskomponente In ↑Odysseus sorgt die ∼ für die Erstellung der logischen ↑Anfragepläne aus Anfragen in verschiedenen Sprachen, wie z. B. ↑PQL.

Wirkleistung Die tatsächlich nutzbare elektrische Leistung in Watt [W].

Wirkwiderstand Der ∼ ist der elektrische Widerstand eines Leiters, die abhängig von Material, Temperatur, Länge und Querschnitt ist.

Abkürzungen

AMQP	Advanced Message Queuing Protocol
ASE	Adaptive State Estimation
CQL	Continuous Query Language
DBMS	Datenbankmanagementsystem
DEBS	Conference on Distributed Event Based Systems
DSMS	Datenstrommanagementsystem
HDF5	Hierarchical Data Format 5
HWM	High-Water Mark
LMSE	Least Mean Square Error
MoSL	Mosaik Specification Language
MSE	Mean Square Error
NR-Verfahren	Newton-Raphson-Verfahren
ONB	Orthonormalbasis
OSGi	Open Services Gateway initiative
OSI	Open System Interconnection
PMU	Phasor Measurement Unit
PQL	Procedural Query Language
RDF	Resource Description Framework
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SCADA	Supervisory Control and Data Aquisition
SQL	Structured Query Language
SVD	Singulärwertzerlegung
SVN	Subversion
TCP	Transmission Control Protocol
TS	Zeitstempel (timestamp)
ZMQ	ZeroMQ

Abbildungen

Tabellen

Listings

Literatur

[AAA+08] AIYAGARI, Sanjay ; ARROTT, Matthew ; ATWELL, Mark ; BROME, Jason ; CONWAY, Alan ; GODFREY, Robert ; GREIG, Robert ; HINTJENS, Pieter ; O'HARA, John ; RADESTOCK, Matthias ; RICHARDSON, Alexis ; RITCHIE, Martin ; SADJADI, Shahrokh ; SCHLOMING, Rafael ; SHAW, Steven ; SUSTRIK, Martin ; TRIELOFF, Carl ; RIET, Kim van d. ; VINOSKI, Steve: *AMQP - Advanced Message Queuing Procotol - Protocol Specification*. Nov 2008. – `http://www.amqp.org/specification/0-9-1/amqp-org-download`

[AAB+05] ABADI, Daniel J. ; AHMAD, Yanif ; BALAZINSKA, Magdalena ; CHERNIACK, Mitch ; HWANG, Jeong hyon ; LINDNER, Wolfgang ; MASKEY, Anurag S. ; RASIN, Er ; RYVKINA, Esther ; TATBUL, Nesime ; XING, Ying ; ZDONIK, Stan: The design of the borealis stream processing engine. In: *In CIDR*, 2005, S. 277–289

[ABB+04] ARASU, A. ; BABCOCK, B. ; BABU, S. ; CIESLEWICZ, J. ; DATAR, M. ; ITO, K. ; MOTWANI, R ; SRIVASTAVA, U. ; WIDOM, J.: STREAM: The Stanford Data Stream Management System / Stanford InfoLab. Version: 2004. `http://ilpubs.stanford.edu:8090/641/`. Stanford InfoLab, 2004 (2004-20). – Technical Report

[AE04] ABUR, A. ; EXPÓSITO, A.G.: *Power System State Estimation: Theory and Implementation*. Taylor & Francis, 2004 (Power Engineering (Willis)). `http://books.google.co.in/books?id=NQhbtFC6_40C`. – ISBN 9780203913673

[AGG+12] APPELRATH, H.-Jürgen ; GEESEN, Dennis ; GRAWUNDER, Marco ; MICHELSEN, Timo ; NICKLAS, Daniela: Odysseus: A Highly Customizable Framework for Creating Efficient Event Stream Management Systems. In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. New York, NY, USA : ACM, 2012 (DEBS '12). – ISBN 978–1–4503–1315–5, 367–368

[BGJ+09] BOLLES, Andre ; GRAWUNDER, Marko ; JACOBI, Jonas ; NICKLAS, Daniela ; APPELRATH, H.-Jürgen: Odysseus: Ein Framework für maßgeschneiderte Datenstrommanagementsysteme. In: *Informatik 2009*, Workshop Verwaltung, Analyse und Bereitstellung kontextbasierter Informationen, 2009

[Bü13] BÜSCHER, Martin: *Konzeption und Evaluierung eines Verfahrens zur optimierten Messstellenanordnung im Energieversorgungsnetz*, Carl von Ossietzky University Oldenburg, Diplomarbeit, Februar 2013

[CCD+03] CHANDRASEKARAN, Sirish ; COOPER, Owen ; DESHPANDE, Amol ; FRANKLIN, Michael J. ; HELLERSTEIN, Joseph M. ; HONG, Wei ; KRISHNAMURTHY, Sailesh ; MADDEN, Samuel ; RAMAN, Vijayshankar ; REISS, Frederick ; SHAH, Mehul A.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: *CIDR*, 2003

[CD06] CLINE, Alan K. ; DHILLON, Inderjit S .. ; HOGBEN, Leslie (Hrsg.): *Handbook of Linear Algebra*. Berlin Heidelberg : Chapman and Hall/CRC, 2006

[Ecl15] ECLIPSE: *Eclipse PDE/User Guide*. Website, Jan 2015. – `http://wiki.eclipse.org/PDE/User_Guide`; Letzter Zugriff: 02.01.2015.

