Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Gregor Balthasar

Skalierbare Integration von 4GL-Modellen für massive Smartgrid-Simulationen zur Einbindung von Endverbrauchern

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Gregor Balthasar

Skalierbare Integration von 4GL-Modellen für massive Smartgrid-Simulationen zur Einbindung von Endverbrauchern

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Wolfgang Renz
Zweitgutachterin: Prof. Dr. Bettina Buth

Eingereicht am: 26. November 2014

Gregor Balthasar

Thema der Arbeit
Skalierbare Integration von 4GL-Modellen für massive Smartgrid-Simulationen zur Einbindung von Endverbrauchern

Stichworte
Demand Side Management, massive Simulationen, Multiagentensysteme, Integration von Matlab-Simulink

Kurzzusammenfassung
Demand Side Management ist ein Schlüsselfaktor für das Smartgrid der Zukunft. Um nutzbare DSM-Potenziale zu erforschen, ist es nötig Simulationen durchzuführen. Diese Arbeit stellt ein Konzept zur skalierbaren Integration von Matlab-Simulink-Modellen in eine performantere Simulationsumgebung vor. Außerdem wird ein DSM-Simulations-Framework erstellt, welches dieses Konzept nutzt. Das Ziel ist es, die Performanz und Skalierbarkeit des Integrationskonzepts zu prüfen und Potenziale von strombasierten Heizungssystemen aufzuzeigen.

Gregor Balthasar

Title of the paper
Scalable Integration of 4GL-Models for Massive Smartgrid-Simulations Embedding Consumers

Keywords
Demand side management, massive simulations, multi-agent systems, Matlab-Simulink integration

Abstract
For future smartgrids demand side management is a key factor. Investigating exploitable DSM-potentials constitutes the need for simulations. In this work a concept for a scalable integration of Matlab-Simulink-models into a faster performing environment is conceived. Furthermore, a DSM-simulation-framework is developed utilizing this concept. The goal is to prove the performance and the scalability of the integration on the one hand and to identify the potentials of electric heating systems on the other.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1 Einführung

Regenerative Energiequellen wie Windkraft- und Photovoltaikanlagen werden nicht erst durch den mit der Energiewende[1] einhergehenden Abbau von Atomkraft und die darauf folgenden Novellierungen des Erneuerbare-Energien-Gesetzes (EEG)[2] ein zunehmender Faktor der Energieversorgung. Jedoch werden auch die Problemstellungen für eine qualitativ gleichbleibende Energieversorgung mit der Zunahme von regenerativen Energiequellen immanenter.

Das europäische Wechselspannungsnetz wird nach [IEC 60038 (2009)] mit einer Frequenz von 50 Hertz betrieben. Dabei müssen Energieerzeugung und -verbrauch immer im Gleichgewicht sein, um diese Frequenz stabil zu halten und so das Netz vor einem Zusammenbruch zu bewahren. Hier zeigt sich die Problematik der Integration erneuerbarer Energiequellen deutlich: Windkraftanlagen produzieren nur Strom, wenn ausreichend Wind vorhanden ist, Photovoltaikanlagen nur bei ausreichend Sonneneinstrahlung. Diese Art der Energieerzeugung ist also schlecht plan- und prognostizierbar. Somit muss bei auftretenden Lastspitzen sowohl des Energieverbrauchs als auch der Energieerzeugung kurzfristig mit Regelenergie (siehe [ENTSOE (2004)]) ein Ausgleich geschaffen werden. Diese Regelenergie kann positiv sein - zusätzliche Energie muss in das Netz gespeist werden - oder negativ - überschüssige Energie muss verbraucht werden. Dabei wird zwischen Primär- und Sekundärregelung sowie Minutenreserve unterschieden. Sowohl Primär- als auch Sekundärregelung müssen sehr schnell verfügbar sein. Die Minutenreserve hingegen muss ihre volle Leistung erst nach einer Viertelstunde erreicht haben und diese über einen Zeitraum von bis zu vier Stunden halten können.

Für positive Regelenergie werden häufig z.B. Gasturbinen genutzt, die innerhalb kürzester Zeit an- und abschaltbar sind. Das dafür benötigte Gas kann beispielsweise durch die Power-to-Gas Methode (vgl. [Schäfers u. a. (2013)]), bei der mittels Elektrolyse und Methanisierung aus Strom und Wasser Methan gewonnen wird, erzeugt werden. Die Power-to-Gas Methode kann wie-

[1] §7, Artikel 1, Dreizehntes Gesetz zur Änderung des Atomgesetzes, Bundesgesetzblatt Nr. 43 (2011)
[2] Offizieller Titel: „Gesetz für den Ausbau erneuerbarer Energien“, Inkrafttreten: 01.04.2000

derum für negative Regelenergie eingesetzt werden, um Lastspitzen in der Energieerzeugung auszunutzen.

Ein weiteres nachhaltiges Konzept sind Pumpspeicherkraftwerke, bei denen mittels negativer Regelenergie Wasser auf ein höheres Level gepumpt wird und das bei Bedarf positiver Regelenergie durch eine Turbine zur Stromerzeugung zurückgeleitet wird. Diese Pumpspeicherkraftwerke können allerdings nicht ohne weiteres überall gebaut werden, natürliche Höhenunterschiede wie z.B. in Norwegen und der Schweiz sind dabei von großem Vorteil.

Das dritte Konzept, um den Schwankungen von Stromerzeugung und -verbrauch entgegenzuwirken, ist das Demand Side Management (DSM), dessen Begriff von [Gellings (1985)] geprägt wurde und dessen Definition seitdem ständigen Erweiterungen und Änderungen unterliegt. DSM bedeutet nach heutigem Stand nicht nur die Erhöhung der Verbrauchereffizienz und eine bewusstere Nutzung von Strom, sondern insbesondere eine Anpassung des Stromverbrauchs in Bezug zur Stromerzeugung, um dadurch ebenfalls positive und negative Regelenergie bereitstellen zu können. Hierbei kann es sich um Abnehmer aus Industrie und Gewerbe, aber auch um Privathaushalte handeln. Industrielle Verbraucher wie z.B. Aluminiumhütten können unproblematisch für einen bestimmten Zeitraum abgeschaltet werden, wobei ein Abschalten eines Verbrauchers dem Einspeisen positiver Regelenergie gleichkommt und dieses damit unnötig macht. Aber auch das Einbeziehen von in Schwärmen großer Anzahl zusammengefassten Privathaushalten und den dort installierten Haushaltsgeräten bzw. strombasierten Heizungen wie Wärmepumpen und Nachtspeicherheizungen könnte große Potenziale bereithalten. Häufig wird in Bezug auf DSM auch vom Smartgrid gesprochen, dem intelligenten Energienetz, für dessen Etablierung das DSM eines der Schlüsselelemente ist. Für einen weiteren Einblick in DSM empfiehlt sich z.B. [Schäfers u. a. (2010)] oder [Dethlefs (2014)].

1.1 Motivation

Das Projekt „Beladealgorithmen“ ist entstanden in Kooperation eines großen Energieversorgungsunternehmens (EVU) mit dem *Center For Demand Side Integration* (C4DSI)[3] sowie dem *Labor für multimediale Systeme* (MMLab)[4] der Hochschule für Angewandte Wissenschaften (HAW) Hamburg[5]. Ziel dieses Projektes ist einerseits die Analyse der DSM-Potenziale für strombasierte Heizungen in großen Schwärmen von Privathaushalten. Andererseits sollen für

[3] http://www.haw-hamburg.de/cc4e/c4dsi.html
[4] http://cms-server.ti-mmlab.haw-hamburg.de/mmlab/
[5] http://www.haw-hamburg.de/

derartige Systeme dezentrale Algorithmen zur optimierten Lastverlaufsplanung und Lastverlagerung entwickelt und getestet werden [Braunagel u. a. (2013)].

Folgende Annahmen liegen dieser Zielstellung dabei zugrunde und sollen im Rahmen des Projektes geprüft werden:

- Ein Haus bzw. eine Wohnung ist ein träges thermisches System und bietet somit bestimmte Flexibilitäten bezüglich des Heiz-Zeitpunkts, woraus ein Demand-Side-Management Potenzial folgt.
- Eine zentrale Optimierung der Lastverlaufspläne tausender Privathaushalte ist durch die NP-schwere Charakteristik kombinatorischer Optimierung nicht effizient (vgl. [Merz (2003)].
- Durch den Einsatz eines dimensionslosen Präferenzsignals in Zusammenspiel mit selbstlernenden Algorithmen zur Bedarfserfassung innerhalb eines Haushalts ist eine dezentrale Lastverlaufsplanung möglich.

Da die Prüfung dieser Annahmen im Feld bereits für wenige Haushalte nur mit erheblichem Aufwand möglich ist, insbesondere aber eine große Anzahl von bis zu 100.000 Haushalten für Skalierungs- und Durchmischungseffekte, sowie Gesamtpotenziale betrachtet werden soll, ist der Einsatz von Simulationen unumgänglich.

Dafür baut ein Team des C4DSI, bestehend aus Ingenieuren der Umwelttechnik, der Elektrotechnik und des Maschinenbaus, in einem ersten Schritt in Zusammenarbeit mit dem EVU, das Daten für z.B. die Bausubstanz von Bestandshaushalten liefert, ein Simulationssystem auf. Dieses Simulationssystem wird mit der Modell und Skriptsprachen orientierten Matlab-Simulink-Entwicklungsumgebung umgesetzt, einer sogenannten *Fourth generation language* (4GL)[6]. In diesem System werden modular reale Akteure bzw. Teilsyteme wie Dispatcher, Aggregator, SubAggregatoren und Haushalte mit Steuerboxen abgebildet. Die Haushalte werden durch mikroskopische, thermodynamische Modelle repräsentiert, womit sich das Projekt von anderen Arbeiten abgrenzt, bei denen größere Simulationen auf makroskopischer Ebene durchgeführt werden (z.B. [Ramchurn u. a. (2011)]). Die dezentrale Algorithmik für die verschiedenen Teilsysteme wird ebenfalls mit Matlab-Simulink entwickelt.

Die Matlab-Simulink-Umgebung ermöglicht dabei den modellbasierten Aufbau der benötigten thermodynamischen Haushaltssimulationen und die schnelle skriptbasierte Implementation

[6] Der Begriff wurde geprägt durch [Martin (1982)] und gewinnt durch die zunehmende modellgetriebene Softwareentwicklung wieder an Bedeutung, s. z.B. [Beydeda u. a. (2006)]

von Algorithmen, jedoch stehen dabei Skalierbarkeit und Performanz nicht im Fokus. So sind zwar kleine Simulationen im Bereich von 100-200 Haushalten und einer Simulationszeit von einigen Wochen noch praktikabel, Simulationen über mehrere Monate und mit mehreren zehntausend simulierten Haushalten z.B. aber gar nicht mehr ausführbar.

Allerdings bietet Matlab-Simulink die Möglichkeit, bestehende Modelle als nativen C-Code bzw. als kompilierte Bibliothek zur Verfügung zu stellen. Dadurch können diese Modelle z.B. in einer performanteren Simulationsumgebung genutzt werden.

1.2 Zielsetzung

Da die Projektziele nur durch die Auswertung massiver Simulationen mit mehreren zehntausend Haushalten erreicht werden können, soll im Rahmen dieser Arbeit ein leicht konfigurierbares Simulations-Framework erstellt werden, mit dem das in Matlab-Simulink bestehende Simulationssystem abgebildet werden kann.

Der Hauptfokus des Frameworks liegt dabei darauf, die skalierbare und performante Integration von 4GL (spezifisch: Matlab-Simulink)-Modellen zu ermöglichen, die aus dem bestehenden Simulationssystem exportiert werden sollen. Hierbei ist es wichtig, dass der Ablauf von Import und Export nicht zu aufwändig sein darf, da die Modelle permanent weiterentwickelt werden.

Bei der zu nutzenden Ausführungs- bzw. Simulationsplattform sollen ebenfalls die Aspekte Skalierbarkeit und Performanz im Fokus stehen. Hier wird der Einsatz von *Multiagenten basierten Simulationen* (MABS) vorgeschlagen, da diese gute Verteilungs- und Skalierungseigenschaften innehaben [Schuldt u. a. (2008)].

Ein weiterer Teil der Arbeit ist es, die in Matlab-Simulink bereits bestehenden Algorithmen in das Simulations-Framework zu überführen und deren Funktionalität zu testen.

Die Skalierbarkeit und Performanz des Simulations-Frameworks soll nach Fertigstellung ausgewertet werden. Daraufhin soll es von den Ingenieuren des C4DSI genutzt werden können, um die Simulationsläufe zum Test der dezentralen Algorithmik und zur Analyse des DSM-Potenzials ausführen zu können. Ausgewählte domänenspezifische Ergebnisse werden ebenfalls im Rahmen dieser Arbeit besprochen.

1.3 Aufbau der Arbeit

Die Arbeit ist in acht Kapitel gegliedert. Nach der Einführung in die Thematik und der Zielsetzung der Arbeit in Kapitel 1, wird in Kapitel 2 anhand ausgewählter Werke ein Stand der Technik präsentiert und eine Abgrenzung zu dieser Arbeit vorgenommen.

Die Analyse der Anforderungen wird in Kapitel 3 realisiert, es werden Anwendungsfälle definiert und funktionale sowie nicht-funktionale Anforderungen aufgestellt.

In Kapitel 4 wird eine ausführliche Analyse der technischen Herausforderungen an die Simulationsumgebung durchgeführt und es werden praktikable Lösungsansätze vorgestellt. Die Reihenfolge orientiert sich dabei an der Wichtigkeit der Anforderungen.

Im Anschluss daran wird in Kapitel 5 die Architektur der Simulationsumgebung und das Design der darin enthaltenen Komponenten präsentiert.

Anhand relevanter Beispiele wird in Kapitel 6 die dem Design folgende Implementation des erstellten Simulations-Frameworks besprochen.

Mithilfe von Testfällen werden im ersten Teil des Kapitels 7 die Tests, denen das Simulations-Framework während und nach Abschluss der Entwicklung unterzogen wurde, erläutert. Im zweiten Teil wird zuerst eine Auswertung hinsichtlich Skalierbarkeit und Performanz durchgeführt. Im Anschluss werden ausgewählte Ergebnisse der domänenspezifischen Auswertung vorgestellt.

Zuletzt folgt in Kapitel 8 eine Zusammenfassung der Arbeit und ein Ausblick auf mögliche und bereits entstandene Folgearbeiten.

2 Stand der Technik

Dieses Kapitel gliedert sich in zwei Teilbereiche. Im ersten Bereich wird dabei auf den Stand der Technik für Simulationswerkzeuge mit dem Fokus auf DSM bzw. Smartgrids eingegangen. Dafür werden zwei Simulationswerkzeuge vorgestellt, die sich in stetiger Entwicklung befinden und von denen auch in aktuellen Arbeiten zum Thema DSM vielfältig Gebrauch gemacht wird.

Der zweite Bereich geht auf Arbeiten ein, die einen ähnlichen domänenspezifischen Fokus wie die hier vorgestellte haben. Am Ende dieses Kapitels wird eine Abgrenzung zu dieser Arbeit vorgenommen.

2.1 Simulationswerkzeuge für Smartgrid-Simulationen

Dem Forschungsbereich des DSM bzw. weiter gefasst der Smartgrid-Technologien kommt aktuell sehr viel Aufmerksamkeit zu. Dementsprechend entstehen viele Projekte, die es zum Ziel haben, Forschungsgruppen Werkzeuge an die Hand zu geben, um in diesem Bereich Simulationen durchführen zu können und Strategien zu testen. An dieser Stelle werden zwei populäre Ansätze vorgestellt.

2.1.1 GridLAB-D

GridLAB-D wird entwickelt vom *US Department of Energy* (D.O.E.)[1] am *Pacific Northwest National Laboratory* (PNNL)[2]. Es bietet eine flexible Simulationsumgebung zur Analyse von Stromapplikationen wie z.B. DSM mit *Verteilten Stromerzeugern - Distributed Energy Resource* (DER) - oder Haushaltsgeräten, aber auch zur Marktanalyse.

[1] http://www.energy.gov/
[2] http://www.pnl.gov/

Kern des Systems ist ein fortschrittlicher Algorithmus zur Ermittlung des simultanen Zustands von Millionen unterschiedlicher Geräte, die durch vielfache Differentialgleichungen beschrieben sind [Chassin u. a. (2008)]. Dieser Algorithmus soll gegenüber der Finite-Differenzen-Methode die Vorteile haben, (i) genauer zu sein, (ii) unterschiedliche Zeitskalen zu unterstützen und (iii) sich leichter in neue Module integrieren zu lassen.

Weiterhin bietet GridLAB-D die Möglichkeit der Simulation und Kontrolle des Stromflusses, Module zur Simulation von Haushaltsgeräten und deren Steuerung, sowie zur Sammlung und Auswertung von Ergebnisdaten. Auch differentialgleichungsbasierte Endverbraucher-Modelle, z.B. für Wärmepumpen, Nachtspeicherheizungen und Durchlauferhitzer, sind Bestandteil des Systems.

Zur Integration von z.B. DSM Strategien bietet GridLAB-D agenten- und informationsbasierte Modellierungswerkzeuge an. Zur Beschreibung von Simulationskonfigurationen werden *Grid Lab Model* (GLM) genannte Dateien genutzt. Ferner bestehen Schnittstellen zur Nutzung externer Werkzeuge wie z.B. Matlab oder SynerGEE, jedoch nicht zur Integration von 4GL-Modellen.

2.1.2 mosaik

Das mosaik-Framework[3] wird vom *OFFIS - Institut für Informatik*, einem An-Institut der Universität Oldenburg, entwickelt. Die Idee hinter mosaik ist es, verschiedene ggf. bereits bestehende, technisch heterogene Simulationsmodelle in einer großen Smartgrid-Simulation zusammenzubringen [Rohjans und Lehnhoff (2013)]. Mosaik bietet dabei selbst keine Möglichkeiten, Kontrollstrategien oder -algorithmen direkt zu integrieren, sondern trennt klar zwischen Simulation und deren Konfiguration sowie Kontrollkomponenten.

Um ein solches integratives System zu erreichen, werden verschiedene Artefakte bereitgestellt. Zum einen gibt es das *SimAPI* genannte Interface, das von zu nutzenden Simulationsmodellen implementiert werden muss. Solche Modelle können zum Beispiel auch Matlab-Simulink-Modelle bzw. 4GL-Modelle im Allgemeinen sein. Es muss allerdings möglich sein, das SimAPI-Interface zu implementieren bzw. ein Modell damit zu kapseln. Zum anderen wird eine *Domänenspezifische Sprache* (DSL) mit dem Namen *Mosaik Specification Language* (MoSL) bereitgestellt, mit der sowohl Simulationsszenarien zur Ausführung mit mosaik beschrieben, als auch die darin verwendeten Modelle spezifiziert werden können. Um die Event basierte

[3] https://mosaik.offis.de/

Koordination und Synchronisierung der verschiedenen Bestandteile zu erreichen, wird auf *SimPy*[4], einem Python basierten Simulationsframework, aufgesetzt. Mosaik ist dementsprechend ebenfalls in Python implementiert und somit auch interoperabel einsetzbar, da für die meisten Betriebssysteme Python-Interpreter zur Verfügung stehen.

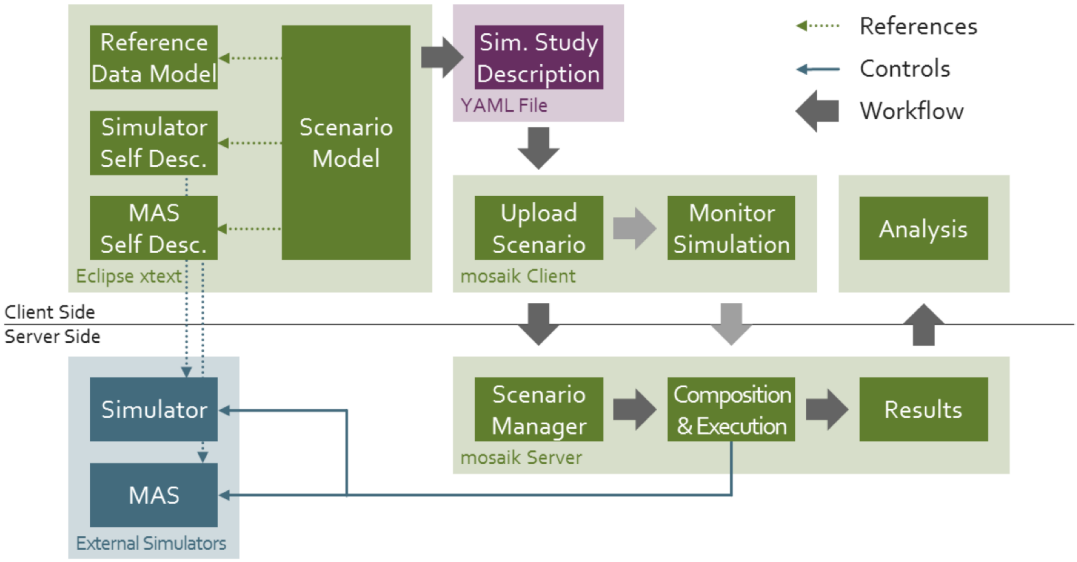


Abbildung 2.1: mosaik Komponenten. Quelle: [Rohjans und Lehnhoff (2013)]

Abbildung 2.1 zeigt die Komponenten des mosaik-Frameworks. Hier wird ersichtlich, dass mosaik nur für die Simulation und deren Beschreibung zuständig ist. Die Farben grün und lila markieren die mosaik-Komponenten. Blau markiert sind die externen Komponenten. Das sind sowohl die Simulationsmodelle, als auch z.B. ein oder mehrere MAS, die die Steuerungsstrategien und -algorithmen für diese Modelle bereitstellen können. Diese MAS müssen wiederum ein *ControlAPI* genanntes Interface implementieren, das dem Steuerungssystem z.B. asynchrone Funktionen zur Ein- bzw. Ausgabe von Daten in bzw. aus den Simulationsmodellen zur Verfügung stellt. In welcher Form die Kommunikation stattfindet, also ob z.B. ein Message basierter oder RPC basierter Ansatz gewählt wurde, wird nicht beschrieben. Hier wäre die Analyse des frei zugänglichen Source-Codes nötig. Ferner existiert für mosaik sowohl eine Online-Dokumentation, als auch eine umfangreiche Publikationsbasis.

Mosaik wirbt mit „large-scale simulations“ mit tausenden Entitäten, allerdings wird dieser Aspekt in den Publikationen zu mosaik nicht belegt. Zu bedenken ist, dass zumindest im

[4] https://simpy.readthedocs.org/

weit verbreiteten CPython[5] durch den *Global Interpreter Lock*[6] (GIL) eine Multi-Threading Anwendung im schlimmsten Fall Nachteile und im besten Fall keine Vorteile hat. Dies kann allerdings z.B. durch den Einsatz von Jython[7], eines Java basierten Python-Interpreters, in dem es keinen GIL gibt, umgangen werden.

2.2 Simulationen zur Analyse von DSM Potenzialen und Strategien

Der Bereich Simulationen für DSM erlangte in den vergangenen Jahren eine große Beliebtheit, es existiert eine Vielzahl an Arbeiten mit weiterhin steigender Tendenz. Im Folgenden wird eine Auswahl solcher Arbeiten anhand der von ihnen betrachteten Strategien zur Lastverminderung kategorisiert.

Viele Arbeiten etablieren marktbasierte Ansätze und nutzen MAS zur Implementation von Biet-Strategien, um dadurch eine Lastglättung und -verminderung zu erzielen, z.B. [Bunn und Oliveira (2001)], [Deindl u. a. (2008)] und [Logenthiran und Srinivasan (2011)]. In [Dethlefs (2014)] wird ebenfalls ein marktbasiertes Modell entworfen, allerdings werden dort Metaheuristiken in Form von z.B. genetischen Algorithmen bzw. der *Ant-Colony-Optimization* (ACO) zur Lastoptimierung verwendet.

Andere Arbeiten analysieren den Nutzen von verteilten Algorithmen unter Nutzung von MAS [Ahat u. a. (2013)], teilweise in Verbindung mit einem Preissignal [Ramchurn u. a. (2011)]. Auch in [Ghaemi und Schneider (2013)] wird ein Preissignal verwendet, allerdings unter Nutzung von GridLAB-D als Plattform.

In [Wang u. a. (2012)] wird im Gegensatz zu den anderen Arbeiten ein zentrales Kontrollsystem für Microgrids[8] im Zusammenspiel von GridLAB-D und Matlab beschrieben.

Weiterhin lässt sich diese Auswahl von Arbeiten einer Einteilung hinsichtlich der von ihnen genutzten Simulationsmodelle unterziehen.

So werden in den Arbeiten von [Ahat u. a. (2013)], [Bunn und Oliveira (2001)] und [Deindl u. a. (2008)] die für die Simulation verwendeten Modelle von Stromverbrauchern bzw. -erzeugern entweder gar nicht oder als äußerst vereinfacht beschrieben.

[5] https://www.python.org/
[6] https://wiki.python.org/moin/GlobalInterpreterLock
[7] http://www.jython.org/
[8] Microgrids sind kleinere, lokale Systeme von Stromerzeugern, -speichern und -verbrauchern

Auf statistischen Daten basierende Modelle werden in [Logenthiran und Srinivasan (2011)] und [Ramchurn u. a. (2011)] für die simulationsbasierte Analyse verwendet.

In [Dethlefs (2014)] wird ein einfaches thermisches Modell für die simulierten Haushalte verwendet, während in [Ghaemi und Schneider (2013)] ein von GridLAB-D bereitgestelltes, differentialgleichungsbasiertes Wärmepumpen-Modell genutzt wird. Das detaillierteste Modell wird in [Wang u. a. (2012)] eingesetzt. Dort werden einige hundert, in Matlab simulierte thermodynamische Modelle für Häuser mit Wärmepumpen, sowie das Stromfluss-Modell von GridLAB-D genutzt.

Im folgenden Abschnitt werden die zwei Ansätze, die ein Preissignal für das DSM verwenden, etwas ausführlicher beschrieben, da auch der in dieser Arbeit verwendete Ansatz ein (wenn auch normalisiertes, dimensionsloses) Preissignal verwendet.

2.2.1 Simulationen mit Preissignal basiertem DSM

[Ramchurn u. a. (2011)] beschreiben die Definition eines „Smart Home“-Modells, bei dem eine Einteilung in verlagerbare statische Lasten und thermische Lasten vorgenommen wird. Anhand dieses „Smart Home“-Modells wird die Entwicklung eines verteilten Algorithmus zur Lastminderung und Glättung von Lastspitzen auf Basis eines Preissignals und Agententechnologien vorgenommen. Dieser verteilte Algorithmus soll verhindern, dass sich neue Lastspitzen dadurch ergeben, dass alle Empfänger eines Preissignals gleichzeitig zu besonders günstigen Zeiten ihren Verbrauch planen. Dies wird erreicht indem eine durch Wahrscheinlichkeiten erzeugte Trägheit in die Lastoptimierung der „Smart Home“-Kontroll-Agenten, so dass nicht alle Agenten gleichzeitig ihre Optimierung durchführen. Zum simulationsbasierten Test des verteilten Algorithmus wird ein MAS eingesetzt. Dabei werden Simulationen mit 5.000 Agenten durchgeführt, deren Lastprofile auf Durchschnittswerten von Verbrauchsprofilen von 26 Millionen Haushalten in Großbritannien basieren. Auch für die verlagerbaren und thermischen Lasten werden statistische Daten genutzt. Dabei wurden Lastspitzenreduzierungen von bis zu 17% beobachtet.

In [Ghaemi und Schneider (2013)] wird eine Analyse des Demand Side Management Potenzials für Privathaushalte mit Wärmepumpen in Österreich durchgeführt. Als Annahme liegt dabei zugrunde, dass Häuser mit Wärmpepumpen Wärme in Abhängigkeit zu ihrer Bausubstanz speichern können und somit Flexibilitäten für Lastverlagerungen bestehen.

Um diese Analyse durchführen zu können, wurde mittels GridLAB-D eine Simulation aufgebaut. Für die Haushalte mit Wärmepumpen wurde dabei auf ein von GridLAB-D bereitgestelltes, differentialgleichungsbasiertes Modell zurückgegriffen, welches die Innentemperatur anhand von Klimadaten, Gebäudeparametern und der Funktion der Wärmepumpe berechnet. Es wurde eine Testgröße von 1.000 Haushalten festgelegt, für die alle die gleichen Klimadaten, eine durchschnittliche Hausgröße und derselbe Typ Wärmepumpe, jedoch unterschiedliche Bausubstanzen gelten.

Zur Vereinfachung der Erzeugung von Simulationsszenarien wurde eine Werkzeug entwickelt, mit dem die GLM-Dateien mithilfe einer graphischen Benutzeroberfläche erzeugt werden können.

Der Verbrauch der Haushalte wird über Preissignale gesteuert. Dabei wird die Länge der Preissignal-Spitzen variiert, um daraus ableiten zu können, wie lang eine mögliche Lastverlagerung tatsächlich abgerufen werden kann und welche Auswirkungen ein solcher Abruf auf die verschiedenen Bausubstanz-Typen hat. Diese Länge wird durch unterschiedlich konfigurierte *Controller* bestimmt. Ebenfalls betrachtet wird ein „Zurückfederungs-Effekt“, der bei zu langer Abschaltung der Heizungen auftreten kann. Die Gesamtlast des Systems kann dann zwar für einen bestimmten Zeitraum verringert werden, es entstehen nach diesem Zeitraum allerdings neue Lastspitzen, die ihren Ursprung in einer zu großen Auskühlung der Häuser haben.

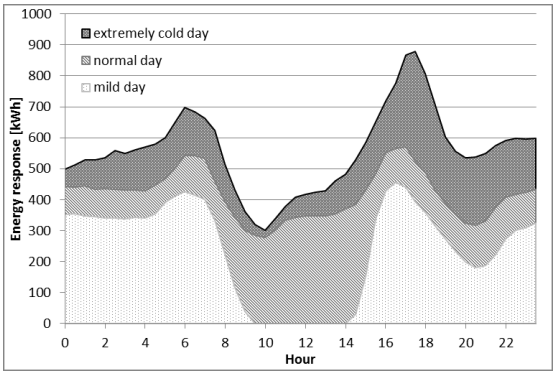


Abbildung 2.2: Lastverlagerungspotenzial für milde, normale und extrem kalte Tage. Quelle: [Ghaemi und Schneider (2013)]

Abbildung 2.2 zeigt die ermittelten Lastverlagerungspotenziale für unterschiedlich kalte Wintertage bei einer Testgröße von 1.000 Haushalten. Dabei wird eine starke Tageszeitabhängigkeit deutlich. Ein weiterer Faktor für die Größe des Potenzials ist die erlaubte Innentemperaturabweichung. In der Arbeit werden nur Abschaltpotenziale betrachtet.

2.3 Abgrenzung zu dieser Arbeit

Die beiden vorgestellten Simulationsumgebungen gehören zum aktuellen Stand der Technik für Smartgrid und DSM-Simulations-Frameworks. GridLAB-D bietet allerdings keine Möglichkeit zur nahtlosen Integration von z.B. aus Matlab-Simulink exportierten 4GL-Modellen. Dadurch ist GridLAB-D in dieser Hinsicht keine Alternative zu dem in dieser Arbeit vorgestellten Integrationsansatz. Mosaik bietet diese Möglichkeit und nominell auch eine hohe Skalierbarkeit, die jedoch nicht belegt ist. Da Mosaik erst nach Abschluss dieser Arbeit verfügbar war, konnte es nicht in der technischen Analyse und beim Entwurf des Systems berücksichtigt werden.

Es wurde eine Auswahl von Arbeiten mit ähnlichem Fokus vorgestellt. Anhand der vorgenommen Kategorisierung konnte gezeigt, dass einige dieser Arbeiten ähnliche, domänenspezifische Lösungsansätze verfolgen. Allerdings werden überwiegend entweder statistische Daten oder einfache differentialgleichungsbasierte Ansätze zur Simulation von Verbrauchern oder Erzeugern genutzt. Diese haben weder den Detailierungsgrad, noch bilden sie stochastische Faktoren ab, wie es die in dieser Arbeit genutzten Matlab-Simulink-Modelle vermögen. In [Schneider u. a. (2011)] wird gezeigt, dass detaillierte physikalische Modelle besonders in Bezug auf Demand Side Management Anwendungen einfachen Modellen gegenüber einen Mehrwert besitzen. Der vorgestellte Ansatz von [Ghaemi und Schneider (2013)] kommt dennoch zu ähnlichen Ergebnissen hinsichtlich der Abschaltpotenziale von Wärmepumpen, wie jenen, die in dieser Arbeit präsentiert werden, beschränkt sich dabei aber auch auf diese.

In einer der Arbeiten werden ähnlich detaillierte, Matlab basierte Verbrauchermodelle verwendet. Die Simulation dieser Modelle findet jedoch innerhalb von Matlab statt und wird über ein Interface genutzt. Es findet also keine skalierbare Integration dieser Modelle, wie in dieser Arbeit vorgestellt, statt.

3 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an das zu entwickelnde Softwaresystem aufgestellt. Diese Anforderungen stützen sich auf Vorgaben aus dem Projekt „Beladealgorithmen“ und ergeben sich aus der Zielsetzung dieser Arbeit. Dazu werden in einem ersten Schritt Anwendungsfälle spezifiziert. Diese helfen bei der Formulierung der Anforderungen, die in funktionale - die Funktionsweise des Systems beschreibende - und nicht-funktionale - die Eigenschaften des Systems beschreibende - Anforderungen aufgeteilt sind.

3.1 Anwendungsfälle

Um die Anwendungsfälle für diese Arbeit zu definieren, wird ein UML-Anwendungsfall-Diagramm genutzt (Abbildung 3.1).

Fünf Anwendungsfälle teilen sich auf drei Subsystemen auf. Es werden zwei verschiedene Nutzertypen eingeführt, der Typ Ingenieur und der Typ Entwickler, für die jeweils vier Anwendungsfälle relevant sind. Der Haupt-Anwendungsfall „Durchführung einer Simulation“ befindet sich innerhalb des Subsystems *Simulationssystem* und ist für beide Nutzertypen von Bedeutung. Dieser schließt den Anwendungsfall „Simulationskonfiguration auswählen“ mit ein, welcher ebenfalls Teil desselben Subsystems ist und von beiden Nutzertypen gebraucht wird. Wenn eine entsprechende Konfiguration nicht existiert, wird dieser Anwendungsfall durch „Simulationskonfiguration erstellen“ erweitert. Letzter wird mithilfe eines *XML-Editors* durchgeführt, dem zweiten Subsystem. Auch dieser Anwendungsfall ist für alle Nutzertypen relevant. Schließlich wird auch der Haupt-Anwendungsfall nach Abschluss und Abspeicherung der Ergebnisse durch „Simulationslauf domänenspezifisch auswerten“ sowie „Simulationslauf nach Skalierbarkeit und Performanz auswerten“ erweitert. Der erstgenannte Anwendungsfall wird dabei vom Nutzertyp Ingenieur wahrgenommen und innerhalb des Subsystems *Matlab* durchgeführt. Der letztgenannte Anwendungsfall ist dem Nutzertyp Entwickler zugeordnet und findet in keinem spezifischen Subsystem statt.

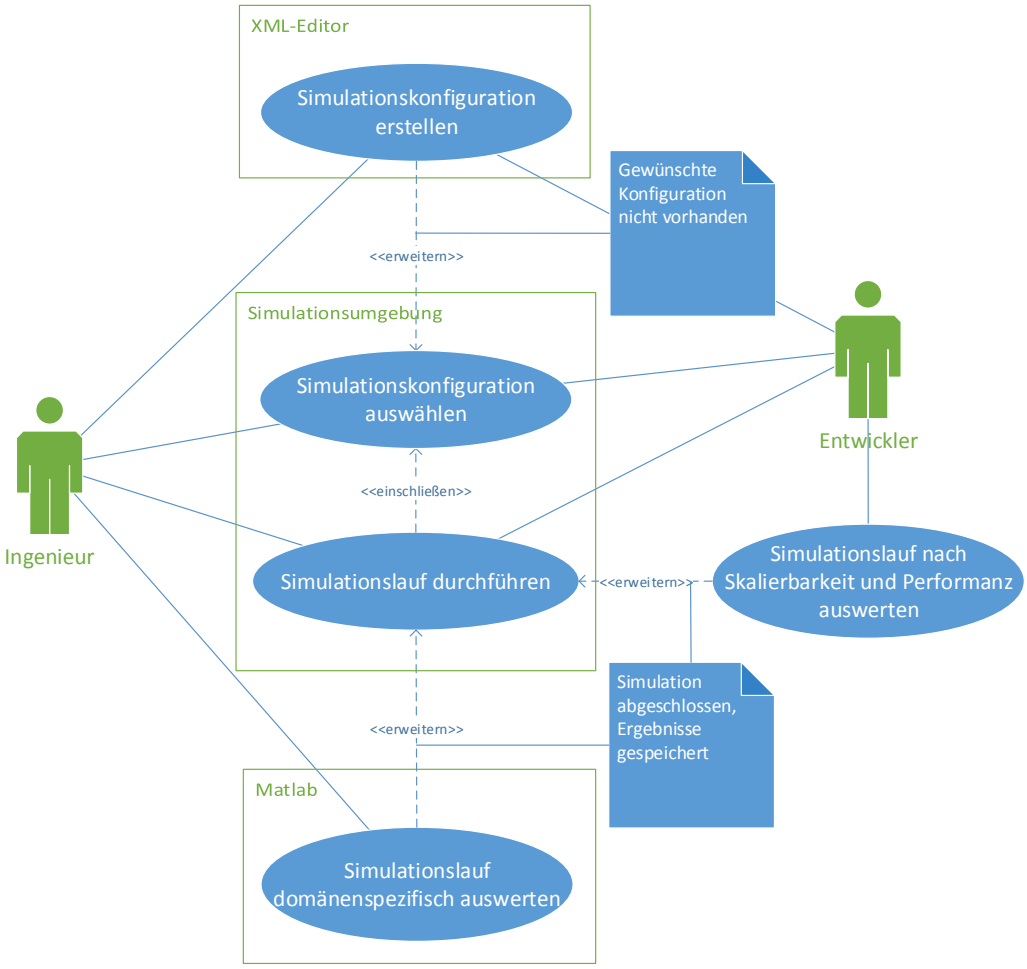


Abbildung 3.1: UML-Anwendungsfall-Diagramm

3.2 Funktionale Anforderungen

Um die innerhalb des Projektes definierten „Massensimulationen“ durchführen und Auswertungen hinsichtlich des DSM-Potenzials vornehmen zu können, ist es notwendig, die in Matlab-Simulink schon vorhandene Simulation strukturell nachzubilden.

Zu diesem Zweck muss eine konfigurierbare Simulationsumgebung erstellt werden, die die energielogistische Akteure wie *Dispatcher*, *Aggregator*, *Sub-Aggragtoren* und *Haushalte* der Matlab-Simulink Simulation beinhaltet und strukturell korrekt abbildet. Diese Umgebung soll von verschiedenen Nutzertypen bedient werden können. Dabei sollen auch die in Matlab-Simulink bestehenden Algorithmen möglichst eins zu eins in diese Struktur überführt werden können. Hierbei wäre auch ein Export dieser Algorithmen denkbar, ist aber als optional angesetzt. Ferner muss es diese Simulationsumgebung analog zur bereits bestehenden Simulation und den thermodynamischen Modellen ebenfalls ermöglichen, diskrete Simulationen auf Minutenbasis durchzuführen. Diese Anforderungen haben eine hohe Priorität, da sie für die Durchführung von Massensimulationen zwingend benötigt werden.

Eine weitere funktionale Anforderung ist die Einbindung von in Matlab Simulink erstellten Haushalts-Modellen in die Simulationsumgebung selbst. Dabei wird es zwei grundsätzlich verschiedene Haushaltsmodelle geben, nämlich die per Wärmepumpe und die per Nachtspeicherheizung beheizten Haushalte, die dann wiederum vielfältig parametrierbar sind, um möglichst viele verschiedene Hausarten abzubilden. Die Integration dieser Modelle steht im Mittelpunkt der Arbeit und ist somit essentiell.

Da mit dem Simulations-Framework nach Fertigstellung analysiert werden soll, welche DSM-Potenziale die verteilte Lastverlaufsplanung sowie die Lastverlagerung für positive und negative Regelenergie in Bezug auf große Schwärme von Haushalten mit Nachtspeicherheizungen bzw. Wärmepumpen innehat, wird auch die Möglichkeit zur Auswertung der simulierten Ergebnisse als wichtige Anforderung mit aufgenommen.

3.2.1 Übersicht: Funktionale Anforderungen

Tabelle 3.1 fasst die hier aufgestellten funktionalen Anforderungen zur besseren Übersicht zusammen und beinhaltet auch deren Klassifizierung. Hieraus geht hervor, dass die Einbindung der Matlab Simulink-Modelle Dreh- und Angelpunkt für die Funktionalität der geplanten Simulationsumgebung darstellt. Die direkte Einbindung von exportierten Matlab-Algorithmen

ID	Funktionale Anforderung	Bedeutung
1	Erstellung einer konfigurierbaren DSM-Simulationsumgebung	wichtig
2	Nachbildung der Kommunikationsstruktur aus Simulink	wichtig
3	Diskrete Simulation auf Minutenbasis	wichtig
4	Einbindung von Matlab Simulink-Modellen	essentiell
5	Einbindung von übersetzten Matlab-Algorithmen	wichtig
6	Direkte Einbindung exportierter Matlab-Algorithmen	optional
7	Potenzialanalyse der verteilten Lastverlaufsplanung und Lastverlagerung für große Schwärme von Nachtspeicherheizungen und Wärmepumpen	wichtig

Tabelle 3.1: Funktionale Anforderungen

wird hingegen als optional betrachtet, alle anderen Anforderungen hingegen auch als wichtig für einen erfolgreichen Abschluss des Projektes.

3.3 Nicht-funktionale Anforderungen

Zusätzlich zu den funktionalen Anforderungen sind durch das Projekt noch verschiedene nicht-funktionale Anforderungen vorgegeben, die in dieser Arbeit berücksichtigt werden müssen. Diese werden in mehrere Untergruppen unterteilt und in den folgenden Unterkapiteln besprochen.

Benutzbarkeit

Die zu entwickelnde Lösung richtet sich nicht an Endnutzer, sondern an Ingenieure. Somit können bezüglich der Benutzbarkeit andere Maßstäbe angelegt werden. Allerdings muss dennoch beachtet werden, dass die designierten Nutzer der Simulationsumgebung keine Informatiker sein werden. Also muss das Simulationssystem auf wohldefinierte Art zu bedienen und zu konfigurieren sein. Die Konfigurationen für verschiedene Simulationsszenarios sollen hierbei durch möglichst in XML beschriebene Konfigurationsdateien definiert werden können. Eine graphische Oberfläche wird nicht benötigt, die Auswertung der Ergebnisse soll laut Vorgabe wieder in Matlab erfolgen.

Umgebungsbedingungen

Um die Ergebnisse einzelner Simulationsläufe möglichst einfach in Matlab auswerten zu können, ist eine weitere Vorgabe, dass diese in .MAT-Dateien geschrieben werden. Ebenso sollen gewisse Eingangsdaten auch aus .MAT-Dateien zum Start einer Simulation ausgelesen werden können.

Wart- und Erweiterbarkeit

Die in der DSM-Simulationsumgebung vorhandenen Akteure *Dispatcher*, *Aggregator*, *Sub-Aggregator* und *Haushalt* sollen zu einem späteren Zeitpunkt noch in ihrer Funktionalität erweitert werden können. Hierfür bietet es sich an, wenn der Einsatz von Standardtechnologien im Sinne von Programmiersprachen, Entwicklungstools, Libraries etc. unterstützt wird, um auch weitere Aspekte wie z.B. die Wartbarkeit und Wiederverwendbarkeit von Code-Blöcken zu fördern.

Die aus Matlab in die Simulationsumgebung zu integrierenden Modelle werden fortlaufend modifiziert. Um einen effektiven Arbeitsfluss zu gewährleisten, muss diese Integration unter möglichst geringem Aufwand durchführbar sein.

Effizienz

Es sollen bis zu 100.000 thermodynamische Haushaltsmodelle parallel simuliert werden können. Mindestanforderung sind aber 20.000 Modelle. Dementsprechend muss die Simulationsumgebung hoch skalierbar sein. Weiterhin sollen die Simulationen nur einen Bruchteil der simulierten Zeit in Echtzeit in Anspruch nehmen, also in beschleunigter Echtzeit ablaufen können.

3.3.1 Übersicht: Nicht-funktionale Anforderungen

In Tabelle 3.2 werden die nicht-funktionalen Anforderung nach Kategorie gelistet und klassifiziert. Zentraler Aspekt ist hierbei die Konfigurationsmöglichkeit der Simulationsumgebung durch XML-Dateien. Alle andere Anforderungen sind jedoch auch für eine erfolgreiche Entwicklung unerlässlich.

ID	Nicht-funktionale Anforderung	Bedeutung
	Benutzbarkeit	
8	Bedienbarkeit durch Ingenieure (keine Informatiker, keine Endnutzer)	wichtig
9	Möglichkeit zur Konfiguration von Simulationsszenarien durch XML-Dateien	essentiell
	Umgebungsbedingungen	
10	.MAT-Dateien als Eingangs- und Ausgangsmöglichkeit für Daten/Ergebnisse	wichtig
	Wart- und Erweiterbarkeit	
11	Möglichkeit zur Weiterentwicklung der Akteure der Simulation durch Einsatz von Standardtechnologien	wichtig
12	Wenige Arbeitsschritte zur Einbindung veränderter Modelle	wichtig
	Effizienz	
13	Parallele Simulation von mindestens 20.000 bis zu 100.000 thermodynamischen Haushaltsmodellen	sehr wichtig
14	Simulation in beschleunigter Echtzeit möglich	wichtig

Tabelle 3.2: Nicht-funktionale Anforderungen

4 Technische Analyse

In diesem Kapitel wird die technische Machbarkeit kritischer Anforderungen betrachtet. Es werden mögliche Lösungswege unter Hinzunahme bestehender Technologien betrachtet und gegebenenfalls miteinander verglichen.

Zuerst wird dabei analysiert, mit welchen Standardtechnologien respektive Programmiersprachen eine Integration von exportierten Matlab Simulink-Modellen überhaupt möglich ist. Dies ist essentiell, um im nächsten Schritt eine Ausführungs- bzw. Simulationsplattform auswählen zu können, mit der die DSM-Simulationsumgebung realisiert werden soll. Nach dieser ersten Auswahl wird für diese Plattform evaluiert, ob und wie eine skalierbare Integration der Modelle zu bewerkstelligen ist.

Daraufhin soll analysiert werden, wie die XML-basierten Konfigurationsdateien hinsichtlich der bis dahin getroffenen Auswahl der Programmiersprache und Plattform umgesetzt werden können, welche ein weiterer kritischer Bestandteil der Software sind. Zuletzt wird das Lesen und Schreiben von .MAT-Dateien betrachtet und ein Lösungsweg dafür aufgezeigt.

4.1 Integration der Matlab-Simulink-Modelle

Matlab-Simulink bietet die Möglichkeit, bestehende Modelle auf verschiedene Arten zu exportieren. Der sogenannte Real-Time Workshop (in neueren Versionen Coder genannt), der ein Bestandteil von Matlab Simulink ist, ermöglicht die Erzeugung und Kompilierung von C/C++-Code aus Simulink Modellen. Auch die direkte Kompilierung von Shared Libraries wird unterstützt. Je nach Betriebssystem werden DLL-Dateien (Windows), SO-Dateien (Linux) oder DYLIB-Dateien (Mac OS) erzeugt. Diese Shared Libraries sind eigenständig und besitzen keine weiteren Abhängigkeiten.

Um die Tests im Rahmen der technischen Analyse zu ermöglichen, wird ein einfaches Simulink-Modell mit zwei Eingängen und einem Ausgang erstellt, das eine simple Addition abbildet. Die

Ein- und Ausgänge sind Zeiger-Variablen vom Typ Double. Zur Erzeugung wird Matlab Simulink 2011b in der 64bit Version von Windows 7 genutzt, welches auch von den Projektpartnern genutzt wird. Damit wird eine 64bit-DLL des Modells erstellt.

Im Folgenden werden Wege zur Integration dieser DLL für die drei weit verbreiteten Programmiersprachen Java, C# und Python geprüft.

4.1.1 Java

Für Java wurden zwei Möglichkeiten zum Aufruf nativen C/C++-Codes gefunden. Zum einen, das zu Java standardmäßig zughörige Java Native Interface (JNI)[1], zum anderen eine freie Bibliothek namens Java Native Access (JNA)[2], die auf JNI aufsetzt.

Java Native Interface (JNI)

JNI setzt voraus, dass am Quellcode der Shared Library Änderungen vorgenommen werden und somit eine Schnittstelle für den Zugriff durch Java implementiert wird. Da im Laufe des Projektes immer wieder Änderungen an den Modellen vorgenommen und diese dann neu exportiert werden müssen, wäre dieser Weg mit erheblichem Aufwand verbunden. Dies ist im Rahmen des Projektes nicht möglich, von daher wird diese Methode nicht näher betrachtet.

Java Native Access (JNA)

Die JNA-Library bietet einen Wrapper für JNI und übernimmt das Mapping von Datentypen und -strukturen. Dadurch entfällt der Eingriff in den Quellcode der Shared Library und es muss nur ein Java-seitiges Interface implementiert werden.

Quelltext 4.1 zeigt die Java-Interface-Klasse für die beispielhafte Modell-DLL. Dort wird der Pfad zur DLL angegeben und diese registriert und initialisiert. Ferner werden dann die externen Funktionen publiziert.

In der in Quelltext 4.2 gezeigten Klasse werden dann zuerst Ein- und Ausgabe-Parameter als Double-Arrays gesetzt und diese dann zum Aufruf der DLL genutzt. Diese Arrays sind per

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/
[2] https://github.com/twall/jna

Definition Pointer-Objekte und können somit auch Daten zurückgeben, die Funktion selbst hat keinen expliziten Rückgabewert.

Diese Art der Einbindung ist problemlos und auch für einen Arbeitsfluss, bei dem die Modelle häufig geändert werden, geeignet.

```
package de.hawhamburg.beladealgorithmen.nativeholder;

import com.sun.jna.Native;

public class DllInterface {

    static
    {
        // Holen des User-Verzeichnisses
        String userDir = System.getProperty("user.dir");
        // Setzen des DLL-Pfads
        System.setProperty("jna.library.path", userDir + "/dllModels");
        // Registrieren der DLL
        Native.register("TestModel");
        // Initialisieren der DLL
        testModell_initialize('1');
    }

    // Externe Funktionen veroeffentlichen
    public synchronized static native void testModell_initialize(Character
        firstTime);
    public synchronized static native void testModell_custom ( double[]
        inputA, double[] inputB, double[] output );
}
```

Quelltext 4.1: DllInterface.java

4.1.2 C#

Die Programmiersprache C# bietet eine integrierte Möglichkeit zur Einbindung von nativem C/C++-Code, ohne dass dabei Eingriffe im Quelltext stattfinden müssten. Auch das Datentyp-Mapping geschieht automatisiert.

In Quelltext 4.3 wird beispielhaft gezeigt, wie durch das *[DllImport]*-Attribut die beiden externen Funktionen zur Verfügung gestellt und in der Main-Methode genutzt werden. Dazu werden Input- und Output-Parameter genau wie in Java als Double-Arrays gesetzt und mit der Funktion aufgerufen.

Auch hier steht der im Projekt benötigten Nutzungsweise kein Hindernis im Weg.

```java
package de.hawhamburg.beladealgorithmen.tests;

import de.hawhamburg.beladealgorithmen.nativeholder.DllInterface;

public class TestModell {

    public static void main(String[] args) {
        // Input Params
        double[] inputA = { 1.0 };
        double[] inputB = { 2.0 };

        // Output Param
        double[] output = { 0.0 };

        // Aufrufen der DLL
        DllInterface.testModell_custom(inputA, inputB, output);

        System.out.println(â Ergebnis: " + output[0]);
    }
}
```

Quelltext 4.2: TestModell.java

4.1.3 Python

Wie auch schon C# stellt die Programmiersprache Python ab Version 2.5 eine integrierte Einbindungsmöglichkeit für nativen C/C++-Code in Form des Packages *ctypes* zur Verfügung. Anders als bei JNA für Java oder in C# wird hier kein automatisiertes Mapping der C-Datentypen vorgenommen, allerdings Möglichkeiten zum Erstellen, Zugreifen und Verändern von diesen bereitgestellt[3].

Quelltext 4.4 zeigt den beispielhaften Aufruf der TestModell.dll mit dem Package *ctypes* in Python. Anders als mit JNA unter Java oder in C# müssen hier die externen Funktionen nicht erst händisch publiziert werden, sondern können nach dem Laden der DLL direkt aufgerufen werden. Zu erkennen ist die Nutzung der C-Datentypen und das durch die Python-anhängige dynamische Typisierung leicht sperrige Mapping auf die *c_double*-Arrays.

Insgesamt gilt aber auch hier, dass der Einsatz von Python innerhalb des Projektes unproblematisch wäre.

[3] http://python.net/crew/theller/ctypes/

```
using System;
using System.Runtime.InteropServices;

namespace de.hawhamburg.beladealgorithmen
{
    class TestModell
    {
        // Deklarieren der externen Funktionen
        [DllImport(@"..\TestModell.dll")]
        static extern void testModell_initialize(char init);
        [DllImport(@"..\TestModell.dll")]
        static extern void testModell_custom(double[] inputA, double[]
            inputB, double[] output);

        static void Main(string[] args)
        {
            // Initialisieren der DLL
            testModell_initialize('1');

            // Inputparams setzen
            double[] inputA = { 2 };
            double[] inputB = { 2 };

            // Outputparam setzen
            double[] output = { 0 };

            // DLL aufrufen
            testModell_custom(inputA, inputB, output);

            // Ausgabe in Konsole
            Console.WriteLine("Ergebnis: "output[0]);
        }
    }
}
```

Quelltext 4.3: TestModell.cs

```
import logging
import ctypes
import os

logging.basicConfig(level=logging.DEBUG,format='(%(threadName)-10s) %(
    message)s',)

class TestModell():
    __dllPath = "C:\\"
    __DLLNAME = "TestModel.dll"

    if __name__ == "__main__":
        # DLL Pfad setzen
        self.__dllPath = os.path.abspath(os.path.join(os.path.dirname(
            __file__), os.pardir, "util", self.__DLLNAME))

        logging.debug("DLL Pfad: " + self.__dllPath)

        # DLL einbinden
        testModell = ctypes.CDLL(self.__dllPath)

        # DLL initialisieren
        logging.debug("Initialize DLL")
        testModell.testModell_initialize('1')
        logging.debug("DLL initialized")

        # Input Params
        inputA = (self.__doubleArray(1))(2)
        inputB = (self.__doubleArray(1))(2)

        # Output Params
        output = (self.__doubleArray(1))(0.0)

        # DLL aufrufen
        testModell.testModell_custom(inputA, inputB, output)

        # Werte ausgeben
        _outputStr = ""
        for i in output: _outputStr += "Ergebnis: " + str(i) + "\n"
        logging.info(_outputStr)

    def __doubleArray(self, elements=1):
        return ctypes.c_double * elements
```

Quelltext 4.4: TestModell.py

4.1.4 Fazit

Alle drei betrachteten Programmiersprachen ermöglichen die Einbindung nativen C/C++-Codes in der Form, wie es die nicht-funktionalen Anforderungen des Projektes vorgeben. Am einfachsten ist die Einbindung in C#, danach folgt Java, weil zuerst eine externe Bibliothek benötigt wird, und erst dann Python, da kein automatisiertes Mapping der C-Datentypen erfolgt.

Tabelle 4.1 zeigt zusammengefasst noch einmal die Eigenheiten der drei Programmiersprachen bezüglich der nativen C/C++-Codeeinbindung.

Programmier-sprache	**Integrierte Unterstützung nativen C/C++-Codes**	**Keine Änderungen am nativen Code nötig**	**Automatisches Datentyp-Mapping**
Java	☒	☐	☐
Java mit JNA		☒	☒
C#	☒	☒	☒
Python	☒[4]	☒	☐

Tabelle 4.1: Übersicht Programmiersprachen bzgl. Einbindung nativen C/C++-Codes

4.2 Ausführungs- bzw. Simulationsplattform

Zur Erfüllung der funktionalen Anforderungen mit den IDs 1-3 (s. Tabelle 3.1 auf Seite 16) sowie der nicht-funktionalen Anforderungen mit den IDs 10, 12 und 13 (s. Tabelle 3.2 auf Seite 18) muss eine geeignete Ausführungs- bzw. Simulationsplattform gefunden werden. In Kapitel 2.1 wurden mit GridLAB-D und mosaik zwei Simulationswerkzeuge vorgestellt, deren Fokus die Simulation von Smartgrid-Systemen ist. GridLAB-D bietet zwar ein Interface zu Matlab aber keine Möglichkeit zur skalierbaren Integration von Matlab-Simulink-Modellen und deren massiver Nutzung. Da diese aber ein essentieller Bestandteil der Arbeit ist, wird von der Nutzung von GridLAB-D als Simulationswerkzeug abgesehen. Mosaik auf der anderen Seite bietet diese Möglichkeiten zwar nominell, war aber zum Zeitpunkt der technischen Analyse und der Implementation des Systems noch nicht verfügbar. Da jedoch die Entwicklung einer gänzlich neuen Simulationsplattform den Rahmen dieser Arbeit sprengen würde, sollen im

[3] ab Python Version 2.5

Folgenden weitere, nicht domänenspezifische Simulationsplattformen auf deren Nutzbarkeit analysiert werden.