[GO03] GOLAB, Lukasz ; ÖZSU, M. T.: Issues in Data Stream Management. In: *SIGMOD Rec.*

32 (2003), Juni, Nr. 2, 5–14. http://dx.doi.org/10.1145/776985.776986. – DOI 10.1145/776985.776986. – ISSN 0163–5808

[GO10] GOLAB, Lukasz ; ÖZSU, M. T.: *Data Stream Management (Synthesis Lectures on Data Management)*. Morgan & Claypool Publishers, 2010

[GWC+07] GYLLSTROM, Daniel ; WU, Eugene ; CHAE, Hee-Jin ; DIAO, Yanlei ; STAHLBERG, Patrick ; ANDERSON, Gordon: SASE: Complex Event Processing over Streams. In: *Proceedings of the Third Biennial Conference on Innovative Data Systems Research*, 2007 (CIDR 2007)

[Gü14] GÜNTHER, Hanno: *Entwicklung einer Visualisierungskomponente für das Smart Grid Simulationsframework mosaik*, Universität Oldenburg, bachelor thesis, Januar 2014

[Inf14] *IBM Infosphere Stream Homepage*. Website, Aug 2014. – http://www-03.ibm.com/software/products/en/infosphere-streams; Letzter Zugriff: 27.08.2014.

[JVHB14] JACOMY, Mathieu ; VENTURINI, Tommaso ; HEYMANN, Sebastien ; BASTIAN, Mathieu: ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software. In: *PLoS ONE* 9 (2014), 06, Nr. 6, e98679. http://dx.doi.org/10.1371/journal.pone.0098679. – DOI 10.1371/journal.pone.0098679

[Kal60] KALMAN, R. E.: A New Approach to Linear Filtering and Prediction Problems. In: *Transactions of the ASME – Journal of Basic Engineering* (1960), Nr. 82 (Series D), 35–45. http://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf

[KB13] KNABNER, Peter ; BARTH, Wolf: *Lineare Algebra, Grundlagen und Anwendungen*. Berlin Heidelberg : Springer, 2013

[KL] KRAUSE, Olav ; LEHNHOFF, Sebastian: *Adaptive Power System Static-State Estimation*

[KL12] KRAUSE, Olav ; LEHNHOFF, Sebastian: Generalized static-state estimation. In: *Universities Power Engineering Conference (AUPEC), 2012 22nd Australasian*, 2012, S. 1–6

[KP09] KREUSSLER, Bernd ; PFISTER, Gerhard: *Mathematik für Informatiker*. Berlin Heidelberg : Springer, 2009

[KS04] KRÄMER, Jürgen ; SEEGER, Bernhard: PIPES: a public infrastructure for processing and exploring streams. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 2004 (SIGMOD '04), S. 925–926

[KS09] KRÜGER, Guido ; STARK, Thomas: *Handbuch der Java Programmierung*. München : Addison-Wesley, 2009

[Leh14] LEHNHOFF, Sebastian: *Folien zur Vorlesung Intelligentes Netzmanagement*. Universität Oldenburg, 2014

[mos14] *mosaik 2.0 Dokumentation*. Website, Okt 2014. – http://mosaik.readthedocs.org; Letzter Zugriff: 10.10.2014.

[MW11] MARINESCU, Marlene ; WINTER, Jürgen: *Grundlagenwissen Elektrotechnik, Gleich-,*

Wechsel- und Drehstrom. Vieweg + Teubner, 2011

[OAS12] OASIS: *Advanced Message Queuing Procotol (AMQP) Version 1.0*. Okt 2012. – http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf

[osg11] *OSGi Service Platform Core Specification*. Website, Apr 2011. – http://www.osgi.org/download/r4v43/osgi.core-4.3.0.pdf; Letzter Zugriff: 24.09.2012.

[Pen55] PENROSE, R.: A generalized inverse for matrices. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 51 (1955), Nr. 03, 406–413. http://dx.doi.org/10.1017/s0305004100030401. – DOI 10.1017/s0305004100030401

[Piv14] PIVOTAL: *AMQP 0-9-1 Model Explained*. Website, Nov 2014. – http://www.rabbitmq.com/tutorials/amqp-concepts.html; Letzter Zugriff: 10.11.2014.