Multiagenten basierte Simulationen besitzen verschiedene Eigenschaften wie z.B. Verteil- und Skalierbarkeit, Laufzeitbeschleunigung und das natürliches Mapping von realen Entitäten [Schuldt u. a. (2008)] und können somit vielen der spezifizierten Anforderungen gerecht werden. Einen Einstiegspunkt in das Thema bietet [Davidsson (2001)]. Ferner entstehen immer mehr Projekte, die die Vorteile von Multiagenten Systemen (z.B. Dezentrale Koordination, Kooperation, Emergenz, Möglichkeit zur Selbst-Organisation) auf die Probleme des DSMs anwenden [Deindl u. a. (2008); Logenthiran und Srinivasan (2011); Ramchurn u. a. (2011)]. Einen äußerst umfassenden Überblick gibt [McArthur u. a. (a,b)]. Daraus folgend werden in diesem Abschnitt zwei verschiedene Simulationsplattformen betrachtet, die auf einem MABS-Ansatz beruhen bzw. mit denen sich ein solcher umsetzen lässt. Für einen tieferen Einblick in die Thematik der intelligenten Agenten bzw. der MAS empfehlen sich [Wooldridge und Jennings (1995); Wooldridge (2002)] sowie [Weiss (2000)] und [Ferber (2001)]. Eine kleine Einführung in BDI-Agenten findet sich in Kapitel 5.1.1 auf Seite 46.

4.2.1 Jadex Active Components

Die *Jadex Active Components*-Plattform (Jadex AC)[5] ist eine Eigenentwicklung der Arbeitsgruppe *Verteilte Systeme und Informationssysteme* (VSIS) an der Universität Hamburg. Hervorgegangen ist diese aus der ursprünglichen Jadex v1, die auf dem *Java Agent DEvelopment*-Framework (JADE)[6] basierte und dieses um Konzepte wie z.B. BDI-Agenten und wiederverwendbare Capabilities erweiterte [Braubach u. a. (2005)].

Jadex AC löst sich im Gegensatz zu Jadex v1 von der reinen Multiagenten-Metapher und formuliert den Ansatz der aktiven Komponenten [Pokahr und Braubach (2011)]. Hierbei handelt es sich um eine Erweiterung der *Service Component Architecture* (SCA)[7], die diese um Konzepte der Verteilten Systeme und der Nebenläufigkeit erweitert. Komponenten können nicht nur Dienste anbieten, sondern auch autonom und pro-aktiv agieren. Anwendungen lassen sich auf verschiedenen Rechnern verteilt ausführen, Jadex AC bietet hierzu verschiedene Awareness-Methoden, über die sich Jadex-Plattformen finden können.

[5] http://www.activecomponents.org/
[6] http://jade.tilab.com/
[7] http://www.oasis-opencsa.org/sca/

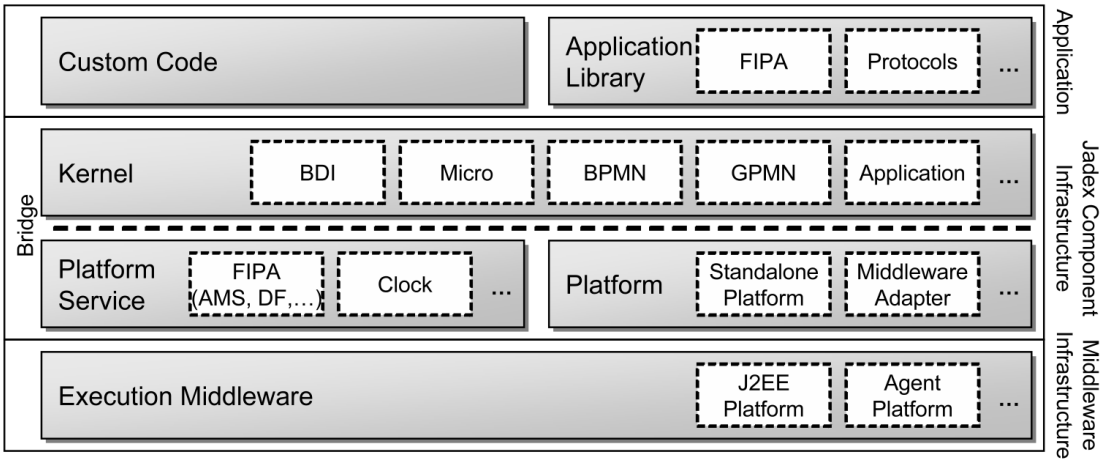


Abbildung 4.1: Jadex Plattform Architektur. Quelle: [Braubach u. a. (2013)]

Abbildung 4.1 zeigt die Architektur von Jadex AC. Hier wird ersichtlich, dass von der Java SE abgesehen keine weitere Middleware benötigt wird, es allerdings ermöglicht wird, andere Middlewares per Adapter und Platform Services einzubinden. Durch die lose Kopplung der verschiedenen Kernel und der Platform und ihren Services, werden heterogene Applikationen ermöglicht, die aus unterschiedlichen Komponententypen bestehen können [Braubach und Pokahr (2013)]. Von den unterstützten Typen sind BDI und Micro von besonderem Interesse, da beides agentenbasierte Komponenten sind. Der BDI-Kernel ermöglicht die unterstützte Implementation von BDI-Agenten über ein sogenanntes *Agent Definition File* (ADF) in XML, mit dem die Struktur eines BDI-Agenten modelliert werden kann. Weiterhin beinhaltet der Kernel das Reasoning für die BDI-Agenten. Sowohl das ADF, als auch die in plain-Java zu programmierenden Pläne eines Agenten gehören dann zum Application Layer. Der Micro-Kernel ermöglicht die Implementation besonders leichtgewichtiger Agenten in plain-Java, die den JADE-Agenten sehr ähneln. Es ist möglich mehr als 100.000 Micro-Agenten in einer Standard-Desktop-JVM laufen zu lassen [Pokahr u. a. (2010)].

Jadex AC bietet zwei verschiedene Möglichkeiten der (asynchronen) Kommunikation zwischen Komponenten. Zum einen kann die Kommunikation Message basiert sein, wobei die Form der Messages nicht vorgegeben ist. Zum anderen kann sie Remote-Method-Call basiert sein, wobei die Method-Calls immer asynchron sind, um technische Deadlocks zu vermeiden, indem ein *future* genanntes Aufrufkonzept genutzt wird [Pokahr und Braubach (2011); Sutter und Larus (2005)]. Finden diese Method-Calls innerhalb einer VM statt, sind es sogar ganz normale Methodenaufrufe. Jadex ermöglicht es auch direkt, die von Komponenten angebotene Services

als RESTful bzw. SOAP-Webservices zur Verfügung zu stellen bzw. externe Services auf interne zu mappen und somit transparent zu konsumieren.

Von großer Relevanz für die technische Analyse ist die von Jadex AC angebotene Simulationsinfrastruktur, welche in [Braubach und Pokahr (2013)] ausführlich beschrieben ist. Diese ermöglicht es, Anwendungen als event-basierte Simulationen oder auch in Echtzeit laufen zu lassen, ohne dafür in der Anwendungslogik Änderungen vornehmen zu müssen. Hierzu gibt es einen simulationsspezifischen Simulation Service, der sich in die Reihe der zur Ausführung einer Komponente nötigen Services einreiht.

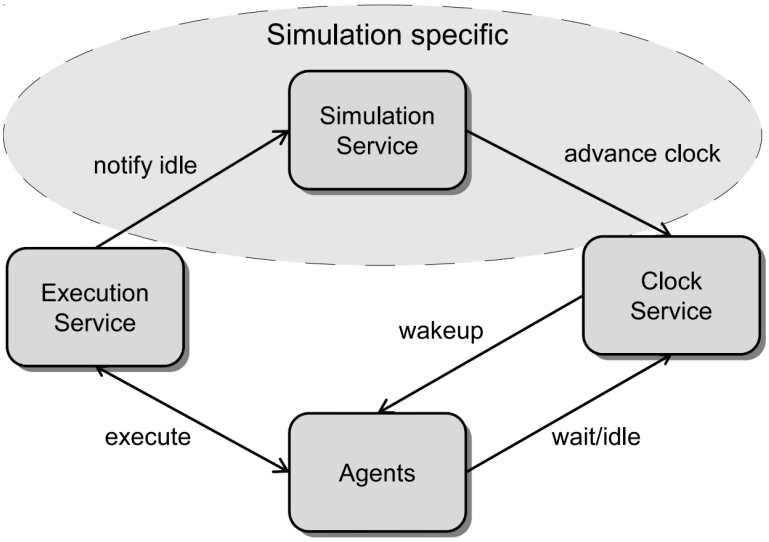


Abbildung 4.2: Jadex Simulations Infrastruktur Komponenten. Quelle: [Braubach und Pokahr (2013)]

Abbildung 4.2 zeigt das Zusammenspiel dieser Services. Während der Clock Service im Falle der Echtzeitausführung automatisch wartende/leerlaufende Agenten zu bestimmten Zeitpunkten weckt und diese dann ihre Ausführung beim Execution Service anmelden, wird im Falle der event-basierten Simulation ein Simulation Service zwischen Execution und Clock Service eingeführt. Dieser wartet darauf, dass ihm der Execution Service meldet, dass alle Komponenten, die für den aktuellen Zeitschritt eine Ausführung beantragt haben, diese beendet haben. Daraufhin kann er den Clock Service um einen Zeitschritt voranschreiten lassen. Somit ist es einzig notwendig *wait*-Kommandos in den Anwendungscode zu integrieren, die allerdings ohnehin für die Jadex-Programmiermetapher notwendig sind.

Weiterhin weist Jadex AC eine hohe Interoperabilität auf, da es auf Java basiert und somit auf allen Betriebssystemen, auf denen eine JVM läuft, einsatzfähig ist. Sogar auf Android basierten, mobilen Geräten ist eine eingeschränkte Jadex AC Version lauffähig.

Ferner wird die Plattform fortlaufend weiter entwickelt, es gibt eine Online-Dokumentation sowie einen regen Austausch mit den Entwicklern. Außerdem gibt es eine lange Reihe von Publikationen, die sich mit den Konzepten von Jadex AC befassen.

Einer Entwicklung des Simulationssystems unter Nutzung von Jadex AC stehen somit keine Argumente im Weg, alle Anforderungen werden abgedeckt. Insbesondere die Möglichkeit, mehr als 100.000 Micro-Agenten innerhalb einer Desktop-VM ausführen zu können, hebt sich hervor.

4.2.2 PlaSMA

Das *Platform for Simulations with Multiple Agents* (PlaSMA)-Framework[8] bietet eine event-basierte Simulationsumgebung auf Basis der JADE-Multiagentenplattform. Diese ist zu den von der *Foundation for Physical Intelligent Agents*[9] (FIPA) definierten Standards zur Interaktion von Agenten [Poslad (2007)] konform. Ursprünglich wurde sie für Logistiksimulationen vom *Sonderforschungsbereich 637*[10] an der Universität Bremen entwickelt und erweitert JADE um ein diskretes, für Nutzer transparentes Simulations-Zeitmanagement [Schuldt u. a. (2008)].

Abbildung 4.3 verdeutlicht das Verteilungs- und Simulationskonzept von PlaSMA. Es gibt pro Simulation einen TopController und in Abhängigkeit von diesem für jeden Rechner bzw. Prozessor einen SubController, der für alle auf diesem Knoten laufenden Agenten verantwortlich ist. Diese Verantwortlichkeit beinhaltet hierarchisch das Lifecycle-Management, Laufzeitkontrolle und die Zeit-Events, mit denen Agenten geweckt werden [Warden u. a. (2010)]. Ähnlich wie bei Jadex AC soll der Entwickler auch hier keinen simulationsspezifischen Quelltext in die Agenten implementieren müssen, um die Agenten ohne Änderungen auch operativ einsetzen zu können, bzw. um bereits operativ eingesetzte Agenten minimal-invasiv in eine Simulation integrieren zu können.

Da die um das Zeitmanagement angereicherte Kommunikation zwischen Agenten in PlaSMA auf JADE basiert, steht hier eine Message basierte Kommunikation zur Verfügung, die FIPA-

[8] http://plasma.informatik.uni-bremen.de/
[9] http://www.fipa.org/
[10] http://www.sfb637.uni-bremen.de/

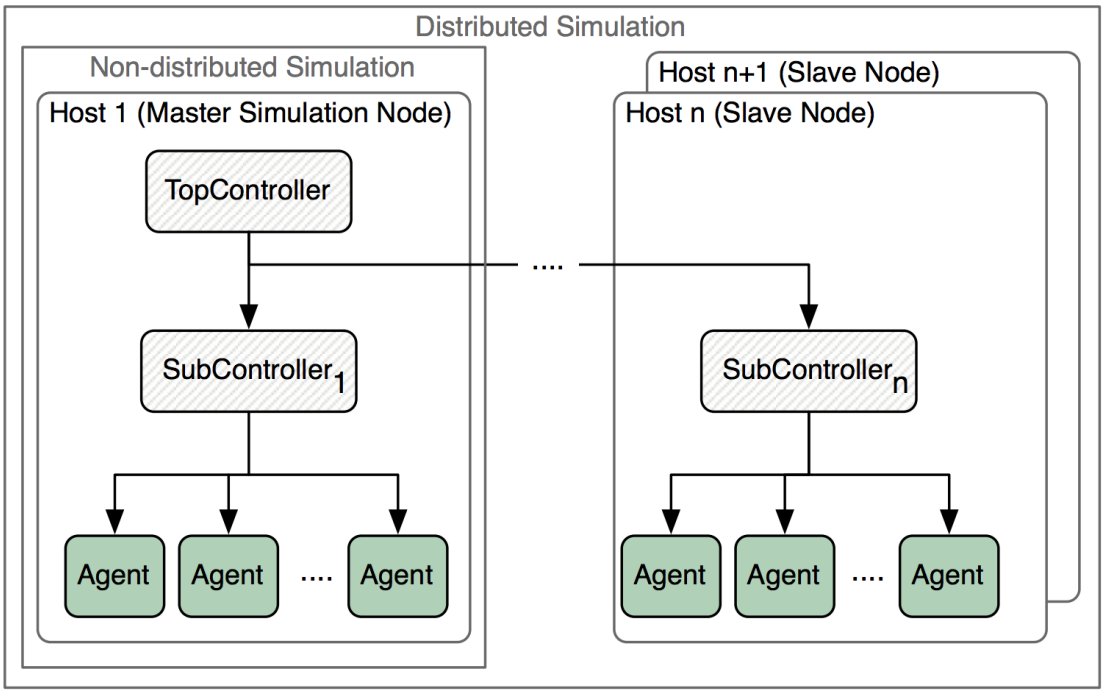


Abbildung 4.3: PlaSMA Simulation Control Model. Quelle: [Warden u. a. (2010)]

konform ist. Diese Messages basieren auf XML, auch Datentypen die verschickt werden, müssen serialisiert werden.

Auch PlaSMA ist rein Java basiert und damit äußerst interoperabel; ob für mobile Geräte lauffähige Versionen existieren, konnte nicht herausgefunden werden, ist aber durch die Nutzung von Java denkbar.

Ferner existiert eine Online-Dokumentation der Middleware und einige Publikationen sind zur finden. Insgesamt ist aber die Architektur der Plattform etwas spärlich dokumentiert, viele Mechanismen bleiben in ihrer Funktionsweise unklar.

Es konnten weder für PlaSMA noch für das zugrundeliegende JADE Publikationen gefunden werden, die sich mit massiven Simulationen $\geq$100.000 Agenten beschäftigen. Allerdings betrachtet z.B. [Mengistu u. a. (2008)] kritisch die Skalierbarkeit von JADE für den Einsatz von sehr vielen Agenten und identifiziert unter anderem den Directory Service von JADE als Bottleneck.

4.2.3 Auswahl der Simulationsplattform

In den Abschnitten 4.2.1 und 4.2.2 wurden zwei Simulationsplattformen näher betrachtet. Es wurde ein Augenmerk auf Aspekte gelegt, die in Bezug auf die erhobenen Anforderungen und den Erfolg des Projektes von großer Wichtigkeit sind.

In Tabelle 4.2 werden die elementarsten Punkte für diese zwei Plattformen zusammengefasst.

Simulations-plattform	**Simulationssupport: Event basiert / Echtzeit**	**Kommunikation: Message basiert / RMI basiert**	**massive Simulationen**	**Doku**
Jadex AC	☒ / ☒	☒ / ☒	☒	☒
PlaSMA	☒ / ☒	☒ / ☐	☒[11]	☒

Tabelle 4.2: Übersicht Simulationsplattformen

Sowohl Jadex AC als auch PlaSMA sind Java basierte Plattformen, was sie interoperabel einsetzbar macht. Dies kann für größere verteilte Simulationen auf heterogener Hardware bzw. auf heterogenen Betriebssystemen von Bedeutung sein. Aber auch die mögliche Verwendbarkeit von Standardtools und -bibliotheken, sowie die Wiederverwendbarkeit bestehenden Codes sind dadurch abgedeckt. In Kapitel 4.1 wurde gezeigt, dass es mit Java unter Verwendung von JNA problemlos möglich ist, die Matlab Simulink einzubinden.

Beide Plattformen bieten einen Simulationssupport für Event basierte und Echtzeit-Simulationen und versuchen dabei, den Implementationsaufwand für den Entwickler gering oder möglichst sogar völlig transparent zu halten.

Einen Vorteil bietet Jadex AC gegenüber PlaSMA auf Seiten der Kommunikation. Neben der asynchronen Message basierten Kommunikation, die auch von PlaSMA angeboten wird, wird auch eine asynchrone Remote-Method-Call basierte Kommunikation zur Verfügung gestellt, die innerhalb von Plattformgrenzen sogar transparent durch schnelle und dennoch asynchrone Methodenaufrufe bewerkstelligt wird. Denn das Kommunikationsaufkommen kann für massive Simulationen, bei denen jedes Modell pro simulierter Minute eine große Datenmenge an die nächste Ebene übergeben muss, sehr stark anwachsen. Dabei kann es für die Performanz der Simulationen entscheidend sein, ob bei der Übergabe eines Objektes innerhalb der Plattformgrenzen nur dessen Zeiger benötigt wird oder das Objekt erst serialisiert, in eine Nachricht verpackt und vom Empfänger wieder entpackt werden muss.

[11] eingeschränkt, weiterführende Evaluation wäre notwendig

Ein weiterer Punkt, der für Jadex AC spricht, ist die massive Instanzierbarkeit von Agenten respektive Komponenten. Es lassen sich Publikationen finden, in denen die gleichzeitige Ausführung von über 100.000 Jadex Micro-Agenten aufgezeigt wird [Pokahr u. a. (2010)]. Im Fall von PlaSMA bzw. JADE findet man lediglich Aussagen zu mehreren tausend Agenten. Da Jadex AC einen eigenen Thread-Pool und -Management besitzt, kann hier auch die Anzahl von parallelen Threads begrenzt werden, was hinsichtlich vorhandener Ressourcen und genutztem Betriebssystem vorteilhaft sein kann. Auch wird an anderer Stelle der Directory Service von JADE als mögliches Bottleneck beschrieben [Mengistu u. a. (2008)].

Für beide Plattformen existiert eine umfangreiche Online-Dokumentation und es gibt eine Vielzahl an Publikationen. Die PlaSMA-Plattform wird allerdings hinsichtlich Architektur und Technik weniger detailliert beschrieben.

Als Schlussfolgerung dieses Plattformvergleichs bezüglich der vorher aufgestellten Anforderungen an die Simulationsplattform fällt die Wahl auf Jadex Active Components, da es hinsichtlich der Skalierbarkeit für massive Simulationen am besten aufgestellt zu sein scheint. Hinzu kommen bestehende Erfahrungen sowohl mit der Vorgängerversion Jadex v1, mit der in einem spielähnlichen, kompetitiven Szenario eine koordiniert agierende Gruppe virtueller Cowboy-Agenten implementiert wurde [Balthasar u. a. (2010)], als auch mit der aktuellen Version, deren Programmiermetapher trotz einer vöölig neuen Code-Basis ähnlich geblieben ist. Dementsprechend wird diese Plattform und somit Java als Programmiersprache für die weitere technische Analyse als gesetzt angenommen.

4.3 Skalierbarkeit der Simulink-Modell-Integration

Nachdem grundlegende Programmiersprache und Simulationsplattform ausgewählt wurden, soll für diese jetzt geprüft werden, ob und wie eine den Anforderungen genügende Skalierbarkeit der Simulink Modelle herzustellen ist.

An dieser Stelle muss zuerst ein Schritt zurück in die Matlab Simulink Welt und zum Export der Modelle gemacht werden. Ein Simulink Modell enthält typischerweise Halteglieder, die bestimmte Werte für den nächsten Simulationsschritt vorhalten. Würde ein solches Modell unmodifiziert exportiert, entstünde eine zustandsbehaftete DLL, deren Datenhaltung eigenständig und von der Simulationsplattform gekapselt wäre. Dementsprechend würden auch für 100.000 Simulink Modelle in der massiven Simulation 100.000 DLLs benötigt, jede mit eigener Speicherhaltung. Davon abgesehen, dass nach ersten Testläufen mit Java und JNA

unter Windows nicht mehr als 117 verschiedene DLLs innerhalb eines Java-Prozesses registriert werden konnten[12] (auch publiziert in [Braunagel u. a. (2013)]), wäre der auch von der Simulationsplattform nicht organisierbare Speicheroverhead durch bis zu 100.000 DLLs nicht hinnehmbar.

Von daher ist es auf Matlab Simulink Seite nötig, die Modelle anzupassen. Dazu muss auf jegliche Halteglieder verzichtet werden und es müssen Ein- und Ausgänge für vorzuhaltende Daten hinzugefügt werden, um zustandslose, im Folgenden *Reentrant-DLLs* genannte Modellexporte zu ermöglichen.

Wenn nun aber nur noch eine DLL, oder ggf. zwei, da es ja projektbedingt zwei grundsätzlich verschieden beheizte Haushaltsmodelle gibt, vorhanden ist, die gleichzeitig von vielen tausend Haushalts-Agenten genutzt werden soll, könnte dies ein potenzielles Bottleneck für Multi-Core-Prozessoren darstellen. Denn die von Simulink generierte DLL kann lediglich single-threaded genutzt werden, da der Zugriff thread-safe erfolgen muss. Da als Hardwareausführungs-Plattform für das Projekt ein 64-Core Server mit 256GB RAM bereitgestellt werden soll, muss diese Problematik analysiert werden.

Ein Lösungsansatz besteht in der Duplizierung der DLL, so dass so viele DLLs vorhanden sind, wie auch Kerne im System. Um dies zu testen wird ein Prototyp eingesetzt. Dieser Prototyp wird bereits mit Jadex AC realisiert und soll schon grundlegend die angestrebte Struktur aus Dispatcher/Aggregator, Sub-Aggregatoren und den diesen zugeordneten Haushalts-Agenten abbilden, um einerseits unter Umständen als evolutionärer Prototyp weiterentwickelt zu werden und um andererseits auch die Auswirkung der hierarchischen Synchronisierung auf die Skalierbarkeit abschätzen zu können. Die Haushalts-Agenten sollen auf eine parametrierbare Zahl von duplizierten DLLs zugreifen und deren Step-Funktion 1.440 Mal aufrufen, was einem simulierten Tag entspricht, wenn die Step-Größe den Anforderungen entsprechend eine Minute beträgt. Nach jedem Schritt werden die Daten dann an einen zugeordneten Sub-Aggregator-Agenten weitergegeben, der die aggregierten Daten an den Aggregator-Agenten leitet, bevor ein nächster diskreter Schritt ausgeführt wird. Die Ausführung erfolgt Event basiert so schnell wie möglich. Dabei wird von einem Kontroll-Agenten bei Start und Ende die Zeit genommen.

Die reine Duplikation der DLLs hilft allerdings nicht, diese müssen auch gleichmäßig auf die Haushalts-Agenten aufgeteilt werden. Da das System in der Initialisierungsphase hierarchisch gestartet wird, sollen nun die Sub-Aggregator-Agenten den Haushalts-Agenten das Interface der zu nutzenden DLL beim Start mitgeben. Dafür werden zwei Algorithmen präsentiert, die

[12] Unter Linux und MacOS besteht dieses Problem nicht

beide eine das Singleton-Pattern implementierende Klasse als Ressource benötigen. Diese ermittelt während ihrer Initialisierung die Anzahl der zur Verfügung stehenden Prozessor-Kerne und errechnet ein Verhältnis zwischen Kernen und DLL-Typen, zu sehen in Quelltext 4.5.

```
// Ermittlung der zur Verfuegung stehenden Kerne
int cores = Runtime.getRuntime().availableProcessors();

// Duplikationsverhaeltnis Kerne/DllTypen
if (cores % getNumberOfDifferentDLLs() != 0) {
    amountOfDuplicatedDllsPerType = (cores / getNumberOfDifferentDLLs()) +
        1;
}
else {
    amountOfDuplicatedDllsPerType = cores / getNumberOfDifferentDLLs();
}
```

Quelltext 4.5: Auszug 1 aus SingleValues.java - Initialisierung

```
/**
 * Gibt synchronisiert die naechste Zufallszahl in einem Intervall
 * zurueck. (inkl. 0, exkl. interval)
 *
 * @param interval Intervall fuer Pseudezufallszahl
 * @return randomInt Zufallszahl
 */
public synchronized int getNextRandomDllNumber() {
    return this.random.nextInt(amountOfDuplicatedDllsPerType);
}

/**
 * Gibt in einem Ring synchronisiert die naechste dupl. DLL-Nummer
 * zurueck.
 *
 * @return int nextDllNumber - Nummer der zugewiesenen dupl. DLL
 */
public synchronized int getNextDllNumber() {
    if (nextDLL < amountOfDuplicatedDllsPerType) {
        return nextDLL++;
    }
    else {
        nextDLL = 0;
        return nextDLL++;
    }
}
```

Quelltext 4.6: Auszug 2 aus SingleValues.java - Öffentliche Funktionen

Quelltext 4.6 zeigt in einem weiteren Auszug die für die beiden Algorithmen nötigen threadsafen Funktionen *getNextRandomDllNumber()*, die die nächste im Intervall [0; Verhältnis Kerne/DLL-Typen[liegende Pseudozufallszahl mittels der einmalig instanzierten Random-Klasse liefert und durch den Singleton-Charakter sicherstellt, dass eine Normalverteilung der Zufallswerte vorliegt, und *getNextDllNumber()*, die mittels einer Instanz-Zählvariable in einem Zahlenring des oben genannten Intervalls immer die nächste Zahl zurückgibt.

```
SingleValues singVa = SingleValues.getInstance();

String assignedDllName = dllType;

// Auswahl: Algorithmus 1 oder 2
int algorithm = 1;

int customersPerSubAggregator = singVa.getCustomersPerSubAggregator();

int amountOfDuplicatedDllsPerType = singVa.getAmountOfDuplicatedDllsPerType
    ();

int multiCoreHelper = (int)(customersPerSubAggregator %
    amountOfDuplicatedDllsPerType);

for (int i = 0; i < customersPerSubAggregator; i++) {
    if (algorithm == 1) {
        if (i <= customersPerSubAggregator - multiCoreHelper) {
            assignedDllName += i % amountOfDuplicatedDllsPerType;
        }
         else {
            assignedDllName += singVa.getNextRandomDllNumber();
         }
    }
    else if (algorithm == 2) {
        assignedDllName += singVa.getNextDllNumber();
    }
}
```

Quelltext 4.7: Auszug aus SubAggregatorAgent.java

Der in Quelltext 4.7 bereitgestellte Auszug aus dem Subaggregator-Agenten zeigt die Funktionsweise beider Algorithmen, die sich anfangs auswählen lassen. Algorithmus 1 basiert darauf, dass, gesetzt es sind zwar mehr Haushalt-Agenten als das Verhältnis der duplizierten DLLs, diese jedoch ein Teiler der Anzahl der Haushalt-Agenten, einfach mittels des *modulo*-Operators eine gleichmäßige Verteilung hergestellt wird. Gibt es allerdings einen Rest, muss sichergestellt werden, dass nicht alle Subaggregator-Agenten dieselben DLLs auswählen. Hierfür wird die von der Singleton-Klasse angebotene Funktion *getNextRandomDllNumber()* genutzt, die

für die übrigen Haushalt-Agenten eine Normalverteilung der DLLs herstellen soll (je mehr Haushalt-Agenten im System sind, desto besser sollte es funktionieren). Für den Fall, dass ein SubAggregator-Agent weniger Haushalte zugeordnet bekommen hat, als das Verhältnis der duplizierten DLLs, wird nur die Random-Funktion genutzt.

Algorithmus 2 basiert auf dem oben beschriebenen Zahlenring und der per *synchronized* hergestellten Thread-Safety der Funktion, so dass auch bei einer kleineren Anzahl von Haushalt-Agenten eine optimale Nutzung der DLLs hergestellt wird.

Für den Fall, dass wirklich zwei DLL-Typen im System vorhanden sind und deren Anzahl unterschiedlich ist, muss die Anzahl noch in ein Verhältnis gesetzt und dieses auf die Zahl der duplizierten DLLs pro DLL-Typ angewandt werden.

Die Tests mit dem Prototypen werden auf einem Workstation-PC mit 4 Kernen a 4,5GHz mit Hyperthreading und 16 GB RAM unter Windows 7 64bit mit der Oracle 64bit JVM durchgeführt, da der Multi-Core-Server zu diesem Zeitpunkt noch nicht verfügbar war. Für erste Testläufe wird die automatische Erkennung der nutzbaren Kerne überschrieben und Läufe mit jeweils 1, 2, 4, 6, 8, 16, 32 und 64 duplizierten DLLs für jeden Algorithmus durchgeführt und für jedes Szenario wird der Mittelwert aus zehn Läufen gebildet. Es wird nur ein DLL-Typ getestet.

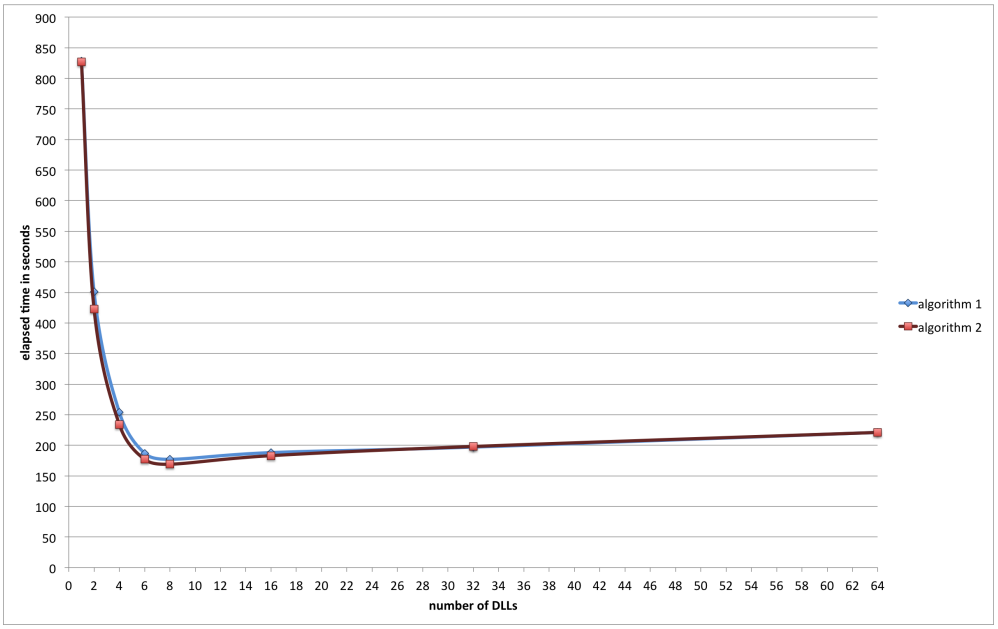


Abbildung 4.4: Auswirkung unterschiedlicher DLL-Anzahl für beide Algorithmen bei 1.000 Haushaltsagenten

Abbildung 4.4 zeigt die Auswirkung der Nutzung unterschiedlich vieler duplizierter DLLs. Hierbei ist klar ersichtlich, dass das System von einer Duplikation der DLLs profitiert. Das Optimum duplizierter DLLs wird durch die Anzahl virtueller Kerne des PCs markiert und ist um den Faktor 4,9 schneller als die Nutzung nur einer DLL. Hier wird der Vorteil von Hyperthreading ersichtlich, das System hat nur vier „echte“ Kerne, dennoch kann ein Beschleunigungs-Faktor > 4 durch Nutzung der virtuellen Kerne erzielt werden. Erhöht sich die Zahl der DLLs aber über die Anzahl virtueller Kerne, verlangsamt sich das System wieder etwas. Eine mögliche Erklärung ist der Overhead, der durch die Mehrfachregistrierung der DLLs entsteht und sich aber für den Prozessor kein zusätzlicher Nutzen daraus ergibt, er viel mehr immer wieder eine andere DLL in den Cache laden muss. Weiterhin ist ersichtlich, dass Algorithmus 2 leichte Vorteile gegenüber Algorithmus 1 bei 1.000 Haushalt-Agenten hat.

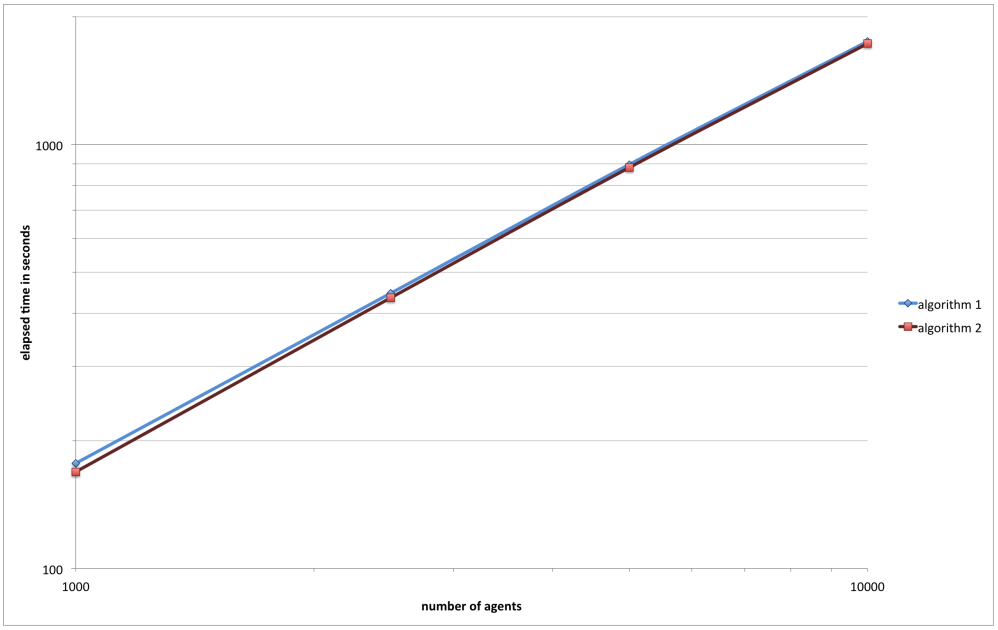


Abbildung 4.5: Veranschaulichung der Skalierung per doppelt logarithmischem Vergleich der Algorithmen bei unterschiedlicher Agentenzahl

Für die Ergebnisse in Abbildung 4.5 wird bei gleichbleibender DLL-Anzahl (8) die Zahl der Haushalt-Agenten von 1.000 über 2.500 und 5.000 auf 10.000 erhöht. Die doppelt logarithmische Darstellung zeigt auf, dass das System bei Erhöhung der Haushalt-Agenten für die Ausführungszeit linear ziemlich exakt mit dem Faktor der Erhöhung skaliert. Bei 10.000 Haushalt-Agenten dauert die Simulation von 1440 Sim-Minuten etwa 29 tatsächliche Minuten, es erreicht also eine Beschleunigung um den Faktor 50. Sollte das System auch bei mehr als 10.000 Agenten

weiterhin genauso skalieren, würde das 290 Minuten für 100.000 Agenten bedeuten, was nur noch einem Faktor von 5 entspräche. Mithilfe des 64-Core Servers sollte sich dieser Faktor allerdings wieder erhöhen lassen.

Eine weitere interessante Beobachtung ist das Konvergieren beider Algorithmen für höhere Agentenzahlen. Dies stimmt mit der vorher geäußerten Vermutung überein, dass Algorithmus 1 mit größer werdender Anzahl von Agenten durch die Normalverteilung der Pseudo-Zufallswerte immer besser funktioniert.

Damit ist das Problem der Skalierung von Seiten der Performanz betrachtet und eine Lösung konnte gefunden werden. Allerdings erschwert die nötige Duplikation den Workflow im Projekt. Wie in Quelltext 4.1 in Kapitel 4.1.1 auf Seite 21 gezeigt, benötigt jede duplizierte DLL ein Interface, über das sie registriert wird und ihre Funktionen aufrufbar gemacht werden. Gibt es jetzt Änderungen z.B. an der Signatur von Funktionen der DLL, müssen diese in allen Interface-Klassen eingepflegt werden, was einen großen Zeitaufwand bedeutet.

Dazu wird im folgenden Abschnitt ein Weg propagiert, mit dem dieser Problematik begegnet werden könnte.

4.3.1 Einsatz einer Template-Engine

Um sowohl eine performante Simulation aber auch schnelle Änderungen zu ermöglichen, wird an dieser Stelle der Einsatz einer Template-Engine vorgeschlagen. Normalerweise wird eine solche Template-Engine für die Webentwicklung genutzt, um den Programm-Code vom HTML-Code zu trennen und somit nach dem *Model-View-Controller* (MVC)-Modell entwickeln zu können. In Java gibt es zum Beispiel das etablierte Apache Velocity Project[13], das die Velocity Template-Engine beinhaltet und eine Alternative zu Java Server Pages und PHP bietet.

Die Idee an dieser Stelle allerdings ist es, die Template-Engine zur Erstellung der Java-DLL-Interfaces zu nutzen. Da diese Interfaces bis auf die genutzte DLL gleich sind, würde sich hier ein Template anbieten. Somit müssten Änderungen nur an einer Stelle stattfinden. Weiterführend könnte dieser Vorgang automatisiert und beim Start des Simulationssystems ausgeführt werden. Nach Erstellung der Java-DLL-Interfaces könnten diese zur Laufzeit kompiliert werden und per Java-Reflection dann von den Haushalt-Agenten aufgerufen werden. Abbildung 4.6 veranschaulicht das Zusammenspiel der verschiedenen Komponenten.

[13] http://velocity.apache.org/

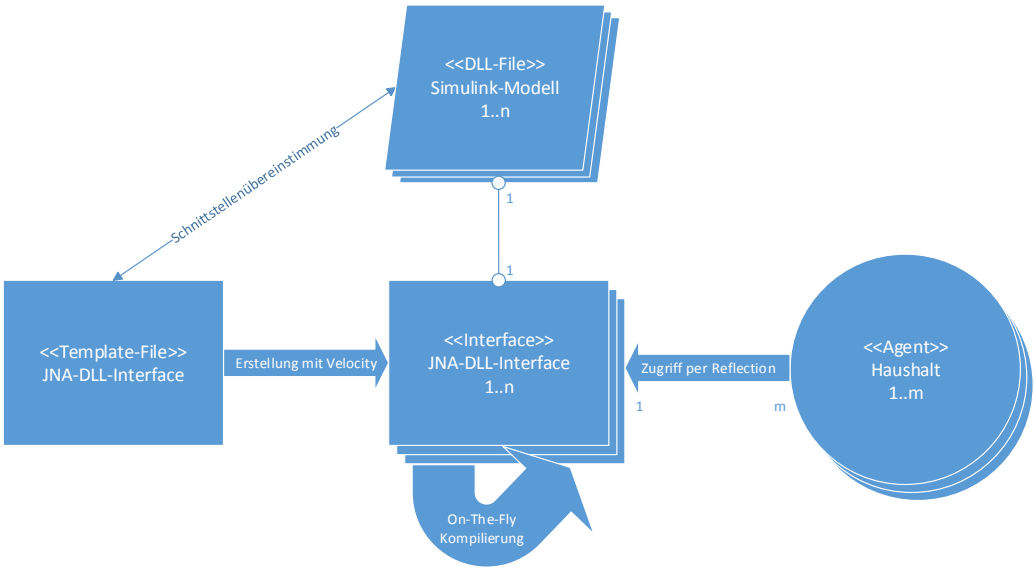


Abbildung 4.6: Nutzung der Template-Engine

4.4 XML-Konfigurationsdateien

Ein weiterer essentieller Punkt der Anforderungen ist die Erstellung von Simulationsszenarien in Form von XML-Dateien. Die Vorgabe XML wurde gewählt, da XML auch für Nicht-Programmierer leicht zu lesen ist. Auch enthalten mittlerweile alle gängigen Programmiersprachen ausgefeilte Möglichkeiten XML einfach zu handhaben. Da Java als Programmiersprache gesetzt ist, wird im Folgenden eine für diese Form von Konfigurationsdateien in Frage kommende Handhabungsform betrachtet.

4.4.1 Binding-Framework JAXB

Das *Java Architecture for XML Binding* (JAXB)-Framework ermöglicht das Binden von XML-Schemata und deren Java-Repräsentationen und ist Bestandteil der Java SE.

Abbildung 4.7 zeigt den Data Binding Prozess von JAXB. Dabei wird gezeigt, dass ein XML-Dokument einem gewissen Schema (XSD) folgt, welches wiederum gegenseitig mit den abgebildeten Java-Klassen gebunden ist. Objekt-Instanzen dieser Klassen können entweder aus

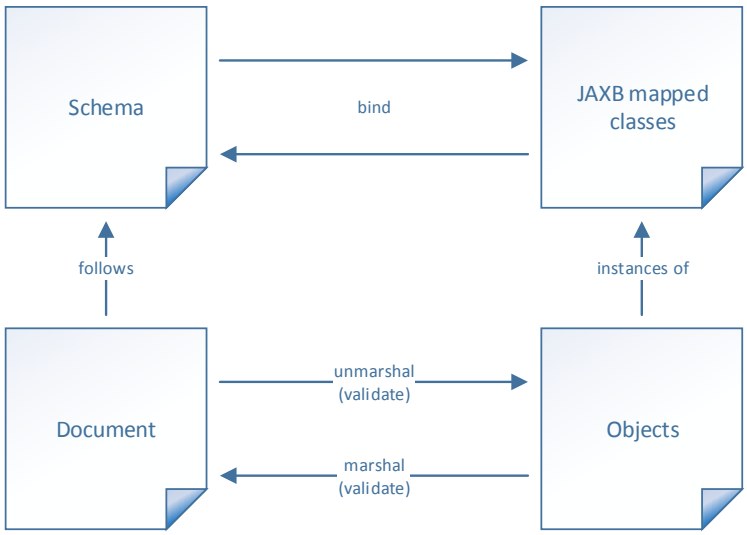


Abbildung 4.7: JAXB Data Binding Prozess, basierend auf [Oracle]

einem XML-Dokument direkt erzeugt und validiert werden (*unmarshalling*) oder andersherum können aus diesen XML-Dokumente generiert und validiert werden (*marshalling*).

Diese Metapher passt sehr gut zu dem gewünschten Modell der XML-Konfigurationsdateien. Ein Schema kann erstellt werden, entweder aus annotierten JAXB Java-Klassen heraus oder direkt als XSD. Auf diesem Schema, das sich unaufwändig ändern lässt, basieren dann die Konfigurationsdateien, aus denen dann zum Simulationsstart Objekte instanziert werden.

4.5 Lesen und Schreiben von .MAT-Dateien

Der letzte Punkt der technischen Analyse beschäftigt sich mit der geforderten Les- und Schreibbarkeit von .MAT-Dateien nach dem Matlab 5 Standard. Die .MAT-Dateien beinhalten von Matlab gespeicherte Matrizen verschiedenen Typs und liegen in binärem (nicht menschenlesbaren) Format vor. In [MathWorks] wird dieses Format beschrieben. Es gibt bestehende Bibliotheken von MathWorks für den .MAT-Datei I/O in C und C++. Auch in Java gibt es eine freie Third-Party-Library namens *JMatIO*, die nachfolgend kurz beschrieben und getestet wird.

4.5.1 JMatIO

Die JMatIO-Library ermöglicht sowohl das Lesen, als auch das Schreiben von .MAT-Dateien nach dem Matlab 5 Standard. Dabei werden folgende Matrizentypen unterstützt:

- Double-Matrizen
- UInt8, Int8-Matrizen
- UInt64, Int64-Matrizen
- Char-Matrizen
- Structure-Matrizen
- Cell-Matrizen
- Sparase-Matrizen

Für das Projekt werden nur Double-Matrizen benötigt. Diese können laut Beschreibung mit JMatIO gelesen und geschrieben werden.

Quelltext 4.8 zeigt einen Auszug aus der Klasse *MatReadWrite.java*, die zum Test von JMatIO erstellt wurde. Dazu wird eine Double-Matrix aus einer projekttypischen .MAT-Datei eingelesen, zeilenweise ausgegeben und dann werden alle ihre Werte um eins gesenkt. Daraufhin wird eine neue .MAT-Datei mit der veränderten Matrix erstellt. Ein Vergleich mit gleicher Funktionalität in Matlab mit der neu erzeugten Datei ermöglicht dann einen Test der Funktionalität.

```
double[][] testData;

MatFileReader mfr = null;
try {
    mFR = new MatFileReader("testdata.mat");
} catch (IOException e) {
    getLogger().log(Level.SEVERE, e);
}

testData = ((MLDouble)mFR.getMLArray( "Testdaten" )).getArray();

for (int i = 0; i < testData.length; i++) {
    System.out.println(Arrays.toString(testData[i]));
    for (int j = 0; j < testData[i].length; j++) {
        testData[i][j] = testData[i][j] - 1;
    }
}

MLDouble tD = new MLDouble("Testdaten_modifiziert", testData);
ArrayList<MLArray> writeList = new ArrayList<MLArray>();
writeList.add(tD);

try {
    MatFileWriter mFW = new MatFileWriter("testdata_altered.mat", writeList)
        ;
} catch (IOException e) {
    getLogger().log(Level.SEVERE, e);
}
```

Quelltext 4.8: Auszug aus MatReadWrite.java

4.6 Fazit

Es wurden für alle wichtigen Anforderungen und die daraus resultierenden technischen Herausforderungen Lösungswege aufgezeigt. Kritisch bleibt allerdings weiterhin die gleichzeitige Ausführung von 100.000 Haushalt-Agenten, bzw. deren mögliche Zeitbeschleunigung während einer Simulation, da in Ermangelung des leistungsstarken Servers oder einer verteilten Ausführung noch keine qualitativen Aussagen bezüglich dessen Performanz gemacht werden können. Es wird allerdings propagiert zur Kommunikation der Agenten die von Jadex AC angebotene Möglichkeit der Remote-Procedure-Calls zu nutzen. Diese werden innerhalb von Plattformgrenzen automatisch als asynchrone Methodenaufrufe umgesetzt, die keinerlei Serialisierung oder anderweitige Verpackung der übergebenen Daten benötigt. Daraus resultiert ein geringerer Kommunikationsoverhead, was für ein derart großes System sehr wichtig sein kann. Trotzdem besteht aber die Möglichkeit, das System verteilt auszuführen ohne an der Implementation der Kommunikation Änderungen vornehmen zu müssen.

Die als optional angesetzte Möglichkeit des direkten Imports von Matlab-Simulink-Algorithmen konnte aus zeitlichen Gründen nicht analysiert werden und wird somit nicht weiter verfolgt.

5 Design

Nachdem im vorherigen Kapitel die technische Machbarkeit geprüft und Lösungen für die verschiedenen Problemstellungen herausgefiltert wurden, soll in diesem Kapitel auf Basis dieser Erkenntnisse ein Architekturentwurf für das Simulations-Framework vorgestellt und ein Designentwurf für die verschiedenen Agententypen bestimmt werden. Die Entwicklung erfolgt mit Jadex AC, dementsprechend werden zur Modellierung agentenorientierte sowie an die Service Component Architektur angelehnte Methodologien genutzt. Dies ist notwendig, da z.B. UML-Klassendiagramme die Metapher von agentenorientierten oder der Service-Component-Architektur folgenden Systemen nicht adäquat abbilden können. Außerdem wurde ein iterativer Entwicklungsprozess auf Basis eines evolutionären Prototypen gewählt, da auch das nachzubildende Matlab-Simulink Simulationssystems zur gleichen Zeit permanent erweitert und weiterentwickelt wurde. Zuletzt wird noch die Startphase des Systems dargestellt.

5.1 Architektur des Simulations-Frameworks

Da die Architektur des Multiagenten basierten Simulations-Frameworks die Architektur des in Matlab-Simulink bestehenden Simulationssystem nachbilden soll, die an Abbildung 5.1 dargestellt ist, wird vorgeschlagen die dort bestehenden Akteure eins zu eins als Agenten abzubilden.

Der daraus resultierende erste Entwurf einer Architektur wurde mithilfe eines Tropos Dependency-Relationship-Diagramms visualisiert. Tropos ist eine agentenorientierte Methodologie, die Herangehensweisen und Werkzeuge zur Modellierung von Multiagentensystemen bereitstellt [Bresciani u. a. (2004)]. Abbildung 5.2 zeigt diesen Entwurf, der bis auf kleine Änderungen - der Dispatcher war ursprünglich als externes System geplant und die Haushalt-Agenten hießen noch Gebäude-Agenten - für das fertige Simulations-Framework weiter gültig ist.

Dargestellt werden dabei die Abhängigkeiten zwischen den verschiedenen Agententypen. Die Haushalt-Agenten haben dabei Abhängigkeiten zu den Matlab-DLLs sowie zu bestimm-

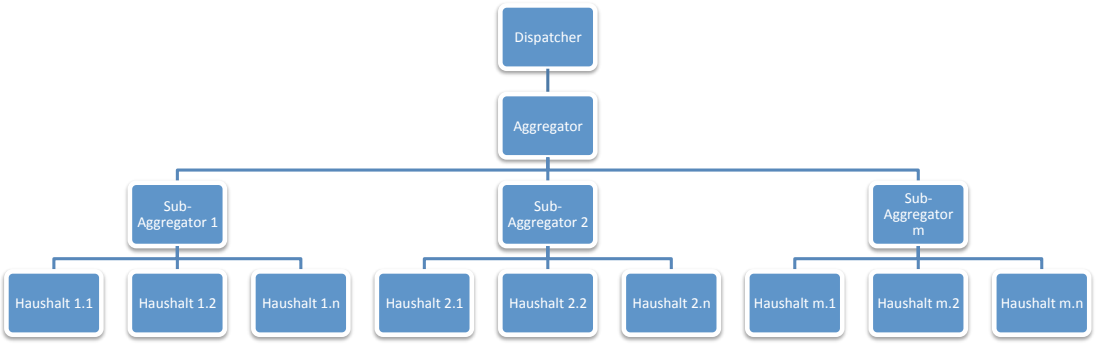


Abbildung 5.1: Architektur der Matlab-Simulink Simulation

ten Umweltparametern, die vom Simulationsszenario vorgegeben werden. Weiterhin haben sie Abhängigkeiten zu Ressourcen, die von den SubAggregator-Agenten angeboten werden, namentlich zu Laständerungsanfragen, Regelungsanweisungen, Anfrage nach dem erzeugten Fahrplan und nach Präferenzsignalen. Dies sind domänenspezifische Daten und werden von der Matlab-Simulink-DLL benötigt. Selbst bieten die Haushalt-Agenten von den DLLs bereitgestellte Daten als Ressourcen an und zwar den über das Präferenzsignal generierten Gebäudefahrplan, die Antwort auf eine Laständerungsanfrage und den Gebäudestatus.

Die SubAggregator-Agenten aggregieren die von den Haushalt-Agenten kommenden Daten und bieten sie selbst wiederum als aggregierte und anonymisierte Ressourcen an. Die Ressourcen, die sie selbst bereitstellen, erhalten sie vom Aggregator-Agenten und verteilen sie über einen aus Matlab in Java überführten Algorithmus an die Haushalt-Agenten.

Der Aggregator-Agent ist äquivalent zu den SubAggregator-Agenten, aggregiert und stellt Daten für den Dispatcher bereit, nimmt Daten vom Dispatcher entgegen und verteilt diese über einen ebenfalls aus Matlab überführten Algorithmus an die SubAggregatoren.

Der Dispatcher-Agent erhält die aggregierten Daten vom Aggregator-Agenten, prozessiert diese auch über einen aus Matlab überführten Algorithmus. In diesem Algorithmus wird die zentrale Regelung des Systems vorgenommen. Weiterhin stellt der Dispatcher-Agent das Präferenzsignal, die Laständerungsanfragen und die Anfrage nach dem Tagesfahrplan für den Aggregator-Agenten bereit.

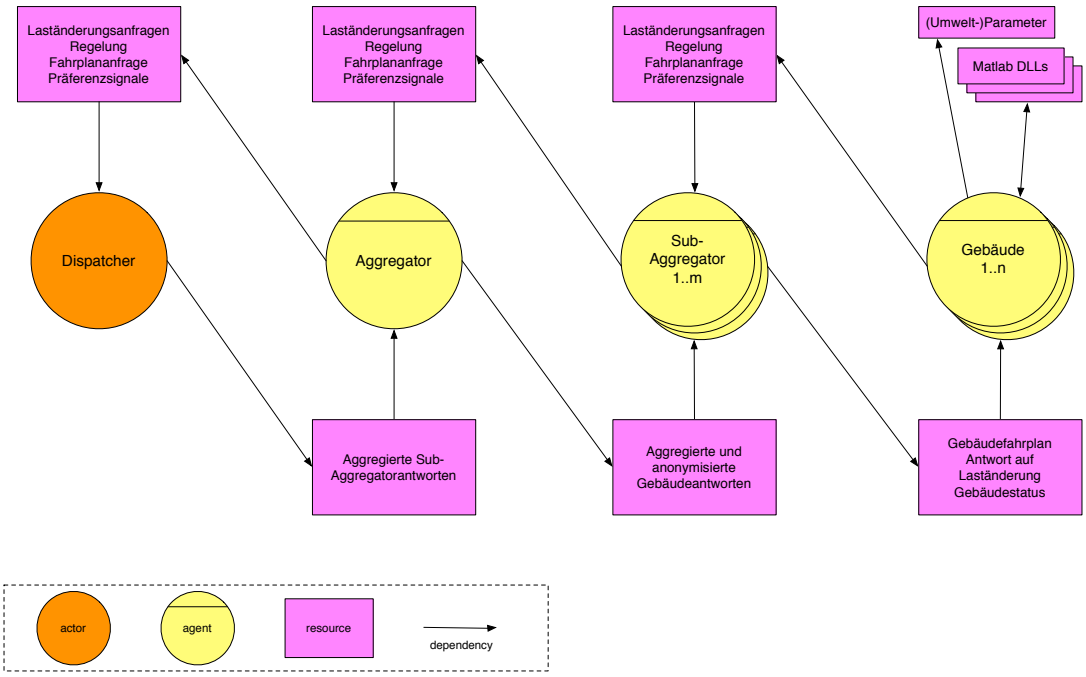


Abbildung 5.2: Architekturentwurf in Form eines Tropos Dependency-Relationship-Diagramms

5.1.1 Architekturverfeinerung für Jadex AC

Die Tropos-Methodologie ist nicht in der Lage, die Programmierparadigmen von Jadex AC abzubilden, da z.B. die Remote-Method-Call basierte Kommunikation nicht Teil von Tropos ist, dass sich an die FIPA Kommunikations-Standards hält. Deswegen wird im Folgenden als Methodik eine Component-Darstellung gewählt, die an die Service-Component-Architektur angelehnt ist (vgl. [Marino und Rowley (2009)]). Diese wird um Jadex-Spezifika, wie z.B. die *IComponentSteps* der Micro-Agenten, erweitert.

In [Dethlefs (2014)] wird diese Methodik eingeführt und noch um Artefakte der *Belief-Desire-Intention* (BDI)-Architektur für pro-aktive Agenten erweitert. Das BDI-Modell für Agenten geht zurück auf [Bratman (1987)], der eine Theorie des menschlichen Schlussfolgerns entwarf, und wurde als Modell für Agenten von [Rao und Georgeff (1995)] verfeinert. Diesem Entwurf folgend haben Agenten eine bestimmte Wissensbasis (Beliefs), ein Set von Zielen (Desires) und Plänen (Intentions), mit denen diese Ziele erfüllt werden können. Diese Art der Architektur wird auch als *Goal-oriented* (Ziel-orientiert) bezeichnet.

In einem ersten Designentwurf des Simulations-Frameworks waren die Agententypen *SubAggregator*, *Aggregator* und *Dispatcher* als BDI-Agenten definiert, da sich die Ziel-orientierte Architektur für diese Agenten besonders gut angeboten hätte. So hätte zum Beispiel ein *Maintain-Goal* des Dispatcher-Agenten lauten können:

- Halte die Last des Gesamtsystems innerhalb eines 5-prozentigen Abweichungskorridors

und zur Umsetzung dieses Ziels hätten unterschiedliche Pläne implementiert werden können.

Allerdings kristallisierte sich bei der prototypischen Entwicklung des Frameworks sehr schnell heraus, dass die rein prozeduralen Matlab-Algorithmen, die in diese Agenten implementiert werden sollten, nur mit hohem Aufwand in eine Ziel-orientierte Form überführt werden konnten. Deshalb wurde von der Nutzung von BDI-Agenten abgesehen.

Zurückkommend auf die in Abbildung 5.2 vorgestellte Architektur nach Tropos, zeigt Abbildung 5.3 jetzt die Architektur in der beschriebenen Methodik.

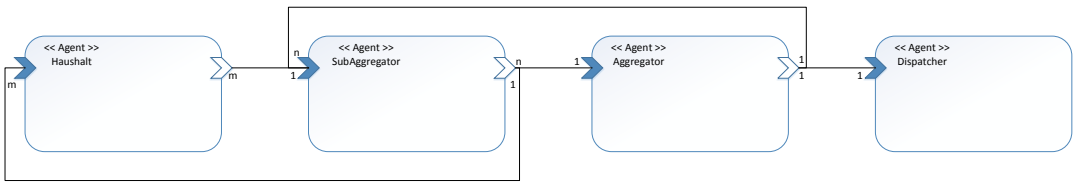


Abbildung 5.3: Architekturentwurf der domänenspezifischen Logik des Frameworks angelehnt an die Service-Component-Architektur

Demnach gibt es eine variable Anzahl von Haushalt- und SubAggregator-Agenten. Das Service-Interface eines SubAggregator-Agenten kann dabei von *m* Haushalt-Agenten referenziert werden, das Service-Interface eines Haushalt-Agenten immer nur von einem SubAggregator-Agenten. Selbiges gilt für die Beziehung zwischen SubAggregator-Agenten und Aggregator-Agent, wobei es allerdings nur einen Aggregator-Agenten gibt. Diese Form des gegenseitigen Aufrufs wurde gewählt, da die Antworten für die in der Hierarchie unten stehenden Agenten von allen über ihnen stehenden Agenten abhängen. Dementsprechend müssten z.B. bei einem Aufruf einer Service-Methode der SubAggregatoren durch alle Haushalt-Agenten die Threads für diese Methode solange offen gehalten (wenn auch im *idle*-Zustand) bis die Antwort des Aggregator-Agenten da ist. Für diese müsste weiterhin ein *Listener* implementiert werden, der diese Threads wieder weckt, da das Konzept der hierarchischen Abhängigkeiten in dieser Form nicht von Jadex abgedeckt wird. Bei der Beziehung von Aggregator- und Dispatcher-Agent

hingegen wird das auf *Futures* basierende asynchrone Antwortkonzept von Jadex genutzt, da diese nur jeweils ein Mal im System vorkommen.