[Pos80] POSTEL, J.: *User Datagram Protocol*. RFC 768 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc768.txt. Version: August 1980 (Request for Comments)

[Pos81] POSTEL, J.: *Transmission Control Protocol*. RFC 793 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc793.txt. Version: September 1981 (Request for Comments). – Updated by RFCs 1122, 3168, 6093, 6528

[RLS+13] ROHJANS, S. ; LEHNHOFF, S. ; SCHÜTTE, S. ; SCHERFKE, S. ; HUSSAIN, S.: mosaik - A modular platform for the evaluation of agent-based Smart Grid control. In: *Innovative Smart Grid Technologies Europe (ISGT EUROPE), 2013 4th IEEE/PES*, 2013, S. 1–5

[Sch70] SCHWEPPE, F.C.: Power System Static-State Estimation, Part III: Implementation. In: *Power Apparatus and Systems, IEEE Transactions on* PAS-89 (1970), Jan, Nr. 1, S. 130–135. http://dx.doi.org/10.1109/TPAS.1970.292680. – DOI 10.1109/TPAS.1970.292680. – ISSN 0018–9510

[Sch11] SCHÜTTE, Steffen: A domain-specific language for simulation composition. In: BURCZYNSKI, T. (Hrsg.) ; KOLODZIEJ, J. (Hrsg.) ; BYRSKI, A. (Hrsg.) ; CARVALHO, M. (Hrsg.): *25th European Conference on Modelling and Simulation*. Krakow, 2011, S. 146–152

[Sch12] SCHWAB, Adolf J.: *Elektronenergiesysteme: Erzeugung, Transport, Übertragung und Verteilung elektrischer Energie*. 3., neu bearb. und erw. Aufl. Berlin : Springer, 2012

[Sch14] SCHÜTTE, Steffen: *Simulation Model Composition for the Large-Scale Analysis of Smart Grid Control Mechanisms*, Carl von Ossietzky University Oldenburg, Diss., 2014

[See04] SEEGER, Bernhard: Datenströme. In: *Datenbank-Spektrum* (2004), September, S. 30–33

[SFR99] STEVENS, W. R. ; FENNER, Bill ; RUDOFF, Andrew M.: *UNIX Network Programming, Vol. 2 - Interprocess Communications*. 2. Prentice Hall, 1999

[SH74] SCHWEPPE, F.C. ; HANDSCHIN, E.J.: Static state estimation in electric power systems. In: *Proceedings of the IEEE* 62 (1974), July, Nr. 7, S. 972–982. http://dx.doi.org/10.1109/PROC.1974.9549. – DOI 10.1109/PROC.1974.9549.

– ISSN 0018–9219

[SR70] SCHWEPPE, F.C. ; ROM, D.B.: Power System Static-State Estimation, Part II: Approximate Model. In: *Power Apparatus and Systems, IEEE Transactions on* PAS-89 (1970), Jan, Nr. 1, S. 125–130. http://dx.doi.org/10.1109/TPAS.1970.292679. – DOI 10.1109/TPAS.1970.292679. – ISSN 0018–9510

[SS12] SCHÜTTE, Steffen ; SCHERFKE, Stefan: mosaik - Architecture Whitepaper / OFFIS - Institut für Informatik Oldenburg. 2012. – Forschungsbericht

[Ste98] STEVENS, W. R.: *UNIX Network Programming, Vol. 1 - Networking APIs: Sockets and XTI*. 2. Prentice Hall, 1998

[Str14] *StreamBase Homepage*. Website, Aug 2014. – http://www.streambase.com; Letzter Zugriff: 27.08.2014.

[SW70] SCHWEPPE, F.C. ; WILDES, J.: Power System Static-State Estimation, Part I: Exact Model. In: *Power Apparatus and Systems, IEEE Transactions on* PAS-89 (1970), Jan, Nr. 1, S. 120–125. http://dx.doi.org/10.1109/TPAS.1970.292678. – DOI 10.1109/TPAS.1970.292678. – ISSN 0018–9510

[Syb14] *SAP Sybase CEP Homepage*. Website, Aug 2014. – http://www.sybase.de/products/financialservicessolutions/complex-event-processing; Letzter Zugriff: 27.08.2014.

[Sú14] SÚSTRIK, Martin: *ZeroMQ: The Design of Messaging Middleware*. Website, Feb 2014. – http://www.drdobbs.com/architecture-and-design/zeromq-the-design-of-messaging-middlewar/240165684; Letzter Zugriff: 22.10.2014.

[TH67] TINNEY, William ; HART, Clifford: Power Flow Solution by Newton's Method. In: *Transactions on Power Apparatus and Systems*, IEEE, Nov 1967, S. 1449–1460

[Tra13] TRAN, Thanh Thi L.: *High-Performance Processing of Continuous Uncertain Data*, University of Massachusetts - Amherst, Diss., 2013

[Zera] *ØMQ - Broker vs. Brokerless*. Website, . – http://zeromq.org/whitepapers:brokerless; Letzter Zugriff: 23.10.2014.

[Zerb] *ØMQ - The Guide*. Website, . – http://zguide.zeromq.org; Letzter Zugriff: 22.10.2014.

Index

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 9. Februar 2015

Jan Sören Schwarz