Das UML-Sequenzdiagramm in Abbildung 5.4 veranschaulicht das Zusammenspiel der Agenten nach der Startphase.

Zuerst findet dabei eine von den Haushalt-Agenten ausgehende, kaskadierende Registrierung statt. Sobald diese abgeschlossen ist, findet die erste Ausführung eines DLL-Steps anhand der initialen Parameter statt. Am Ende dieser Ausführung speichern die Haushalt-Agenten interne DLL-Daten für den nächsten DLL-Step zwischen und rufen die *transmitMinuteValues*-Service-Methode der SubAggregator-Agenten auf, wobei sie alle relevanten Daten übergeben. Die SubAggregator-Agenten speichern die empfangenen Daten für jeden einzelnen Haushalt-Agenten zwischen, aggregieren diese und übergeben die Aggregate dann, nachdem sie alle erhalten haben, durch den Aufruf der *transmitSubAggregatedMinuteValues*-Service-Methode des Aggregator-Agenten an eben jenen. Der Aggregator funktioniert auf dieselbe Art und Weise mit dem Unterschied, dass er bei der Übergabe des Aggregats an den Dispatcher auf eine Antwort der *transmitAggregatedMinuteValues*-Service-Methode von diesem wartet. Der Dispatcher-Agent wertet die aggregierten Daten des gesamten Systems anhand einer aus dem in Matlab bestehenden Simulationssystem übernommenen Algorithmik aus, bestimmt anhand seiner konfigurierten Parameter, ob ein Regelungseingriff nötig ist und fordert ggf. Lastverlagerungen des Systems an. Diese Daten werden als Antwort an den Aggregator-Agenten gesendet, welcher die Daten auswertet und bei Bedarf - es muss geregelt werden, eine Lastverlagerung steht an etc. - auf die SubAggregatoren aufteilt. Hierfür steht ebenfalls ein aus Matlab überführter Algorithmus bereit. Diese aufgeteilten und aufbereiteten Daten werden durch den Aufruf der *proposeSignals*-Service-Methode des SubAggregator-Agenten übergeben. Dessen Funktionalität ähnelt der des Aggregators im Schritt zuvor, bei Bedarf werden die Daten auf die Haushalt-Agenten aufgeteilt und durch Aufruf der *proposeSignal*-Service-Methode an diese übergeben. Dort werden die betreffenden Daten als weitere Eingangsparameter der DLL gesetzt. Die Jadex Ausführungsplattform registriert selbst, wenn alle Agenten ihre Aufgabe ausgeführt haben und schaltet die Simulationsuhr einen Schritt weiter, woraufhin die nächste DLL-Step Ausführung angestoßen wird.

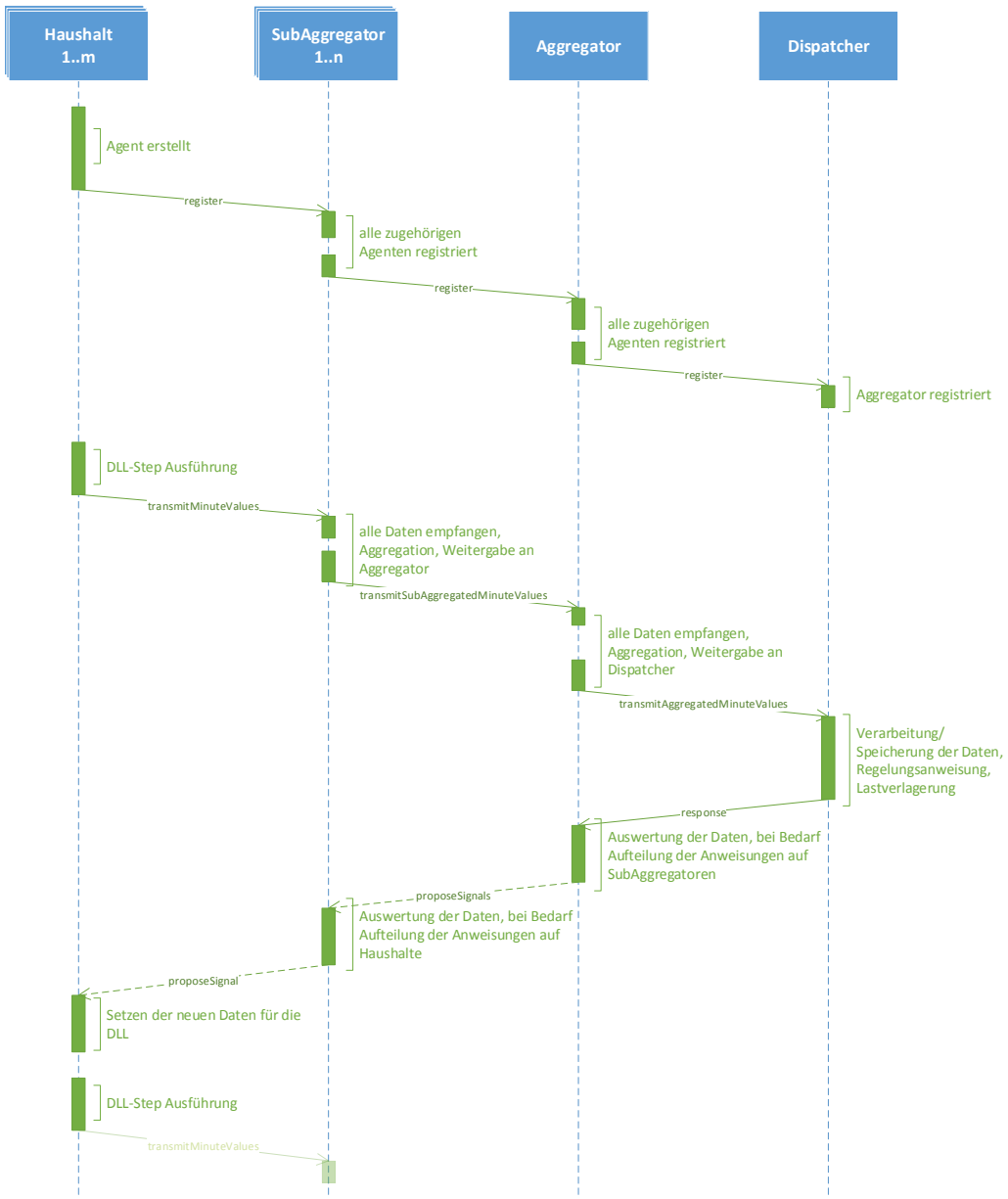


Abbildung 5.4: Zusammenspiel der Komponenten nach der Startphase des Simulationssystems

5.2 Design der Agententypen

In diesem Abschnitt wird das Design der im System vorkommenden Agententypen vorgestellt. Dies geschieht ebenfalls mithilfe der propagierten, erweiterten Service-Component-Architektur Annotation. Die Agententypen werden in der Reihenfolge, in der sie gestartet werden, beschrieben.

5.2.1 Starter-Agent

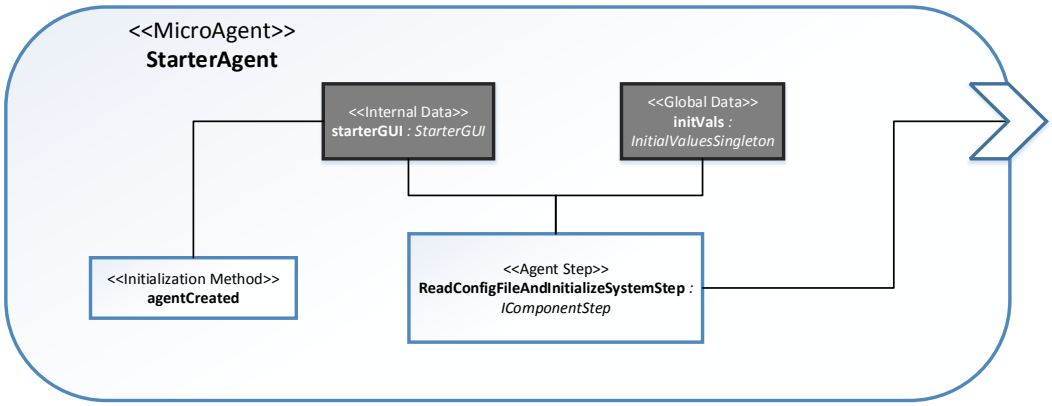


Abbildung 5.5: Design des Starter-Agenten

Der Starter-Agent in Abbildung 5.5 ist der einfachste der vorgestellten Agententypen. Er besitzt kein eigenes Service-Interface und referenziert nur Services der Jadex-Plattform, die benötigt werden, um andere Agenten zu starten. Er hält eine Referenz zur einmal systemweit instanzierten Singleton-Klasse. Diese wird während der Ausführung des *ReadConfigFileAndInitializeSystemStep* mit Startwerten initialisiert. Dieser IComponentStep wird angestoßen von der *StarterGUI*, einer graphischen Bedienungsoberfläche, mit der eine Konfigurationsdatei für den Simulationslauf ausgewählt und dieser gestartet werden kann. Die *StarterGUI* selbst wird instanziert von der *agentCreated*-Methode, die von der Jadex-MicroAgent-Klasse geerbt und überschrieben wird. Zuletzt wird der Dispatcher-Agent gestartet und dabei parametriert.

5.2.2 Dispatcher-Agent

Das Design des Dispatcher-Agenten wird in Abbildung 5.6 dargestellt. Er besitzt ein Service-Interface *DispatcherService* mit den Service-Methoden *registerAggregator* und *transmitAggregatedMinuteValues*. Beide Methoden haben Zugriff auf die interne *aggregatorHold*-Variable. Die *transmitAggregatedMinuteValues*-Methode löst den Start des *ComputeValuesAndGenerateSignalsStep* aus. Dieser *IComponentStep* schreibt Daten in die interne *SimResults*-Variable und kann bei Simulationsende den *WriteResultsToFileStep* initiieren. Weiterhin gibt es die interne Variable *step*, die den aktuellen Simulationsschritt zählt, und eine Referenz auf die Singleton-Klasse *InitialValuesSingleton*, in der z.B. Abrufzeiten für geplante Lastverlagerungen oder ein voreingestelltes Lastband vorliegen. In der *agentCreated*-Methode werden Services der Jadex-Plattform referenziert, um in diesem Fall nach der eigenen Initialisierung den Aggregator-Agenten starten und parametrieren zu können.

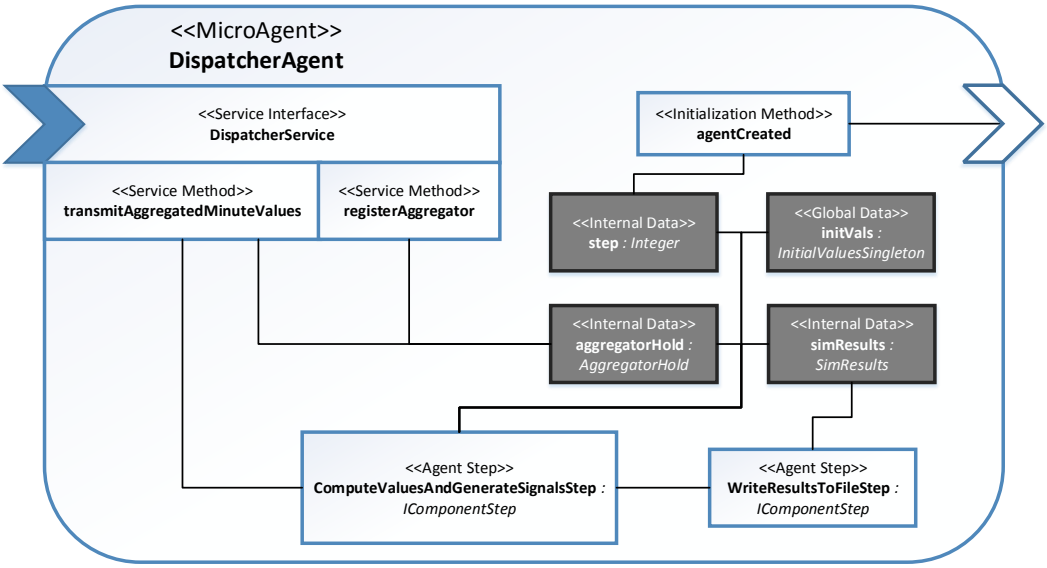


Abbildung 5.6: Design des Dispatcher-Agenten

5.2.3 Aggregator-Agent

Der Aggregator-Agent, dessen Design in Abbildung 5.7 dargestellt ist, besitzt analog zum Dispatcher-Agenten ein Service-Interface mit zwei Service-Methoden. Die Service-Methode *registerSubAggregator* wird für die in Abbildung 5.4 gezeigte Registrierungsphase benötigt;

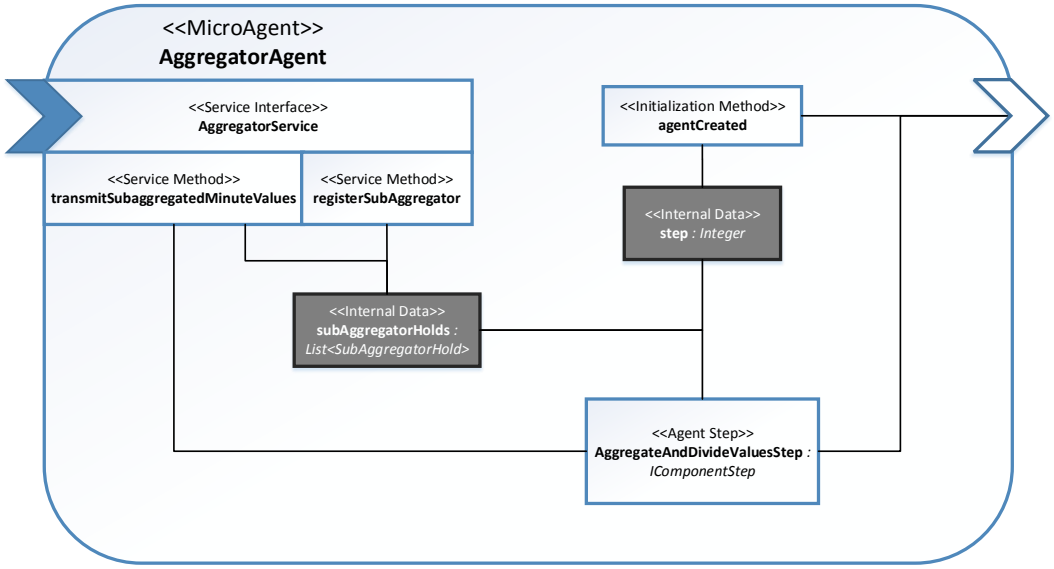


Abbildung 5.7: Design des Aggregator-Agenten

transmitSubAggregatedMinuteValues speichert Daten in der Liste *subAggregatorHolds* zwischen und löst den *AggregateAndDivideValuesStep* aus. Externe Service-Interfaces werden sowohl von diesem *IComponentStep* referenziert (des Service-Interface des Dispatcher-Agenten), als auch von der *agentCreated*-Methode, die Plattform-Services nutzt, um SubAggregator-Agenten zu starten und zu parametrieren.

5.2.4 SubAggregator-Agent

Abbildung 5.8 zeigt das Design des SubAggregator-Agenten. Dieser bietet zusätzlich zu den Service-Methoden *registerCustomer* und *transmitMinuteValues* für die Haushalt-Agenten noch die Service-Methode *proposeSignals* für den Aggregator-Agenten an. Alle drei haben Zugriff auf die *customerHolds*-List, in der die Daten der Haushalt-Agenten für einen Simulationsschritt zwischengespeichert werden. Den Service-Methoden *transmitMinuteValues* und *proposeSignals* ist jeweils ein expliziter *IComponentStep* zugeordnet, der von ihnen bei Bedarf ausgelöst wird. Der *DivideSignalsStep* referenziert wiederum das Service-Interface der Haushalt-Agenten, der *SubAggregateValuesStep* das Service-Interface des Aggregator-Agenten. Die *agentCreated*-Methode referenziert analog zu den vorherigen Agenten Plattform-Services zum Starten der Haushalt-Agenten.

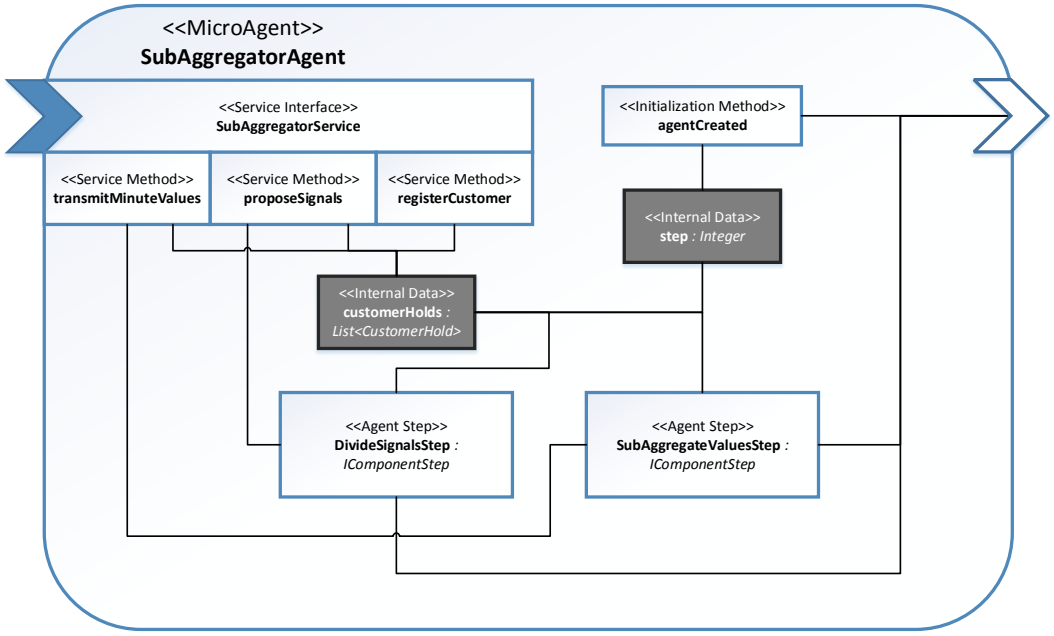


Abbildung 5.8: Design des SubAggregator-Agenten

5.2.5 Haushalt-Agent

Der Haushalt-Agent, in Abbildung 5.9 vorgestellt, besitzt nur die Service-Methode *propose-Signal*, die vom SubAggregator-Agenten aufgerufen wird. Diese hat Zugriff auf die interne *dllHold*-Variable, wodurch Werte für den nächsten DLL-Ausführungsschritt gesetzt werden können. Die *agentCreated*-Methode löst den *PerformNextModelStep* aus, der sich mit der Plattform-Methode *waitForTick* immer wieder selbst aufruft und durch den Simulations-Service der Plattform synchronisert wird. Dieser *IComponentStep* greift über die per Reflection zugewiesene DLL-Interface-Methode *performSyncedDllStep* auf eine der DLLs zu und referenziert das Service-Interface des SubAggregator-Agenten. Ebenfalls in Assoziation steht er mit allen internen Daten des Haushalt-Agenten, sowie der Referenz zur Singleton-Klasse *InitialValues-Singleton*, die verschiedene Umgebungsparameter wie z.B. Wetterdaten oder Zapfprofile für die Haushalt-Agenten vorhält.

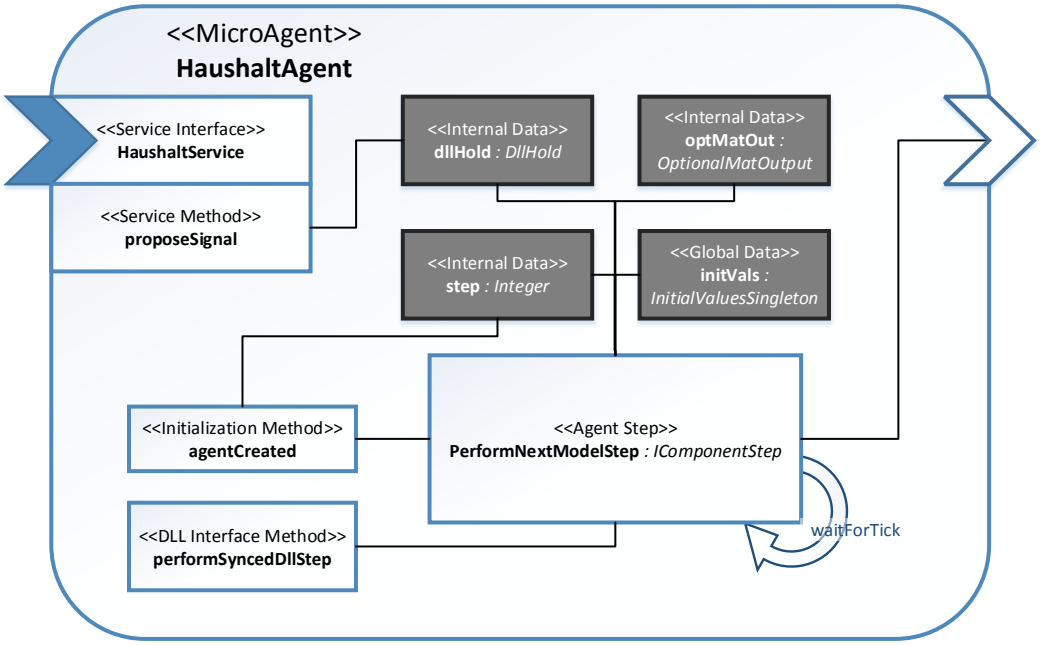


Abbildung 5.9: Design des Haushalt-Agenten

5.3 Startphase

Nachdem die Architektur des Frameworks und das Design der einzelnen Agententypen vorgestellt worden ist, soll jetzt die Beschreibung der Startphase des Systems den Abschluss dieses Kapitels bilden. Diese erfolgt anhand der Abbildung 5.10, einer Weiterentwicklung der in Kapitel 4.3.1 gezeigten Grafik und spiegelt so auch den evolutionär prototypischen Entwicklungsansatz wieder.

Beim Start des Simulationsframeworks wird der Starter-Agent erstellt. Dabei instanziert er die *StarterGUI*. Diese erwartet als Input vom Benutzer die Auswahl einer Konfigurationsdatei (i)-a. Nach der Auswahl wird der Dateiname an den Starter-Agenten übergeben (i)-b. Dieser liest zuerst die entsprechende Konfigurationsdatei mittels JAXB-Unmarshalling ein (ii)-a, liest dann per JMatIO-Library eine .MAT-Datei mit Umgebungsdaten ein (ii)-b und instanziert dann die einzige Instanz der Singleton-Klasse *InitialValuesSingleton* und befüllt diese mit Daten aus der XML-Repräsentation der Konfigurationsdatei und den darin angegebenen Daten aus der .MAT-Datei (iii). Im nächsten Schritt wird die aus der Konfigurationsdatei ausgelesene Master-DLL ausgewählt (iv)-a und in der benötigten Anzahl (abhängig von verfügbaren Prozessorkernen

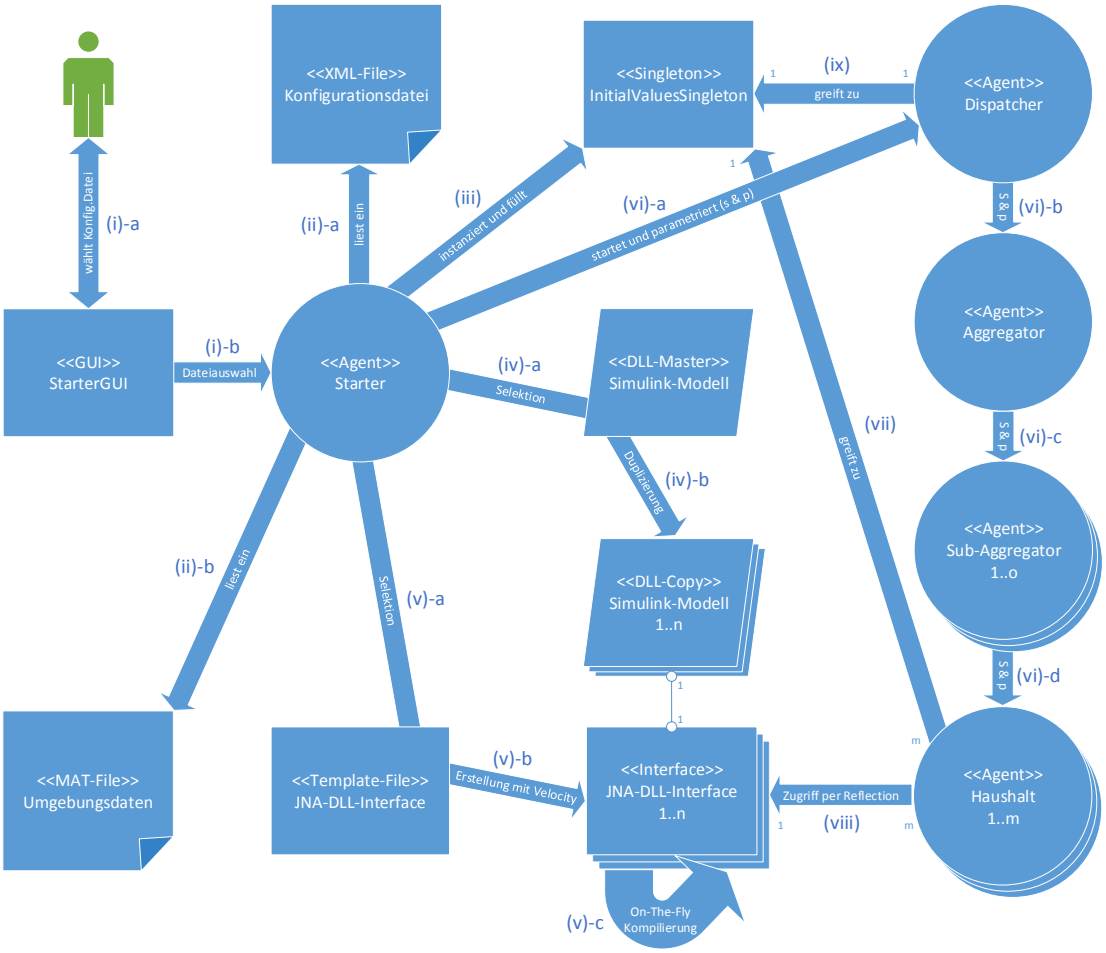


Abbildung 5.10: Startphase des Simulationssystems

und DLL-Typen im System) dupliziert (iv)-b. Daraufhin wird die Template-Datei für das JNA-DLL-Interface selektiert (v)-a und mithilfe der Velocity-Engine werden im Weiteren für die duplizierten DLLs die Interface-Klassen erstellt (v)-b. Diese werden per externem Java-Compiler-Aufruf zur Laufzeit kompiliert und stehen damit sofort zur Verfügung (v)-c. In den Schritten (vi)-a - (vi)-d findet der hierarchische Start der Agenten gemäß Konfigurationsdatei statt. Der Agenten erzeugen über Jadex-Plattform-Services die in der Hierarchie unter ihnen stehenden Agenten und geben ihnen simulationsspezifische Parameter mit. Außerdem wird der eigene Service-Interface-Identifier mit übergeben, damit der richtige Agent über den Directory Service von Jadex gefunden werden kann. Sobald die Haushalt-Agenten erzeugt worden sind, können sie für ihre Umgebungsdaten auf die *InitialValuesSingleton*-Instanz zugreifen (vii), genauso wie per Reflection auf die vorher kompilierten JNA-DLL-Interfaces (viii). Schritt (ix) zeigt, dass auch der Dispatcher-Agent Zugriff auf die Singleton-Klasse hat.

Die Registrierungsphase ist in dieser Grafik aus Gründen der Übersichtlichkeit nicht dargestellt.

6 Implementation

In diesem Kapitel werden aus Gründen des begrenzten Umfangs dieser Arbeit drei besonders signifikante Auszüge aus der Implementation des Simulationssystems ausgewählt und dann detailliert vorgestellt. Dazu wird der Quelltext von Agenten und Methoden, die zu den selektierten Beispielen gehören, aufgetragen und ausführlich besprochen.

Zur Entwicklung des Systems wurde Jadex AC 2.0 RC9 genutzt. Um Probleme zu vermeiden, wurde an dieser Version festgehalten, da sich die Jadex Plattform in permanenter Entwicklung befindet und sich wichtige Interfaces und Implementationsmethodiken ständig ändern können. Dabei wurde an dem im vorherigen Kapitel vorgestellten Design weitgehend festgehalten. Eine technische Änderung war nur bezüglich der Nutzung des *IComponentStep*-Interfaces nötig. Dieses konnte nicht in den Agententypen *SubAggregator*-Agent, *Aggregator*-Agent und *Dispatcher*-Agent genutzt werden, da die Simulationskomponente der Jadex-Plattform in der genutzten Version die Aktivität der Agenten anders als vorgesehen wahrgenommen hat. Der Simulation-Service hat schon neue Simulations-Ticks generiert, obwohl noch nicht alle Agenten mit ihrer Tätigkeit fertig waren, wodurch der zeitliche Ablauf der Simulation gestört wurde. Alternativ wurden die Operationen direkt in den Service des Agenten verlagert.

Der erste im Folgenden vorgestellte Auszug behandelt die XML-Konfigurationsdateien und die StarterGUI, der zweite die Vervielfältigung der DLLs und ihrer JNA-Interfaces zur Laufzeit und der dritte Auszug zeigt anhand eines Beispiels die Nutzung von asynchronen Service-Methoden-Aufrufen mit Jadex.

Zur Implementation im Allgemeinen ist zu erwähnen, dass als Vorgehensweise ein evolutionärer Prototyp gewählt wurde. Mit der Entwicklung wurde während der technische Analyse angefangen und das System wurde iterativ weiterentwickelt.

6.1 XML-Konfigurationsdateien und StarterGUI

Die während der Anforderungsanalyse aufgekommene Notwendigkeit der Konfigurierbarkeit des Simulationssystems mithilfe von XML-Konfigurationsdateien wurde in der technischen Analyse beleuchtet und als Lösung der Einsatz des JAXB Binding-Frameworks vorgeschlagen. Die in Form eines evolutionären Prototypen entstandene, simple *domänenspezifische Sprache* (DSL) in Form einer XML-Konfigurationsdatei, sowie deren JAXB Java-Repräsentation soll im Folgenden vorgestellt werden.

Abbildung 6.1 zeigt beispielhaft eine XML-Konfigurationsdatei für das Simulationssystem. Wurzelknoten ist *SimulationConfig*, von diesem gehen die Knoten *SimulationParameter*, *DataSources*, *DataSinks* und *Dispatchers* ab. Der Knoten *SimulationParameter* existiert genau einmal und beinhaltet für die Simulation notwendige globale Parameter:

- start-week: Die Woche im Jahr, in der die Simulation starten soll.
- steps: Die Anzahl der zu simulierenden Minuten.
- runs: Häufigkeit der Simulationsdurchläufe.
- ww-storage: Nutzung von Warmwasserspeichern in allen Haushaltsmodellen und der Algorithmik des Dispatchers.
- control-algorithm: Welche Steuerungs-Algorithmik in allen Agenten verwendet wird: keine, zentrale Einregelung, Lastverlagerung.
- reaction-potential-test: Testlauf für Reaktionspotenziale.
- cordtest: Testlauf für Bandlast.

Die Knoten *DataSources* und *DataSinks* sind Platzhalter und für eine Weiterentwicklung mit Nutzung von Datenbanken vorgesehen.

Der *Dispatchers*-Knoten kann wiederum mehrere *Dispatcher*-Knoten beinhalten, die die Dispatcher-Agenten im System abbilden, durch Attribute parametriert werden und selbst einen *Aggregators*-Knoten enthalten. Für den *Aggregators*-Knoten gilt selbiges, nur dass er einen *SubAggregators*-Knoten enthält und bei diesem gelten die gleichen Regeln sukzessive bis zum *CustomerType*-Knoten, der nur noch Attribute, aber keine weiteren Knoten mehr hält. Somit bildet die XML-Konfigurationsdatei den hierarchischen Aufbau des Systems ab und kann so als semi-strukturiert klassifiziert werden, da sie dadurch auch für einen Anwender gewissermaßen nachvollziehbar bleibt.

Node	Content
xml	version="1.0" encoding="UTF-8" standalone="yes"
SimulationConfig	
--	Eligible parameters for control-algorithm are: "none", "regulation", "load-balance" (, "both")
SimulationParameter	
start-week	0
steps	1440.0
runs	1
ww-storage	false
control-algorithm	none
reaction-potential-test	false
cordtest	false
DataSources	
DataSource	
DataSource	
DataSinks	
DataSink	
name	someDB
filename	dbconfig.xml
Dispatchers	
Dispatcher	
name	Dispatcher%N
power-band	10000
reaction-percentage	50
Aggregators	
Aggregator	
name	Aggregator%N_%O
SubAggregators	
--	type can be "WP" for heatpipe regulation and "NSP" for nightstorageheater regulation
--	display "true" gives .mat-Files for all customers of he SubAg, otherwise set "false"
--	setting regulator-type and/or number in SubAggregator overrides setting in CustomerTypes
SubAggregator	
name	SubAgg%N_%O_%P
type	WP
environment-type	EnvironmentType1
display	true
regulator-type	3.0
number	4
CustomerTypes	
CustomerType	
number	4
regulator-type	3.0
name	Cus_%N_%O_%P_%Q_%R
house-type	1.0
dll-name	export_22_FBH_win64
CustomerType	
CustomerType	

Abbildung 6.1: Beispiel einer Simulationskonfigurationsdatei

Nachfolgend werden die Attribute der Agenten repräsentierenden Knoten besprochen. Die Attribute des *Dispatcher*-Knotens:

- name: Name mit Platzhalter, der bei Systemstart per regulärem Ausdruck mit einem Identifikator ersetzt wird.

- power-band: Für den Dispatcher eingestelltes zu erzielendes Lastband.
- reaction-percentage: Prozent des Reaktionspotenzials, der abgerufen werden soll.

Der *Aggregator*-Knoten besitzt nur das name-Attribut, das äquivalent zum name-Attribut des *Dispatcher*-Knotens funktioniert.

Die Attribute des *Sub-Aggregator*-Knotens:

- name: Name mit Platzhalter, der bei Systemstart per regulärem Ausdruck mit einem Identifikator ersetzt wird.
- type: Typ der Heizung der zugeordneten Haushalte (die Algorithmik des Sub-Aggregator-Agenten unterstützt zur Zeit nur homogene Heizungstypen.
- environment-type: Die Umgebung für die Haushalt-Agenten des SubAggregator-Agenten (Temperatur, Sonneneinstrahlung etc.).
- display: Zum Debuggen werden spezifische Informationen eines jeden Haushalt-Agenten aufgezeichnet und in einem .MAT-File gespeichert.
- regulator-type (optional): Überschreibt die Reglereinstellung für alle Haushalt-Agenten des SubAggregator-Agenten.
- number (optional): Überschreibt die Anzahl für alle Haushalt-Agenten des SubAggregator-Agenten.

Mit den optionalen Attributen ist es möglich, die Größe der Konfigurationsdatei zu minimieren. Allerdings ist dann keine feingranulare Konfiguration möglich.

Die Attribute des *CustomerType*-Knotens:

- name: Name mit Platzhalter, der bei Systemstart per regulärem Ausdruck mit einem Identifikator ersetzt wird.
- number: Anzahl der Haushalt-Agenten dieses Typs.
- regulator-type: Reglereinstellung für die Haushalt-Agenten dieses Typs.
- house-typ: Genutzter Haustyp (Bauweise, z.B. leicht und gedämmt) für die Haushalt-Agenten dieses Typs (Wertebereich 1-13).
- dll-type: Name der genutzten DLL für die Haushalt-Agenten dieses Typs.

```java
package de.haw.c4das.beladealgorithmen.binding;

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement( name = "SimulationConfig" )
@XmlType( propOrder = { "simulationParameter", "dataSources", "dataSinks", "
    dispatchers" } )
public class SimulationConfig {
    private SimulationParameter simulationParameter;
    private List<DataSource> dataSources = new ArrayList<DataSource>();
    private List<DataSink> dataSinks = new ArrayList<DataSink>();
    private List<Dispatcher> dispatchers = new ArrayList<Dispatcher>();

    @XmlElement(name = "SimulationParameter")
    public SimulationParameter getSimulationParameter() {
        return simulationParameter;
    }
    public void setSimulationParameter(SimulationParameter
        simulationParameter) {
        this.simulationParameter = simulationParameter;
    }

    @XmlElementWrapper(name = "DataSources")
    @XmlElement(name = "DataSource")
    public List<DataSource> getDataSources() {
        return dataSources;
    }
    public void setDataSources(List<DataSource> dataSources) {
        this.dataSources = dataSources;
    }

    @XmlElementWrapper(name = "DataSinks")
    @XmlElement(name = "DataSink")
    public List<DataSink> getDataSinks() {
        return dataSinks;
    }
    public void setDataSinks(List<DataSink> dataSinks) {
        this.dataSinks = dataSinks;
    }

    @XmlElementWrapper(name = "Dispatchers")
    @XmlElement(name = "Dispatcher")
    public List<Dispatcher> getDispatchers() {
        return dispatchers;
    }
    public void setDispatchers(List<Dispatcher> dispatchers) {
        this.dispatchers = dispatchers;
    }
}
```

Quelltext 6.1: SimulationConfig.java, Wurzelklasse des XML-Bindings

Der Quelltext 6.1 zeigt die Wurzelklasse *SimulationConfig.java* des XML-Bindings. Interessant sind hier die Annotationen an der Klasse, sowie an den Getter- und Setter-Methoden der Klassenvariablen. Die Variablen können wie in diesem Fall auch selbsterstellte Klassen sein, diese müssen dann allerdings auch wieder über Annotationen für JAXB zugänglich sein. An der Klassendefinition findet sich die *XmlRootElement*-Annotation für das Wurzelelement. Die *@XmlElement*-Annotation stellt die Zuordnung von XML-Knoten zu Klassenvariable her. Enthält ein Knoten eine variable Anzahl von Sub-Knoten gleichen Typs, kann in Java eine Liste in Verbindung mit der *XmlElementWrapper*-Annotation zur Zuordnung verwendet werden.

Der Aufruf für das *Unmarshalling* einer XML-Konfigurationsdatei in die JAXB-annotierte Java-Klasse ist einzeilig wie folgt:

```
SimulationConfig simConfig = JAXB.unmarshal(confFile, SimulationConfig.class
    );
```

6.1.1 StarterGUI

Die Abbildung 6.2 zeigt die graphische Startoberfläche, mit der beim Systemstart eine Konfigurationsdatei ausgewählt wird. Das Verzeichnis *config* ist voreingestellt, sowie die Dateiendungsfilter *.xml bzw. *.XML. Mit dem *Starte Simulation*-Button wird aus der GUI heraus über ein Handle auf den Starter-Agenten zugegriffen, diesem der Dateiname übergeben und mittels der Plattform-Methode *waitForTick* der *ReadConfigFileAndInitializeSystemStep* ausgeführt.

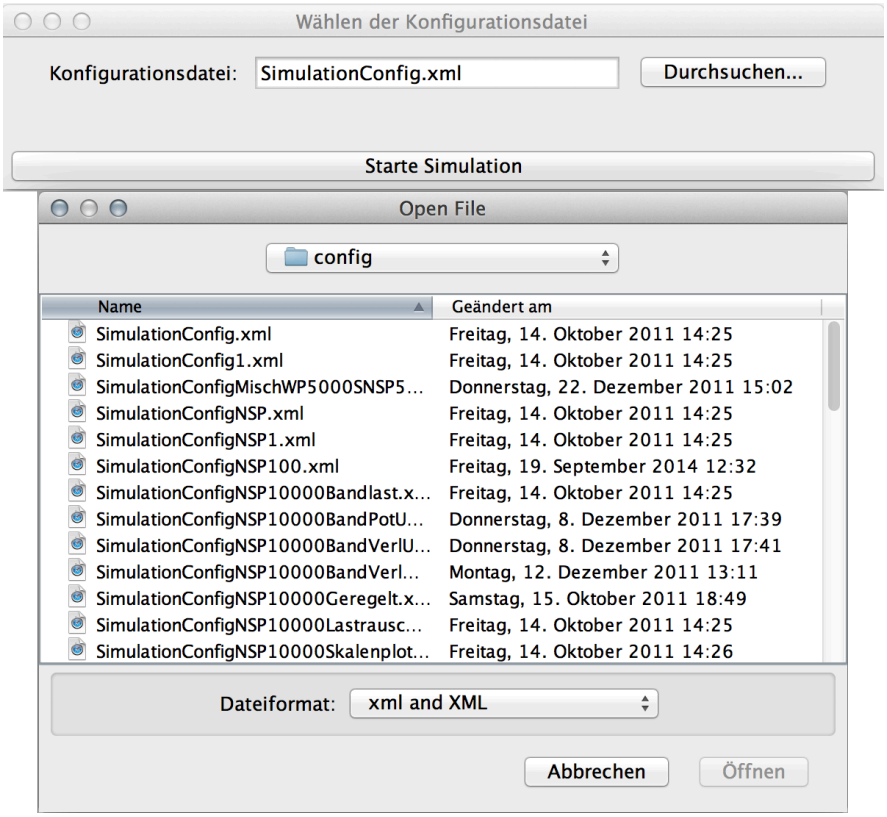


Abbildung 6.2: Die StarterGUI mit offenem Auswahlfenster

6.2 Automatisierte Duplizierung der DLLs und der zugehörigen JNA-Interfaces zur Lastverteilung

Das zweite Beispiel ist die adaptive Duplizierung der DLLs und deren Interface-Klassen, die stattfinden muss, um die Last auf mehrere Prozessorkerne verteilen zu können ohne dabei den Workflow zu beeinträchtigen und Mehraufwände bei Änderungen an der DLL zu erzeugen. Dieser Teil der Software wurde ausgewählt, da es ein zentraler Aspekt der Implementation des Simulationssystems ist.

Rückblickend auf das Schema der Startphase (Abbildung 5.10 auf Seite 55) werden hier die Schritte (iv)-a bis (v)-c, sowie Schritt (viii) betrachtet.

Der Quelltext 6.2 ist ein Auszug aus dem Starter-Agenten und stößt die Schritte (iv)-a bis (v)-c an. Zuerst wird dazu die Anzahl der verfügbaren virtuellen Kerne ausgelesen. Analog

zum Kapitel 4.3 wird aus der Zahl der Kerne und der Anzahl der aus der Konfigurationsdatei ausgelesenen DLL-Typen die Menge der benötigten Duplikate pro DLL-Typ bestimmt. Diese Zahl wird in dem Singleton-Objekt gespeichert und daraufhin wird für jeden DLL-Typ eine Schleife dieser Menge durchlaufen. Darin werden zwei Methoden der Hilfsklasse *DllDuplication* aufgerufen. Die Methode *createDuplicatedDLL(String libName, String targetLibName)* erzeugt ein Duplikat der per String angegebenen DLL, die Methode *createNativeholder(String dllName, int number)* erstellt aus dem DLL-Interface-Template eine DLL-Interface-Klasse. Nach Durchlauf der Schleifen wird die Methode *compileNativeHolders()* ausgeführt, die ebenfalls zur *DllDuplication*-Hilfsklasse gehört.

```
int cores = Runtime.getRuntime().availableProcessors();

Integer multiCoreDllCount;
if (cores % dllTypes.size() != 0)
    multiCoreDllCount = (cores / dllTypes.size()) + 1;
else
    multiCoreDllCount = cores / dllTypes.size();

init.setMultiCoreDllCount(multiCoreDllCount);

for (String dllType : dllTypes) {
    for (int i = 0; i < multiCoreDllCount; i++) {
        try {
            dllDupl.createDuplicatedDLL(dllType, dllType + "_" + i);
        } catch (IOException e) {
            getLogger().severe(e.getMessage());
        }
        try {
            dllDupl.createNativeholder(dllType, i);
        } catch (Exception e) {
            getLogger().severe(e.getMessage());
        }
    }
}

try {
    dllDupl.compileNativeHolders();
} catch (Exception e) {
    getLogger().severe(e.getMessage());
}
```

Quelltext 6.2: Auszug aus dem Starter-Agenten

Die *createDuplicatedDLL*-Methode und ihre Sub-Methoden werden in Quelltext 6.3 gezeigt. Aus den Eingabeparametern werden neue File-Objekte erzeugt und wenn die Zieldatei nicht schon existiert, wird die Methode *copyFile(File src, File dest)* aufgerufen. Diese erzeugt auf

den File-Objekten neue FileInput- respektive FileOutputStreams. Beide Streams werden in die *copy(InputStream in, OutputStream out)* gegeben, welche über einen Byte-Array-Buffer aus dem InputStream liest und in den OutputStream schreibt. Das Exception-Handling wird über *throws* nach außen in den Agenten verlagert, wo die Exceptions per Jadex-eigenem Logging-Framework aufgezeichnet werden.

```
public void createDuplicatedDLL(String libName, String targetLibName)
            throws Exception {
    File sourceLib = new File("dllModels/" + libName + ".dll");
    File targetLib = new File("dllModels/" + targetLibName + ".dll");

    if (!targetLib.exists()) {
        copyFile(sourceLib, targetLib);
    }
    return;
}

private void copyFile(File src, File dest) throws Exception {
    FileInputStream fis = null;
    FileOutputStream fos = null;

    try {
        fis = new FileInputStream(src);
        fos = new FileOutputStream(dest);

        copy(fis, fos);
    } catch (IOException e) {
        throw new IOException(e);
    } finally {
        if (fis != null) {
            fis.close();
        }
        if (fos != null) {
            fos.close();
        }
    }
}

private void copy(InputStream in, OutputStream out)
        throws IOException {
    byte[] buffer = new byte[0xFFFF];
    for (int len; (len = in.read(buffer)) != -1;) {
        out.write(buffer, 0, len);
    }
}
```

Quelltext 6.3: Methode *createDuplicatedDLL(String libName, String targetLibName)* und Sub-Methoden

```
package de.haw.c4das.beladealgorithmen.nativeholder;

import java.util.Random;
import com.sun.jna.Native;

public class $className {

    protected static Random rand = new Random();

    static
    {
        String userDir = System.getProperty("user.dir");
        System.setProperty("jna.library.path", userDir + "/dllModels");
        Native.register("$dllName");
        Kunde_initialize('1');
    }

    public synchronized static native void Kunde_initialize(Character
        firstTime);
    private synchronized static native void Kunde_custom ( double[]
        arg_Modell_Auswahl, ... double[] arg_Out_Hold_d_600 );

    public synchronized static void Kunde_step ( double[] arg_Modell_Auswahl
        , ... double[] arg_Out_Hold_d_600 ) {
        Kunde_initialize('0');
        arg_In_Hold_d_600[arg_In_Hold_d_600.length - 1] = rand.nextDouble();
        Kunde_custom(arg_Modell_Auswahl, ... arg_Out_Hold_d_600);
    }
}
```

Quelltext 6.4: Template für die Interface-KLassen

Die Quelltexte 6.4 und 6.5 beinhalten das Template für die DLL-Interface-Klasse und die Methode *createNativeholder*. Das Template folgt der JNA-Referenzimplementation, die in Kapitel 4.1.1 gezeigt wurde. Allerdings erweitert es diese in zwei Punkten: (i) Es wird ein Random-Objekt erzeugt. (ii) Die *_custom*-Methode der DLL wird nicht direkt veröffentlicht, sondern in einer weiteren Funktion gekapselt, die vor erneutem Aufruf die DLL neu initialisiert und ihr eine Zufallszahl aus dem Random-Objekt beim Aufruf übergibt. Diese Zufallszahl wird in der DLL als Seed für einen Zufallszahlengenerator genutzt. Dies hat sich bei den Systemtests als notwendig herausgestellt. Es muss sichergestellt werden, dass jeder DLL-Aufruf bei einer massiven Simulation unterschiedliche Zufallszahlen innehat. Der Standard-Seed des Zufallszahlengenerators in der Simulink-DLL ist, wie auch häufig in anderen Systemen, die Systemzeit in Millisekunden. Durch die bei heutigen Prozessortakten sehr schnellen Schaltzyklen kann somit der Effekt auftreten, dass viele DLL-Aufrufe mit gleichem Systemzeit-Seed stattfinden

und damit gleiche Reihen von Zufallszahlen erzeugt werden. Dadurch werden die Ergebnisse eines Simulationslaufes verfälscht, da keine Durchmischungseffekte auftreten. Die Signaturen der *_step-* und *_custom-*Methode sind gekürzt dargestellt. Auffällig an dem Template sind weiterhin die zwei Platzhalter *$className* und *$dllName* im Velocity-Stil.

```
public void createNativeholder(String dllName, int number) throws Exception
    {
    // Initialisiere Velocity
    Properties p = new Properties();
    p.setProperty("resource.loader", "file, class");
    p.setProperty("class.resource.loader.description",
            "Velocity Classpath Resource Loader");
    p.setProperty("class.resource.loader.class",
            "org.apache.velocity.runtime.resource.loader.
                ClasspathResourceLoader");
    p.setProperty("file.resource.loader.path",
            "src/de/haw/c4das/beladealgorithmen/templates");
    Velocity.init(p);

    // Datei zum Beschreiben wird erzeugt.
    String fileName = dllName.replaceAll("_", "") + number
            + EXPORT_FILE_EXTENSION;
    File outputFile = new File(EXPORT_FILE_PATH + fileName);
    logger.log(Level.FINER,
            "Exportiere nach: " + outputFile.getAbsolutePath());

    FileWriter fileWriter;
    VelocityContext context = new VelocityContext();

    // Einfuegen der Daten fuer das Template:
    context.put("className", dllName.replaceAll("_", "") + number);
    context.put("dllName", dllName + "_" + number);

    fileWriter = new FileWriter(outputFile);
    if (Velocity.mergeTemplate("nativeholder" + TEMPLATE_FILE_EXTENSION,
            "UTF-8", context, fileWriter)) {
        logger.finer("Nativeholder erfolgreich aus Template erzeugt.");
        fileWriter.close();
        return;
    } else {
        logger.severe("Schreiben durch Velocity nicht moeglich.");
        fileWriter.close();
        return;
    }
}
```

Quelltext 6.5: Methode *createNativeholder(String dllName, int number)*

Quelltext 6.5 stellt den Velocity-Engine Vorgang dar. Dabei werden zuerst Eigenschaften für die Engine gesetzt, unter anderem der Template-Pfad, und dann wird die Enginge initialisiert. Dann wird die zu erzeugende *.java*-Datei erstellt und ein *VelocityContext*-Objekt wird erzeugt. Dieses erhält die Daten, mit denen die Platzhalter im Template ersetzt werden sollen. Daraufhin wird ein *FileWriter* auf der erstellten Datei erzeugt und mit in die *mergeTemplate*-Methode von Velocity gegeben. Diese ersetzt die Platzhalter mit den tatsächlichen Inhalten und schreibt das befüllte Template in den FileWriter.

Wenn alle DLLs dupliziert und alle Klassen aus dem Template erzeugt worden sind, müssen diese Klassen noch zur Laufzeit kompiliert werden, damit sie direkt und nicht erst beim nächsten Start des Systems nutzbar sind. Dazu wird die Methode *compileNativeHolders* in Quelltext 6.6 genutzt. Dort wird in einem String der Aufruf des Java-Compilers zusammengebaut, der daraufhin in einem Sub-Prozess ausgeführt wird. Dazu muss der Ort der JNA-Bibliothek, die im Template genutzt wird, dem Java-Compiler als *classpath*-Variable übergeben. Dementsprechend muss sich die JNA-Library innerhalb der Arbeitsumgebung befinden. Die Ausgaben des Sub-Prozesses müssen mit einem InputStreamReader ausgelesen werden, ansonsten wird dieser nicht ordnungsgemäß ausgeführt.

```
public void compileNativeHolders() throws IOException {
    // Definieren des auszufuehrenden Commands
    String userDir = System.getProperty("user.dir");
    String compileCommand = "javac -sourcepath src -d bin -classpath \".;" +
         userDir + "\\jnalib\\*\" " + EXPORT_FILE_PATH + "*.java";

    // Starten des Sub-Prozesses und Auslesen des Ausgabe-Streams
    String line;
    Process p = Runtime.getRuntime().exec(compileCommand);
    BufferedReader input = new BufferedReader(new InputStreamReader(p.
        getInputStream()));
    while ((line = input.readLine()) != null) {
        System.out.println(line);
    }
    input.close();
}
```

Quelltext 6.6: Methode *compileNativeholders()*

Zuletzt zeigt Quelltext 6.7 einen Auszug aus dem Haushalt-Agenten, der den Aufruf der zuvor erstellten und kompilierten DLL-Interface-Klassen verdeutlicht. Dafür werden zunächst die Methoden der dem Haushalt-Agenten zugewiesenen DLL-Interface-Klasse über einen String mit dem *fully-qualified-classname* gesucht. daraufhin werden aus diesen Methoden die benötigten

ebenfalls per String herausgesucht und jeweils einer Instanz-Variable zugewiesen. Dieser Schritt geschieht beim ersten Aufruf des Haushalt-Agenten einmalig und wird hier nur der Übersichtlichkeit halber sequentiell mit dem danach folgenden Methodenaufruf über die *invoke*-Methode dargestellt. Bei dieser sind wie auch schon oben im Template die Übergabeparameter gekürzt aufgeführt.

```
Method[] methods = null;

try {
    methods = Class.forName("de.haw.c4das.beladealgorithmen.nativeholder." +
         customerType.getDllName().replaceAll("_", "") + multiCoreSetup).
        getMethods();
} catch (Exception e) {
    getLogger().severe(e.getMessage());
}

for (Method m : methods) {
    if (m.getName().equals("Kunde_initialize"))
        initialize = m;
    else if (m.getName().equals("Kunde_step"))
        stepSync = m;
}

try {
    stepSync.invoke(null, new Object[]{ modelSelect_d, ... buf_hold_d_600 })
        ;
} catch (Exception e) {
    getLogger().severe(e.getMessage());
}
```

Quelltext 6.7: Auszug aus dem Haushalt-Agenten

6.3 Asynchroner Service-Aufruf in Jadex am Beispiel „*Starten der Haushalt-Agenten*“

An dieser Stelle soll das Programmiermodell der asynchronen Service-Methoden-Aufrufe von Jadex erklärt werden. Als Beispiel ist ein Auszug aus dem SubAggregator-Agenten gewählt. Die Haushalt-Agenten sollen erst gestartet werden, wenn die Service-Interfaces aller SubAggregator-Agenten initialisiert und beim Directory-Service registriert sind. Damit wird sichergestellt, dass das System synchronisiert starten kann.

```
SServiceProvider.getServices(getServiceProvider(), ISubAggregatorService.
    class).addResultListener(new IResultListener() {

    public void resultAvailable(Object result) {
        Collection<ISubAggregatorService> isaServices = (Collection<
            ISubAggregatorService>)result;
        ISubAggregatorService[] isaServicesArray = isaServices.toArray(new
            ISubAggregatorService[isaServices.size()]);

        if (isaServicesArray == null || isaServicesArray.length !=
            subAggregatorCount) {
            SServiceProvider.getServices(getServiceProvider(),
                ISubAggregatorService.class).addResultListener(this);
        }
        else {
            // Starten der Haushalt-Agenten, gekürzt..
        }
    }

    public void exceptionOccurred(Exception exception) {
     getLogger().severe(exception.getMessage());
     SServiceProvider.getServices(getServiceProvider(),
         ISubAggregatorService.class).addResultListener(this);
    }
});
```

Quelltext 6.8: Beispiel für einen asynchronen Service-Aufruf (Auszug aus SubAggregatorAgent.java)

Der Quelltext 6.8 zeigt die Implementation für dieses Problem. Zuerst wird dabei die Service-Methode *getServices* des Plattform-Service-Interfaces aufgerufen. An diesen Aufruf wird ein *ResultListener* gebunden, dessen Implementation an Ort und Stelle vorgenommen wird. Das ResultListener-Interface bietet die zwei Methoden *resultAvailable* und *exceptionOccured*, deren Implementation vorgenommen werden muss. Die *resultAvailable*-Methode übergibt ein Suchergebnis vom Typ *Object*, das auf die gewünschte Klasse, in diesem Fall eine *Collection* von Service-Interfaces, gecastet werden muss. Dann wird geprüft, ob diese Collection weder **null** noch von der Anzahl kleiner als die Gesamtzahl der SubAggregator-Agenten ist. Sollte dies nicht der Fall sein, werden die Haushalt-Agenten gestartet, ansonsten wird der gleiche Service-Methoden-Aufruf nochmals ausgeführt. Als ResultListener wird dann derselbe wieder mittels **this** gebunden. In dem Fall, dass die *exceptionOccured*-Methode aufgerufen wird, weil z.B. noch kein einziger SubAggregator-Service beim Directory-Service registriert ist, wird der gleiche Service-Methoden-Aufruf wiederholt ausgeführt. Dies konnte bei sehr schnellen Computern mit hohen Prozessor-Taktraten beobachtet werden.

7 Test & Auswertung

Nachdem in den vorherigen Kapiteln das Design und die Implementation des Simulations-Frameworks vorgestellt worden sind, soll nun betrachtet werden, inwiefern die in Kapitel 3 aufgestellten Anforderungen erfüllt werden.

Dazu wird zuerst auf die Tests, denen das Framework sowohl während der Entwicklung, als auch nach deren Fertigstellung unterzogen wurde, eingegangen. In einem zweiten Schritt wird dann eine Auswertung mit den Schwerpunkten „Performanz & Skalierbarkeit“ sowie „domänenspezifische Ergebnisse“ vorgenommen.

7.1 Test

Die Entwicklung eines evolutionären Prototypen bedingt, dass permanent Testfälle erstellt und auch wieder angepasst werden müssen. Dieser Prozess kann innerhalb dieses Kapitels nicht adäquat dargestellt werden. Deswegen werden die Testfälle in der Form besprochen, in der sie nach Beendigung der Arbeit am Simulations-Framework vorliegen.

In den folgenden Abschnitten werden die Testfälle kategorisiert, erläutert und ausgewertet, selbst zu finden sind sie im Anhang A.1.

7.1.1 Komponenten- und Integrationstests

Komponententests sind Tests, die den gesamten Entwicklungsprozess über benötigt werden. Nach Fertigstellung einer Komponente, muss deren Funktionalität gegen einen zuvor definierten Testfall geprüft werden. Diese Tests müssen bei jeglichen Änderungen an dieser Komponente wiederholt werden.

Testfall-ID	Testfall-Name	durchgeführt	bestanden
	Komponenten-Testfälle		
1	SIM-DLL-POS	☒	☒
2	SIM-DLL-NEG	☒	☐
3	SIM-SUBAGG-ALG1-POS	☒	☒
4	SIM-SUBAGG-ALG1-NEG	☒	☒
5	SIM-SUBAGG-ALG2-POS	☒	☒
6	SIM-SUBAGG-ALG2-NEG	☒	☒
7	SIM-AGG-POS	☒	☒
8	SIM-AGG-NEG	☒	☒
9	SIM-DISP-POS	☒	☒
10	SIM-DISP-NEG	☒	☒
	Integrations-Testfälle		
11	SIM-ALG-INTEGRATION	☐	☐

Tabelle 7.1: Komponenten- und Integrations-Testfälle

Tabelle 7.1 gibt eine Übersicht der Komponenten- und Integrations-Testfälle, die für das Simulations-Framework definiert wurden. Als zusätzliche Information wird gezeigt, ob ein definierter Testfall auch durchgeführt wurde und ob diese Durchführung erfolgreich war.

Die Testfälle 1 und 2 testen die DLL und deren JNA-Interface. Sie sind definiert in den Tabellen A.1 und A.2. Der negative Testfall konnte nie bestanden werden, da die DLL bei fehlerhaften oder unvollständigen Daten die Java Virtual Machine zum Absturz bringt. Dieser Umstand konnte nicht behoben werden und wäre für den Einsatz in einem operativen Softwaresystem sehr kritisch. Während der technischen Analyse konnte dieses Verhalten noch nicht beobachtet werden. Es trat erst bei den komplexen thermodynamischen Modellen auf. Nach Abwägung der Relevanz für das Simulations-Framework wurde entschieden, diesen Ansatz dennoch weiter zu verfolgen. Abgedeckt werden mit diesen Testfällen die Anforderungen 4 und 10, die Einbindung der Matlab-Simulink Modelle sowie das Einlesen von .MAT-Dateien.

Die Testfälle 3 bis 10, definiert in den Tabellen A.3 bis A.10, beschreiben die Tests, die durchgeführt wurden, um die Überführung der Matlab-Simulink Algorithmik der Akteure SubAggregator, Aggregator und Dispatcher in das Simulations-Framework auf Korrektheit zu prüfen. Alle Testfälle konnten durchgeführt und nach diversen Iterationen bestanden werden. Die Testfälle decken die Anforderung 5, die Überführung von Matlab-Simulink Algorithmen in das Simulations-Framework, ab.

Der Testfall 11, Tabelle A.11, ist ein Integrations-Testfall und prüft das Zusammenspiel der überführten Algorithmen. Die Durchführung dieses Testfalls wäre aufgrund der losen Kopplung der Entitäten durch Services und deren Implementation als Komponenten innerhalb des Simulations-Frameworks nur unter erheblichem Aufwand möglich, der über den Umfang dieser Arbeit hinausgegangen wäre, und wurde deshalb nicht durchgeführt. Rückblickend auf die technische Analyse und die Design-Phase wird hier deshalb der Einsatz von *Dependency Injection* (DI) bzw. eines *DI-Frameworks* wie z.B. Spring[1] propagiert. Diese aufkommende Technologie ermöglicht die Entkopplung von Abhängigkeiten innerhalb von Klassen und erleichtert so das Schreiben und das Einbinden von Unit- und Integrations-Tests. Der Einsatz von DI in diskreten Simulationen wird in [Coyne u. a. (2008)] beschrieben, weiterführende Informationen zu DI finden sich in [Fowler (2004)]. Die von Testfall 11 betroffenen Anwendungsfälle werden stattdessen durch die Systemtests im folgenden Abschnitt mit abgedeckt.

7.1.2 Systemtests

Die Systemtests sollen das Simulations-Framework mit allen Bestandteilen auf funktionale Korrektheit und Erfüllung der Anforderungen testen. In Tabelle 7.2 sind die Systemtests aufgeführt.

Testfall-ID	Testfall-Name	durchgeführt	bestanden
12	SIM-MATLAB	☒	☒
13	SIM-VALID-20K	☒	☒

Tabelle 7.2: System-Testfälle

Mit Testfall 12, definiert in Tabelle A.12, soll die korrekte Funktionsweise des Simulations-Frameworks anhand des in Matlab-Simulink bestehenden Testszenarios verifiziert werden. Dieser Testfall fängt damit auch den nicht durchgeführten Integrationstest für die Algorithmik aus Matlab-Simulink ab. Wichtige Vorbedingung des Testfalls ist die Umschaltung aller stochastischen Größen in beiden Systemen auf feste Werte, damit die Ergebnisse auf Gleichheit geprüft werden können. Die Anwendungsfälle 1, 2, 3, 4, 5, 8, 9 und 10 (nachzulesen in den Kapiteln 3.2 und 3.3) werden von diesem Testfall abgedeckt.

Der Testfall 13, Definition in Tabelle A.13, soll die Funktionalität des Simulations-Frameworks für eine größere Anzahl von Haushalt-Agenten validieren, als es in Matlab-Simulink möglich ist. In diesem Fall werden 20.000 Haushalt-Agenten festgelegt. Die Ergebnisse werden vom System

[1] https://spring.io/

als .MAT-Dateien gespeichert und einem Experten-Team übergeben. Dieses Team unterzieht die Ergebnisse mithilfe der Matlab-Simulink Umgebung einem Plausibilitäts-Check. Dieser Testfall hat zum Beispiel die in Kapitel 6.2 beschriebene Problematik der Zufallszahlengeneratoren in den Simulink-DLLs aufgedeckt. Zusätzlich zu den von Testfall 12 abgedeckten Anforderungen werden hier die Anforderungen 7 und 13 erfasst.

7.1.3 Performanztests

Die Testfälle für die Performanztests unterscheiden sich von den anderen Testfall-Typen dahingehend, dass sie nicht eindeutig z.B. anhand eines Daten-Vergleichs mit *bestanden* oder *nicht bestanden* bewertet werden können. Sie werden aufgelistet in Tabelle 7.3 und ihre Auswertung erfolgt in Abschnitt 7.2.1.

Testfall-ID	Testfall-Name	durchgeführt
14	SIM-PERF-VERGLEICH	☒
15	SIM-PERF-SKAL	☒

Tabelle 7.3: Performanz-Testfälle

Der mit Testfall 14 in Tabelle A.14 beschriebene Performanztest zielt auf einen Vergleich der Simulationsperformanz für verschiedene wohldefinierte Szenarien zwischen der Matlab-Simulink Plattform und dem mit Jadex AC entwickelten Simulations-Framework ab. Alle Testläufe müssen auf der selben Hardwareplattform stattfinden, um valide Ergebnisse zu erhalten. Mit diesem Testfall wird hauptsächlich die Abdeckung von Anforderung 14 erreicht.

Die generelle Skalierbarkeit sowie die daran gekoppelte Performanz des Simulations-Frameworks sind ein zentraler Aspekt dieser Arbeit. Beides soll mit Testfall 15, definiert in Tabelle A.15, getestet werden. Dafür werden Simulationsszenarien von bis zu 100.000 Haushalt-Agenten erstellt, durchlaufen und deren Zeit gemessen. Damit sollen die Anforderungen 13 und 14 erfasst werden.

7.2 Auswertung

Die Entwicklung des Simulations-Frameworks ist maßgeblich aus zwei zentralen Zielstellungen entsprungen. Zum einen sollte ein Weg aufgezeigt werden, mit dem eine skalierbare Integration von 4GL-Modellen, wie z.B. aus Matlab-Simulink, in ein Simulations-Framework

ermöglicht werden kann, die es erlaubt massive Simulationen durchzuführen. Zum anderen sollten mithilfe dieser massiven Simulationen innerhalb der Anwendungsdomäne, aus der die initiale Problemstellung stammt, relevante Ergebnisse erarbeitet werden können.

In den folgenden Abschnitten sollen nun die Ergebnisse dieser zwei Zielstellungen ausgewertet werden.

7.2.1 Skalierbarkeit & Performanz

In diesem Abschnitt soll ermittelt werden, ob die Ziele bezüglich der Skalierbarkeit und Performanz des Simulations-Frameworks erreicht werden konnten. Dazu werden zuerst die Ergebnisse des Testfalls 14 herangezogen, mit dem ein Vergleich der Performanz zwischen Simulationsläufen in Matlab-Simulink und dem Simulations-Framework vorgenommen wurde.

Performanzvergleich

Die im Testfall 14 beschriebenen Simulationsläufe für 1 / 7 / 14 / 30,5 und 149 Tage wurden jeweils zehnfach durchgeführt und Durchschnittswerte daraus ermittelt. Als Hardwareplattform wurde für beide Systeme ein Workstation-PC mit 4 Kernen a 4,5GHz mit Hyperthreading, 16 GB RAM, Windows 7 64bit und der Oracle 64bit JVM genutzt. Zusätzlich wurde die Anzahl der im Simulationslauf vorkommenden SubAggregator-Agenten zwischen 1, 4 und 8 variiert, um Auswirkungen der hierarchischen Struktur und der dadurch möglicherweise auftretenden Flaschenhälse zu messen. Allerdings sind 4 SubAggregator-Agenten der Referenzwert für den direkten Vergleichstest.

Die Ergebnisse des direkten Systemvergleichs werden in Abbildung 7.1 präsentiert. Hier wird deutlich, dass für beide Systeme bei einem wachsenden Simulationszeitraum die Ausführungsdauer linear ansteigt, also eine Wachstumsrate von $\mathcal{O}(n)$ bezüglich des Simulationszeitraums gilt. Nach einer linearen Trendberechnung gilt dabei für Matlab $y = 1540,2x + 430,78$ und für das Simulations-Framework $y = 20,061x + 42,134$. Das ergibt einen Faktor von etwa 76,8 um den das Simulations-Framework bei derselben Hardware und gleicher Konfiguration schneller ist als die Matlab-Simulink Umgebung. Die deutlich größere Verschiebungskonstante deutet außerdem auf eine langsamere Initialisierung des Systems in Matlab hin.

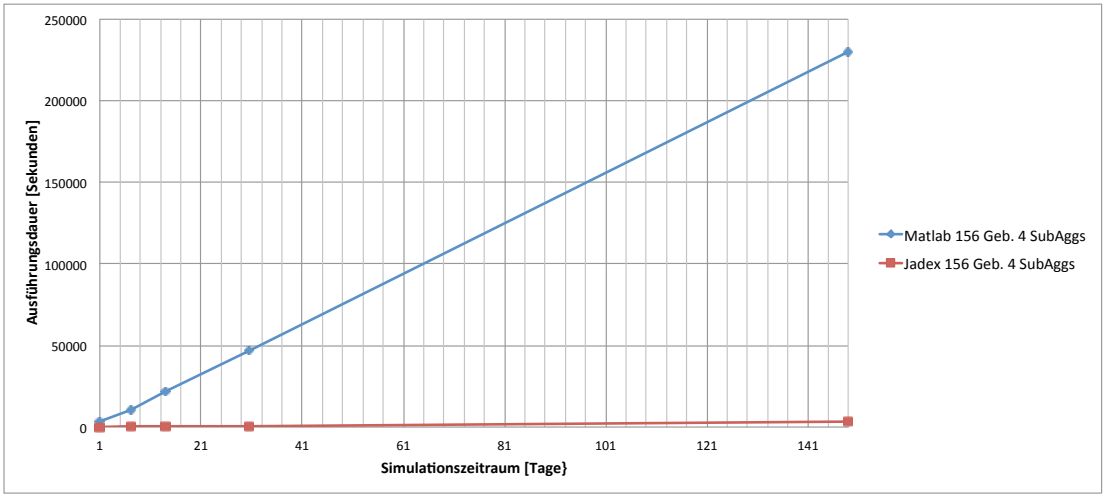


Abbildung 7.1: Performanz-Vergleich Matlab-Simulink vs. Simulations-Framework

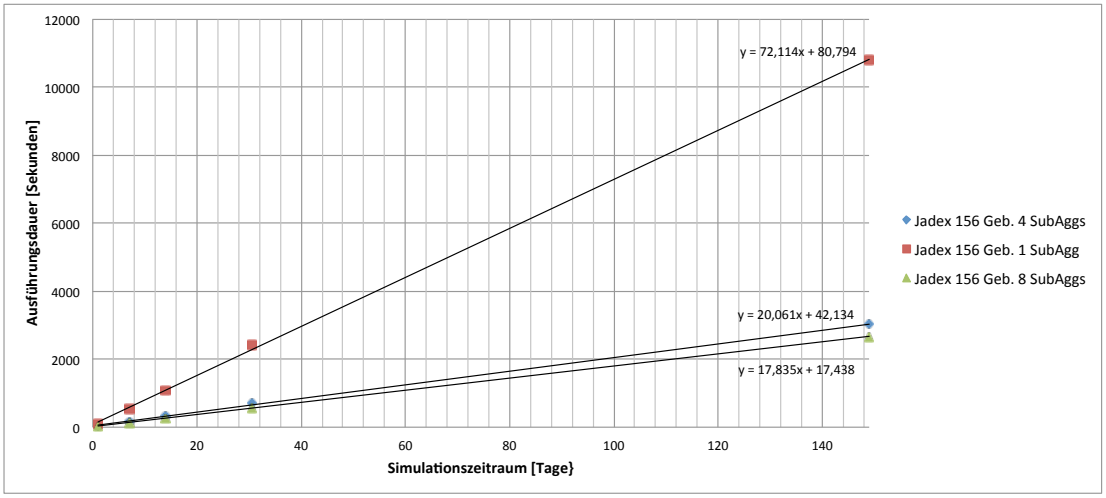


Abbildung 7.2: Performanz-Unterschiede mit unterschiedlicher SubAggregator-Anzahl

Abbildung 7.2 stellt die Ergebnisse für die variierte Anzahl von SubAggregatoren in den Testläufen des Simulations-Frameworks dar. Dabei wird klar, dass auch die Anzahl der Sub-Aggregatoren von ähnlicher Bedeutung für das Simulations-Framework ist, wie schon die Zahl der duplizierten DLLs. Auch hier sind für eine Hardware-Plattform mit vier Kernen und Hyperthreading acht Sub-Aggregator-Agenten geringfügig schneller als vier. Die Nutzung von nur einem SubAggregator-Agenten hingegen bremst das System dabei etwa um den Faktor 4 aus.

Skalierbarkeit des Simulations-Frameworks

Im Folgenden wird die Auswertung der mit Testfall 15 gewonnen Ergebnisse vorgenommen, um die Skalierbarkeit und die damit einhergehende Performanz des Systems zu prüfen. Die dafür durchgeführten Testläufe wurden auf einem Server mit 64 Kernen a 2,66GHZ ohne Hyperthreading, 256GB RAM, Windows Server 2008 R2 Enterprise und der Oracle 64bit JVM vorgenommen. Der Simulationslauf mit 100.000 Haushalt-Agenten konnte auf dem Workstation-PC nicht durchgeführt werden. Alle Testläufe wurden dabei zehnfach ausgeführt und es wurden Durchschnittswerte der Ergebnisse gebildet.

Abbildung 7.3 stellt die Ergebnisse für die gesamte Ausführungszeit dar.

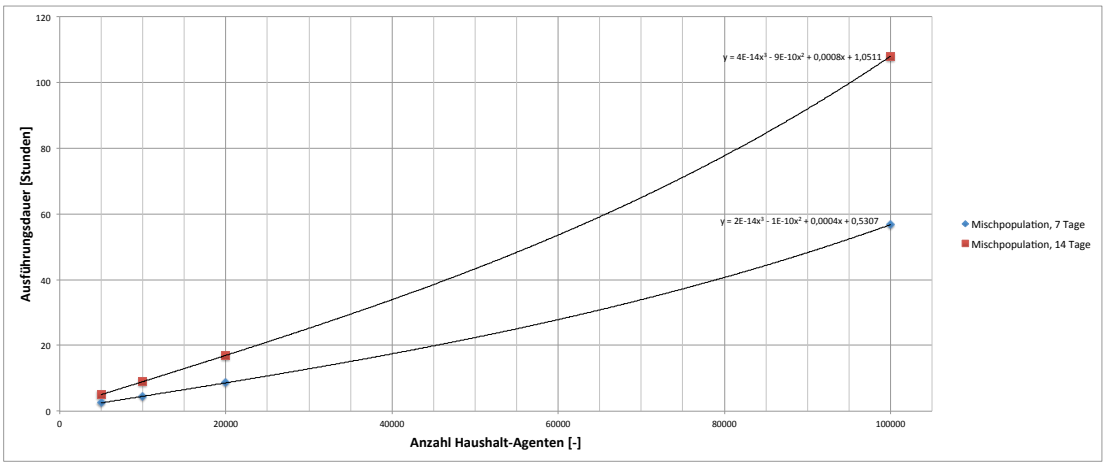


Abbildung 7.3: Simulationsdauer mit bis zu 100.000 Haushalt-Agenten für 7 und 14 simulierte Tage

Anhand der eingefügten polynomischen Trendlinien und deren Polynomfunktionen lässt sich erkennen, dass das Simulations-Framework nicht linear skaliert, sondern eine polynomiale Wachstumsrate von $\mathcal{O}(p(n))$ mit $p(n) = 4 * 10^{-14}n^3 - 9 * 10^{-10}n^2 + 0,0008n + 1,0511$, einem Polynom dritten Grades, für eine Simulationszeit von 14 Tagen bezüglich der Anzahl von Haushalt-Agenten im System aufweist. Dies lässt sich anhand verschiedener Faktoren erklären.

Zum einen spielt die in Abbildung 7.4 aufgetragene Initialisierungsdauer eine Rolle. Diese Dauer steigt laut polynomischer Trendberechnung mit einer polynomiellen Wachstumsrate von $\mathcal{O}(p(n))$ mit $p(n) = 5 * 10^{-10}n^2 + 3 * 10^{-6}n - 0,0019$, also einem Polynom zweiten Grades. Dies ist zurückzuführen auf den Directory Service von Jadex, der hier einen Flaschenhals

bildet. Jadex weist somit eine ähnliche Problematik wie JADE bezüglich des Directory Services auf, die in Kapitel 4.2.2 charakterisiert wird. Dieser Faktor wirkt sich allerdings nur auf den Systemstart aus, mittels einer Zwischenspeicherung der während der Initialisierung ermittelten Service-Interfaces wird der Directory Service während der Simulation nicht weiter benutzt. Allerdings lässt sich hiermit noch nicht die polynomielle Wachstumsrate dritten Grades für das Gesamtsystem erklären.

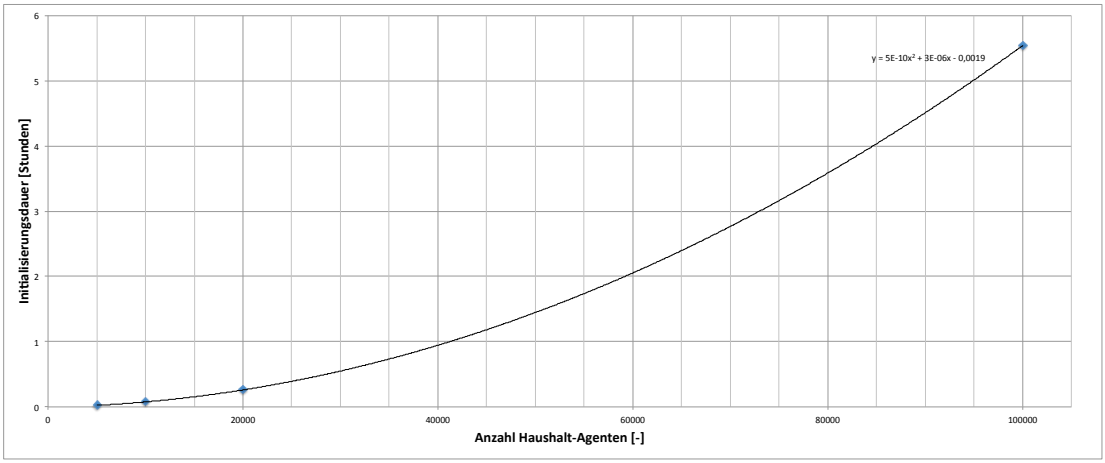


Abbildung 7.4: Initialisierungsdauer für bis zu 100.000 Haushalt-Agenten

Da in Abbildung 7.5 gezeigt werden kann, dass bei Durchführung von Simulationsläufen ohne die Nutzung von Dispatcher-, Aggregator- und SubAggregator-Agenten die Ausführungsdauer in Bezug zur Anzahl der Haushalt-Agenten linear skaliert, also eine Wachstumsrate von $\mathcal{O}(n)$ hat, muss die polynomielle Wachstumsrate dritten Grades in der hierarchischen Struktur des Simulationsszenarios und den daraus entstehenden Flaschenhälsen bzw. in der Komplexität der aus Matlab-Simulink überführten Algorithmen liegen.

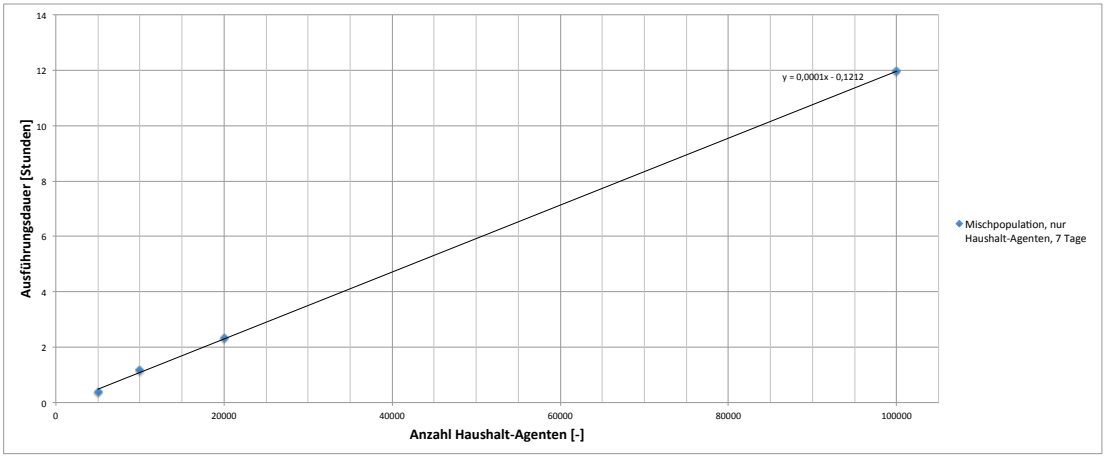


Abbildung 7.5: Simulationsdauer mit bis zu 100.000 Haushalt-Agenten für 7 Tage ohne Dispatcher, Aggregator und SubAggregatoren

Tabelle 7.4 listet die Beschleunigungsfaktoren auf, die das Simulations-Framework für die betrachtete Anzahl von Haushalt-Agenten in Bezug auf den simulierten Zeitraum ermöglicht.

Anzahl Haushalt-Agenten	**Faktor der Beschleunigung**
5.000	66,37
10.000	37,02
20.000	19,48
100.000	2,96

Tabelle 7.4: Faktoren der Beschleunigung der Simulationszeit

Dabei wird deutlich, dass das Simulations-Framework auf der für die Testläufe verwendeten Hardware, bei einer Anzahl von 100.000 Haushalt-Agenten, an die Grenzen der praktikablen Ausführungszeiten für eine Simulation stößt.

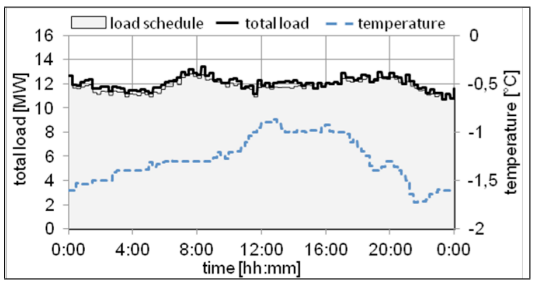
Eine weitere Beobachtung soll an dieser Stelle Erwähnung finden. Der als Hardwareplattform gewählte Server mit 64 Kernen hat zwar die Ausführung von Simulationsläufen mit 100.000 Haushalt-Agenten durch die großzügige Bestückung mit 256GB RAM erst ermöglicht, allerdings konnte das Simulations-Framework die große Zahl der Prozessorkerne nicht adäquat ausnutzen. Dies wurde zuerst auf die hierarchische Struktur des Simulationsszenarios und die damit einhergehenden Flaschenhälse zurückgeführt. Jedoch konnte auch bei der Ausführung ohne Dispatcher-, Aggregator- und SubAggregator-Agenten höchstens eine Prozessorauslastung

von 27-30% erzielt werden. Testweise wurde dann die JRockit JVM[2] genutzt, doch auch mit dieser konnte keine Verbesserung festgestellt werden. Weitere Möglichkeiten für die schlechte Ausnutzung der Prozessorkerne können im Thread-Pool von Jadex AC und im Betriebssystem selbst liegen, eine nähere Eingrenzung für dieses Verhalten war im Umfang dieser Arbeit nicht möglich.

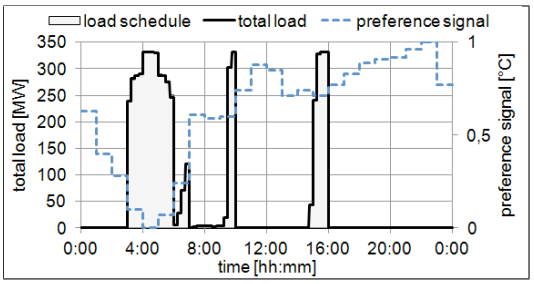
7.2.2 Domänenspezifische Ergebnisse

Durch Nutzung des Simulations-Frameworks konnten relevante Ergebnisse im Bereich des Demand Side Managements erzielt werden, die z.B. in [Braunagel u. a. (2013)] publiziert wurden. An dieser Stelle sollen diese Ergebnisse kurz eingeordnet und vorgestellt werden.

Die Abbildung 7.6(a) zeigt die dezentrale Lastverlaufsplanung mit dem Ziel eine möglichst stabile Last erzeugen. Dafür wurde eine Simulationskonfiguration mit 10.000 Haushalt-Agenten genutzt, die auf den DLL-Typ *Wärmepumpe* zurückgreift und deren Parameter *Bandlast* gesetzt ist. Der erhöhte Bedarf an Warmwasser in den Morgen- und Abendstunden lässt sich hier erkennen.



(a) Ergebnis der Planung einer stabilen Last an einem Wintertag für 10.000 Haushalt-Agenten mit Wärmepumpen-DLLs



(b) Ergebnis der Planung anhand eines Präferenzsignals an einem Wintertag für 10.000 Haushalt-Agenten mit Nachtspeicherheizung-DLLs

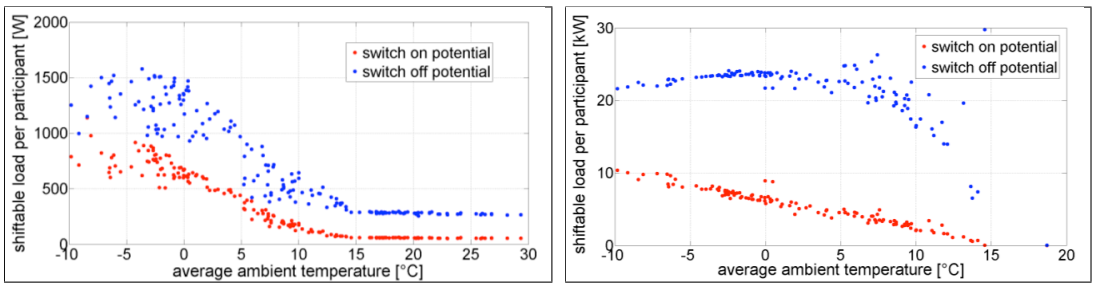
Abbildung 7.6: Lastplanung für stabile Last sowie nach Präferenzsignal. Quelle: [Braunagel u. a. (2013)]

In Abbildung 7.6(b) wird mit der Lastverlaufsplanung anhand eines Präferenzsignals ein weiteres Szenario gezeigt, dass mit dem Simulations-Framework simuliert werden kann. Dabei sendet der Dispatcher-Agent ein dimensionsloses Präferenzsignal an die Haushalt-Agenten, welches in einem Tagesverlauf Viertelstunden kennzeichnet, die bevorzugt bzw. gemieden werden sollen. Die hierfür genutzte Nachtspeicherheizung-DLL plant anhand dieses Signals

[2] http://www.oracle.com/technetwork/middleware/jrockit/overview/index.html

und der eigenen Bedarfe daraufhin einen Tageslastverlauf und versucht, diesem zu folgen. Das Szenario beinhaltet ebenfalls 10.000 Haushalt-Agenten und ist auch an einem Wintertag angesiedelt. Wie zu sehen ist, konnte der geplante Tageslastverlauf dabei eins zu eins eingehalten werden.

Ein weiteres Ziel der Simulationen ist die Ermittlung von Lastverlagerungspotenzialen für Schwärme von Haushalten mit Wärmepumpen und Nachtspeicherheizungen, um Schwankungen zwischen Stromproduktion und -verbrauch im Übertragungsnetz kurzfristig abfangen zu können. Mit dem Simulations-Framework konnten dabei positive und negative Potenziale für beide Heizsysteme ermittelt werden. Abbildung 7.7(a) zeigt die verlagerbaren Lasten in Watt pro einzelnem Haushalt mit Wärmepumpe in Abhängigkeit zur Außentemperatur, Abbildung 7.7(b) in Kilowatt pro einzelnem Haushalt mit Nachtspeicherheizung. Beide sind stark abhängig von der Außentemperatur, über 15 Grad Celsius sind praktisch keine Potenziale mehr vorhanden. Im direkten Vergleich haben Haushalte mit Nachtspeicherheizungen ein deutlich höheres Lastverlagerungspotenzial, da die installierte Leistung deutlich größer ist. Dies begründet sich in der direkten Umsetzung von Strom in Wärme und der Notwendigkeit, den Tageswärmebedarf innerhalb von acht Stunden decken zu müssen [Braunagel u. a. (2013)].



(a) Durchschnittliches Lastverlagerungspotential eines Haushalts mit Wärmepumpe innerhalb einer Stunde

(b) Durchschnittliches Lastverlagerungspotential eines Haushalts mit Nachtspeicherheizung innerhalb einer Stunde

Abbildung 7.7: Positive und negative Lastverlagerungspotentiale in Bezug zur Außentemperatur. Quelle: [Braunagel u. a. (2013)]

Bei einer angenommenen Außentemperatur von 5 Grad Celsius sind somit für einen Schwarm von z.B. 1.000 Haushalten mit Nachtspeicherheizungen Zuschaltpotenziale von bis zu 5 Megawatt und Abschaltpotenziale von bis zu 22 Megawatt vorhanden, um z.B. negative bzw. positive Regelenergie liefern zu können. Für einen gleich grossen Schwarm von Wärmepumpen bei selber Außentemperatur sind deutlich geringere Potenziale vorhanden, bis zu 1 MW beim Abschalten und bis zu 500 KW beim Zuschalten.

Eine weitere Erkenntnis, die laut [Braunagel u. a. (2013)] durch die massiven Simulationen impliziert wird, ist die die Zuverlässigkeit der Lastverlagerungs-Potenziale, die von einem solchen System angeboten werden können.

8 Zusammenfassung & Ausblick

Das Projekt „Beladealgorithmen“ zeigte im Bereich Demand Side Management für Privathaushalte große Potenziale bei der Einbindung von Privathaushalten mit Wärmepumpen und Nachtspeicherheizungen auf [Braunagel u. a. (2013)]. Dabei wurden sowohl Algorithmen für eine dezentrale Lastverlaufsplanung validiert, als auch Aussagen zu Höhe und Zuverlässigkeit von Lastverlagerungen bei großen Schwärmen von Privathaushalten mit strombasierten Heizsystemen getroffen. Dies ist ein weiteres Puzzleteil auf dem Weg zu einem reellen Smartgrid.

Ermöglicht wurden die quantitativen und Skalierungseffekte berücksichtigenden Aussagen durch das Simulations-Framework, dessen Design und Implementation in dieser Arbeit präsentiert worden sind. Das Framework erlaubt eine den Projektzielen entsprechend skalierende Nachbildung des in Matlab-Simulink bestehenden, prototypischen Simulationssystems und ist gleichzeitig deutlich performanter. Die Nachbildung ist ebenfalls im Rahmen dieser Arbeit entstanden. Ein Schlüsselfaktor dafür ist die skalierbare Integration von Matlab-Simulink 4GL-Modellen. Dadurch kann die Anzahl der simulierten Haushaltsmodelle unter einer zeitlichen Wachstumsrate von $\mathcal{O}(n)$ erhöht werden. Außerdem lässt sich der Integrationsvorgang durch die Nutzung von Technologien wie JNA und einer Template-Engine einfach in den Arbeitsablauf eingliedern. Die Integration mikroskopischer Sub-Simulationen unterscheidet diese Arbeit von bestehenden Arbeiten mit ähnlichem Fokus, die aber z.B. Verbraucher auf makroskopischer Ebene abbilden, z.B. in Form statistischer Daten oder vereinfachter Gleichungssysteme (vgl. [Ramchurn u. a. (2011); Ghaemi und Schneider (2013)]).

Das Simulations-Framework entstand mithilfe von Jadex Active Components als Ausführungsplattform. Bei der Auswertung der Performanztests konnte der Directory Service von Jadex als Flaschenhals bei der Kommunikation von Agenten für massive Simulationen von mehreren zehntausend Haushalt-Agenten identifiziert werden. Diese Problematik konnte jedoch durch die Zwischenspeicherung von Service-Interfaces auf die Initialisierungsphase des Systems reduziert werden. Dennoch bedingt diese Tatsache in Zusammenspiel mit einigen aus Matlab-Simulink überführten, nicht dezentralen Algorithmen eine Ausführungsdauer in Polynomialzeit dritten Grades für das Gesamtsystem. Die Ausführungsdauer in Bezug auf

den simulierten Zeitraum skaliert dabei linear mit $\mathcal{O}(n)$. Für die Auswertung von Performanz und Skalierbarkeit wurden massive Simulationen mit bis zu 100.000 Haushalt-Agenten auf entsprechend leistungsfähiger Hardware durchgeführt. So konnte gezeigt werden, dass das Simulations-Framework den Anforderungen entspricht.

Mit den XML-Konfigurationsdateien zur Konfiguration von Simulationsszenarien konnte erfolgreich eine einfache *domänenspezifische Sprache* (DSL) etabliert werden. Diese gibt sowohl den Entwicklern als auch den Ingenieuren des C4DSI eine unkomplizierte Methode zur Konfiguration von Simulationsläufen an die Hand.

8.1 Ausblick

Die Architektur des in dieser Arbeit vorgestellten Simulations-Frameworks ist sehr spezifisch auf die Problemstellung zugeschnitten, bietet aber durch den Service-Komponenten basierten Entwurf und die Integration lose gekoppelter 4GL-Modelle vielversprechende Ansätze für adaptivere Folgearbeiten. In [Preisler u. a. (2014b)] wird ein Architekturentwurf eines adaptiven Simulations- und Informationssystems mit dem Ziel vorgestellt, verschiedene Simulationssysteme mikroskopischer und makroskopischer Art miteinander zu verbinden. Dabei werden Simulationen und Sub-Simulationen als Dienste angeboten und kommunizieren mittels wohldefinierter Interfaces über einen Service-Bus miteinander. Um Flaschenhälse zu vermeiden wird auf eine zentrale Manager-Komponente zugunsten von dezentralen, den einzelnen Simulationskomponenten zugeordneten „Adaptive Manager“-Komponenten verzichtet.

Die als optional gekennzeichnete Anforderung, Matlab-Algorithmen analog zu den Matlab-Simulink-Modelle direkt in das Simulations-Framework zu integrieren, konnte aufgrund des zu hohen zeitlichen Aufwands nicht innerhalb dieser Arbeit erfüllt werden. Ein Weg dieses Ziel zu erreichen wird in [Preisler u. a. (2014a)] beschrieben, der auf den Erfahrungen der hier vorgestellten Arbeit basiert. Dabei wird die skalierbare Integration von 4GL-Modellen und Algorithmen durch eine lose Kopplung und Kapselung dieser 4GL-Konstrukte mittels etablierter Webtechnologien wie RESTful-Webservices und davor geschalteter Lastverteilung ermöglicht.

Weiterhin wird für Folgeprojekte, die einer Service-Komponenten-Architektur folgen, die Nutzung eines *Dependency-Injection-Frameworks* (Kapitel 7.1.1), empfohlen, um für diese lose gekoppelten Systeme einfacher Unit- und Integrationstests vornehmen zu können.

Da für diese Arbeit die Ausgabe der Ergebnisse in Form von .MAT-Dateien äußerst wichtig war, wurde auf die Nutzung einer Datenbank verzichtet. Um aber z.B. Datenverluste durch vorzeitige Simulationsabbrüche zu vermeiden, wäre die Erweiterung des Simulations-Frameworks um eine Datenbank und eine dazugehörige Datenbankanbindung von Interesse. Auch die Nutzung eines Persistenzframeworks wie z.B. Hibernate[1] wäre denkbar.

[1] http://hibernate.org/

Literaturverzeichnis

[Ahat u. a. 2013] Ahat, M ; Amor, SB ; Bui, Marc: Agent based model of smart grids for ecodistricts. In: *Proceedings of the Fourth Symposium on Information and Communication Technology*, ACM, 2013 (SoICT '13), S. 45–52. – ISBN 9781450324540

[Balthasar u. a. 2010] Balthasar, Gregor ; Sudeikat, Jan ; Renz, Wolfgang: On the decentralized coordination of herding activities: a Jadex-based solution. In: *Annals of Mathematics and Artificial Intelligence* 59 (2010), März, Nr. 3-4, S. 411–426. – ISSN 1012-2443

[Beydeda u. a. 2006] Beydeda, Sami (Hrsg.) ; Book, Matthias (Hrsg.) ; Gruhn, Volker (Hrsg.): *Model-Driven Software Development*. Springer Berlin Heidelberg, 2006

[Bratman 1987] Bratman, Michael: *Intention, Plans, and Practical Reason*. Harvard University Press, 1987. – ISBN 978-0-674-45818-5

[Braubach und Pokahr 2013] Braubach, Lars ; Pokahr, Alexander: The Jadex Project: Simulation. In: Ganzha, Maria (Hrsg.) ; Jain, Lakhmi C. (Hrsg.): *Multiagent Systems and Applications* Bd. 45. Springer Berlin Heidelberg, 2013, S. 107–128. – ISBN 978-3-642-33322-4

[Braubach u. a. 2005] Braubach, Lars ; Pokahr, Alexander ; Lamersdorf, Winfried: Extending the Capability Concept for Flexible BDI Agent Modularization. In: *Proc. of PROMAS-2005*, 2005

[Braubach u. a. 2013] Braubach, Lars ; Pokahr, Alexander ; Lamersdorf, Winfried: Jadex Active Components: A Unified Execution Infrastructure for Agents and Workflows. In: *Advanced Computational Technologies* (2013), S. 128–149

[Braunagel u. a. 2013] Braunagel, Johannes ; Vuthi, Petrit ; Renz, Wolfgang ; Schäfers, Hans ; Wiechmann, Holger ; Zarif, Hojat: Determination of load schedules and load shifting potentials of a high number of electrical consumers using mass simulation. In: *22nd International Conference on Electricity Distribution*, 2013, S. 10–13

[Bresciani u. a. 2004] Bresciani, Paolo ; Perini, Anna ; Giorgini, P: Tropos: An agent-oriented software development methodology. In: *Autonomous Agents and Multi-Agent Sytems* (2004), S. 203–236

[Bunn und Oliveira 2001] Bunn, D.W. ; Oliveira, F.S.: Agent-based simulation-an application to the new electricity trading arrangements of England and Wales. In: *IEEE Transactions on Evolutionary Computation* 5 (2001), Nr. 5, S. 493–503. – ISSN 1089778X

[Chassin u. a. 2008] Chassin, DP ; Schneider, K ; Gerkensmeyer, C: GridLAB-D: An open-source power systems modeling and simulation environment. In: *Transmission and Distribution Conference and Exposition*, IEEE, 2008, S. 1–5. – ISBN 9781424419043

[Coyne u. a. 2008] Coyne, ME ; Graham, SR ; Hopkinson, KM ; Kurkowski, SH: A methodology for unit testing actors in proprietary discrete event based simulations. In: *Simulation Conference, 2008. WSC 2008. Winter*, 2008, S. 1012–1019. – ISBN 9781424427086

[Davidsson 2001] Davidsson, Paul: Multi Agent Based Simulation: Beyond Social Simulation. In: Moss, Scott (Hrsg.) ; Davidsson, Paul (Hrsg.): *Multi-Agent-Based Simulation* Bd. 1979. Springer Berlin Heidelberg, 2001, S. 97–107. – ISBN 978-3-540-41522-0

[Deindl u. a. 2008] Deindl, Matthias ; Block, Carsten ; Vahidov, Rustam ; Neumann, Dirk: Load Shifting Agents for Automated Demand Side Management in Micro Energy Grids. In: *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems* (2008), Oktober, S. 487–488. ISBN 978-0-7695-3404-6

[Dethlefs 2014] Dethlefs, Tim: *Ein verbraucherorientiertes Energiesystem für Smart Grids : Entwicklung eines Multi-Agenten-Systems zur dezentralen Optimierung*. Berliner Tor 5, 20099 Hamburg, HAW Hamburg, Diplomarbeit, 2014

[ENTSOE 2004] ENTSOE: *Load-Frequency Control and Performance*. 2004

[Ferber 2001] Ferber, Jacques: *Multiagentensysteme*. Addison-Wesley, 2001. – ISBN 3-8273-1679-0

[Fowler 2004] Fowler, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. 2004. – URL http://martinfowler.com/articles/injection.html. – Zugriffsdatum: 28.09.2014

[Gellings 1985] Gellings, CW: The concept of demand-side management for electric utilities. In: *Proceedings of the IEEE* 73 (1985), Nr. 10, S. 1468–1470

[Ghaemi und Schneider 2013] Ghaemi, Sara ; Schneider, Simon: Potential analysis of residential Demand Response using GridLAB-D. In: *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society* (2013), November, S. 8039–8045. ISBN 978-1-4799-0224-8

[IEC 60038 2009] IEC 60038: *CENELEC-Normspannungen*. 2009

[Logenthiran und Srinivasan 2011] Logenthiran, T ; Srinivasan, D: Multi-Agent System for Demand Side Management in Smart Grid. In: *Power Electronics and Drive Systems (PEDS), 2011 IEEE Ninth International Conference* 0954 (2011), Nr. December, S. 5–8. ISBN 9781457700019

[Marino und Rowley 2009] Marino, Jim ; Rowley, Michael: *Understanding SCA (Service Component Architecture)*. 1st. Addison-Wesley Professional, 2009. – ISBN 0321515080, 9780321515087

[Martin 1982] Martin, James: *Application Development Without Programmers*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1982. – ISBN 0130389439

[MathWorks] MathWorks: *MAT-File Format*. – URL http://www.mathworks.com/help/pdf_doc/matlab/matfile_format.pdf

[McArthur u. a. a] McArthur, S.D.J. ; Davidson, E.M. ; Catterson, V.M. ; Dimeas, A.L. ; Hatziargyriou, N.D. ; Ponci, F. ; Funabashi, T.: Multi-agent Systems for Power Engineering Applications-Part I: Technologies, Standards and Tools for Building Multi-agent Systems. In: *IEEE TRANSACTIONS ON POWER SYSTEMS* Nr. 4, S. 1753–1759

[McArthur u. a. b] McArthur, S.D.J. ; Davidson, E.M. ; Catterson, V.M. ; Dimeas, A.L. ; Hatziargyriou, N.D. ; Ponci, F. ; Funabashi, T.: Multi-agent Systems for Power Engineering Applications-Part II: Technologies, Standards and Tools for Building Multi-agent Systems. In: *IEEE TRANSACTIONS ON POWER SYSTEMS* Nr. 4, S. 1753–1759

[Mengistu u. a. 2008] Mengistu, Dawit ; Tröger, Peter ; Lundberg, Lars ; Davidsson, Paul: Scalability in Distributed Multi-Agent Based Simulations: The JADE Case. In: *2008 Second International Conference on Future Generation Communication and Networking Symposia* (2008), Dezember, S. 93–99. ISBN 978-1-4244-3430-5

[Merz 2003] Merz, Peter: *Moderne heuristische Optimierungsverfahren: Meta-Heuristiken*. Wilhelm-Schickard-Institut für Informatik. Universität Tübingen, 2003

[Oracle] Oracle: *JAXB Architecture.* – URL http://docs.oracle.com/javase/tutorial/jaxb/intro/arch.html

[Pokahr und Braubach 2011] Pokahr, Alexander ; Braubach, Lars: Active Components: A Software Paradigm for Distributed Systems. In: *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology* (2011), August, S. 141–144. ISBN 978-1-4577-1373-6

[Pokahr u. a. 2010] Pokahr, Alexander ; Braubach, Lars ; Jander, Kai: Unifying Agent and Component Concepts - Jadex Active Components. In: *Proceedings of Seventh German conference on Multi-Agent System TEchnologieS (MATES-2010)* (2010)

[Poslad 2007] Poslad, Stefan: Specifying protocols for multi-agent systems interaction. In: *ACM Transactions on Autonomous and Adaptive Systems* 2 (2007), November, Nr. 4, S. 15–es. – ISSN 15564665

[Preisler u. a. 2014a] Preisler, Thomas ; Balthasar, Gregor ; Dethlefs, Tim ; Renz, Wolfgang: Scalable Integration of 4GL-Models and Algorithms for Massive Smart Grid Simulations and Applications. In: Jorge Marx Gómez, Ute Vogel Andreas Winter Barbara Rapp Nils G. (Hrsg.): *Proceedings of the 28th Conference on Environmental Informatics - Informatics for Environmental Protection, Sustainable Development and Risk Management*, 2014, S. 341–348. – ISBN 9783814223179

[Preisler u. a. 2014b] Preisler, Thomas ; Balthasar, Gregor ; Dethlefs, Tim ; Renz, Wolfgang: Servicekomponenten-basierte Architektur für mikroskopische und makroskopische Simulation der städtischen Energieversorgung. In: *VDE-Kongress 2014 „Smart Cities“ (noch nicht veröffentlicht)*, 2014

[Ramchurn u. a. 2011] Ramchurn, S ; Vytelingum, Perukrishnen ; Rogers, Alex ; Jennings, Nick: Agent-based control for decentralised demand side management in the smart grid. In: *The Tenth International Conference on Autonomous Agents and Multiagent Systems AAMAS 2011 (2011)*, 2011, S. 5–12

[Rao und Georgeff 1995] Rao, AS ; Georgeff, MP: BDI Agents: From Theory to Practice. In: *ICMAS* (1995)

[Rohjans und Lehnhoff 2013] Rohjans, S ; Lehnhoff, 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. ISBN 9781479929849

[Schäfers u. a. 2013] Schäfers, Hans ; Hey, Bastian ; Kühl, Matthias ; Schubert, Franz ; Renz, Wolfgang: Power-to-Gas und Demand Side Management - Schlüsselfaktoren für den Erfolg der Energiewende. In: *Fortschritt-Berichte VDI Reihe* 6 (2013), Nr. 608. ISBN 978-3-18-360806-5

[Schäfers u. a. 2010] Schäfers, Hans ; Lüdemann, Karin ; Borst, Detlef ; Schubert, Franz: Demand Side Management (DSM) in Deutschland - Potenziale und Märkte. In: *Fortschritt-Berichte VDI Reihe* 6 (2010), Nr. 593. ISBN 978-3-18-359306-4

[Schneider u. a. 2011] Schneider, Kevin P. ; Fuller, Jason C. ; Chassin, David P.: Multi-State Load Models for Distribution System Analysis. In: *IEEE Transactions on Power Systems* 26 (2011), Nr. 4, S. 2425–2433

[Schuldt u. a. 2008] Schuldt, Arne ; Gehrke, Jan D. ; Werner, Sven: Designing a Simulation Middleware for FIPA Multiagent Systems. In: *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology - Volume 02*, 2008 (WI-IAT '08), S. 109–113. – ISBN 978-0-7695-3496-1

[Sutter und Larus 2005] Sutter, Herb ; Larus, James: Software and the concurrency revolution. In: *Queue* (2005), Nr. September

[Wang u. a. 2012] Wang, Dan ; Wit, Braydon D. ; Parkinson, Simon ; Fuller, Jason ; Chassin, David ; Crawford, Curran ; Djilali, Ned: A Test Bed for Self-regulating Distribution Systems : Modeling Integrated Renewable Energy and Demand Response in the GridLAB-D / MATLAB Environment. In: *Innovative Smart Grid Technologies (ISGT)*, IEEE, 2012, S. 1–7. – ISBN 9781457721595

[Warden u. a. 2010] Warden, Tobias ; Porzel, Robert ; Gehrke, JD ; Herzog, Otthein: Towards Ontology-Based Multiagent Simulations: The Plasma Approach. In: *ECMS* (2010)

[Weiss 2000] Weiss, Gerhard: *Multiagent systems : a modern approach to distributed artificial intelligence*. Cambridge, Mass. [u.a.] : MIT Press, 2000

[Wooldridge und Jennings 1995] Wooldridge, M. ; Jennings, N. R.: Intelligent Agents: Theory and Practice. In: *Knowledge Engineering Review 10(2)*, 1995

[Wooldridge 2002] Wooldridge, Michael: *An introduction to multiagent systems*. Wiley, 2002

A Anhang

A.1 Testfälle

Tabellarisch aufgetragene Testfälle für Komponenten-, Integrations-, System- und Performanztests.

ID	1
Name	SIM-DLL-POS
Testszenario	Es soll die Funktionalität eines als DLL exportierten Modells geprüft werden. Dazu werden in Matlab-Simulink verifizierte Testdatensätze von Ein- und Ausgabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden dazu der im JNA-Interface definierten Funktion übergeben und deren Ergebnisse mit den erwarteten Daten verglichen.
Test-Entität(en)	exportierte DLL und JNA-Interface
Vorbedingungen	-
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	Testdatensätze in Form von .MAT-Dateien
Nachbedingungen	-
Art	Komponententest
zug. Anforderungs-ID(s)	4, 10
Abhängigkeiten	-

Tabelle A.1: Testfall SIM-DLL-POS, positiver DLL-Test

ID	2
Name	SIM-DLL-NEG
Testszenario	Es sollen fehlerhafte Eingabedaten für ein als DLL exportiertes Modell geprüft werden. Dazu werden in Matlab-Simulink generierte Testdatensätze von fehlerhaften bzw. unvollständigen Eingabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden dazu der im JNA-Interface definierten Funktion übergeben und die DLL soll einen Fehlercode ausgeben.
Test-Entität(en)	exportierte DLL und JNA-Interface
Vorbedingungen	-
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	Ausgabe eines Fehlercodes
Nachbedingungen	-
Art	Komponententest
zug. Anforderungs-ID(s)	4, 10
Abhängigkeiten	-

Tabelle A.2: Testfall SIM-DLL-NEG, negativer DLL-Test

ID	3
Name	SIM-SUBAGG-ALG1-POS
Testszenario	Der aus Matlab-Simulink überführte Algorithmus zur Aggregation der Haushalt-Daten im SubAggregator soll auf ordnungsgemäße Funktion getestet werden. Dazu werden in Matlab-Simulink verifizierte Testdatensätze von Ein- und Ausgabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *transmitMinuteValues*-Methode aufbereitet und ihr übergeben. Die erwarteten Ausgabedaten werden dann mit den ausgegebenen auf Gleichheit geprüft.
Test-Entität(en)	*transmitMinuteValues*
Vorbedingungen	Aufbereitung der Daten
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	Testdatensätze in Form von .MAT-Dateien
Nachbedingungen	Methode ohne Fehlermeldung durchlaufen
Art	Komponententest
zug. Anforderungs-ID(s)	5, 10
Abhängigkeiten	-

Tabelle A.3: Testfall SIM-SUBAGG-ALG1-POS, positiver *transmitMinuteValues*-Test

ID	4
Name	SIM-SUBAGG-ALG1-NEG
Testszenario	Es sollen fehlerhafte Eingabedaten für den aus Matlab-Simulink überführten Algorithmus zur Aggregation der Haushalt-Daten im SubAggregator getestet werden. Dazu werden in Matlab-Simulink generierte Testdatensätze von fehlerhaften bzw. unvollständigen Eingabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *transmitMinuteValues*-Methode aufbereitet und ihr übergeben. Die Methode soll darauf mit einer Fehlermeldung abbrechen.
Test-Entität(en)	*transmitMinuteValues*
Vorbedingungen	Aufbereitung der Daten
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	-
Nachbedingungen	Abbruch der Methode mit Fehlermeldung
Art	Komponententest
zug. Anforderungs-ID(s)	5, 10
Abhängigkeiten	-

Tabelle A.4: Testfall SIM-SUBAGG-ALG1-NEG, negativer *transmitMinuteValues*-Test

ID	5
Name	SIM-SUBAGG-ALG2-POS
Testszenario	Der aus Matlab-Simulink überführte Algorithmus zur Aufteilung der vom Aggregator kommenden Anweisungen auf die Haushalt-Agenten im SubAggregator soll auf ordnungsgemäße Funktion getestet werden. Dazu werden in Matlab-Simulink verifizierte Testdatensätze von Ein- und Ausgabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *proposeSignals*-Methode aufbereitet und ihr übergeben. Die erwarteten Ausgabedaten werden dann mit den ausgegebenen auf Gleichheit geprüft.
Test-Entität(en)	*proposeSignals*
Vorbedingungen	Aufbereitung der Daten
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	Testdatensätze in Form von .MAT-Dateien
Nachbedingungen	Methode ohne Fehlermeldung durchlaufen
Art	Komponententest
zug. Anforderungs-ID(s)	5, 10
Abhängigkeiten	-

Tabelle A.5: Testfall SIM-SUBAGG-ALG2-POS, positiver *proposeSignals*-Test

ID	6
Name	SIM-SUBAGG-ALG2-NEG
Testszenario	Es sollen fehlerhafte Eingabedaten für den aus Matlab-Simulink überführten Algorithmus zur Aufteilung der vom Aggregator kommenden Anweisungen auf die Haushalt-Agenten im Sub-Aggregator getestet werden. Dazu werden in Matlab-Simulink generierte Testdatensätze von fehlerhaften bzw. unvollständigen Eingabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *proposeSignals*-Methode aufbereitet und ihr übergeben. Die Methode soll darauf mit einer Fehlermeldung abbrechen.
Test-Entität(en)	*proposeSignals*
Vorbedingungen	Aufbereitung der Daten
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	-
Nachbedingungen	Abbruch der Methode mit Fehlermeldung
Art	Komponententest
zug. Anforderungs-ID(s)	5, 10
Abhängigkeiten	-

Tabelle A.6: Testfall SIM-SUBAGG-ALG2-NEG, negativer *proposeSignals*-Test

ID	7
Name	SIM-AGG-POS
Testszenario	Der aus Matlab-Simulink überführte Algorithmus zur Aggregation der SubAggregator-Daten im Aggregator sowie der Aufteilung der vom Dispatcher kommenden Daten soll auf ordnungsgemäße Funktion getestet werden. Dazu werden in Matlab-Simulink verifizierte Testdatensätze von Ein- und Ausgabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *transmitSubAggregatedMinuteValues*-Methode aufbereitet und ihr übergeben. Die erwarteten Ausgabedaten werden dann mit den ausgegebenen auf Gleichheit geprüft.
Test-Entität(en)	*transmitSubAggregatedMinuteValues*
Vorbedingungen	Aufbereitung der Daten
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	Testdatensätze in Form von .MAT-Dateien
Nachbedingungen	Methode ohne Fehlermeldung durchlaufen
Art	Komponententest
zug. Anforderungs-ID(s)	5, 10
Abhängigkeiten	-

Tabelle A.7: Testfall SIM-AGG-POS, positiver *transmitSubAggregatedMinuteValues*-Test

ID	8
Name	SIM-AGG-NEG
Testszenario	Es sollen fehlerhafte Eingabedaten für den aus Matlab-Simulink überführten Algorithmus zur Aggregation der SubAggregator-Daten im Aggregator sowie der Aufteilung der vom Dispatcher kommenden Daten getestet werden. Dazu werden in Matlab-Simulink generierte Testdatensätze von fehlerhaften bzw. unvollständigen Eingabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *transmitSubAggregatedMinuteValues*-Methode aufbereitet und ihr übergeben. Die Methode soll darauf mit einer Fehlermeldung abbrechen.
Test-Entität(en)	*transmitSubAggregatedMinuteValues*
Vorbedingungen	Aufbereitung der Daten
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	-
Nachbedingungen	Abbruch der Methode mit Fehlermeldung
Art	Komponententest
zug. Anforderungs-ID(s)	5, 10
Abhängigkeiten	-

Tabelle A.8: Testfall SIM-AGG-NEG, negativer *transmitSubAggregatedMinuteValues*-Test

ID	9
Name	SIM-DISP-POS
Testszenario	Der aus Matlab-Simulink überführte Algorithmus zur Auswertung der aggregierten Daten des Gesamtsystems im Dispatcher, sowie zur Erstellung von Regelungsanweisungen und Lastverlagerungsanforderungen soll auf ordnungsgemäße Funktion getestet werden. Dazu werden in Matlab-Simulink verifizierte Testdatensätze von Ein- und Ausgabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *transmitAggregatedMinuteValues*-Methode aufbereitet und ihr übergeben. Die erwarteten Ausgabedaten werden dann mit den ausgegebenen auf Gleichheit geprüft.
Test-Entität(en)	*transmitAggregatedMinuteValues*
Vorbedingungen	Aufbereitung der Daten
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	Testdatensätze in Form von .MAT-Dateien
Nachbedingungen	Methode ohne Fehlermeldung durchlaufen
Art	Komponententest
zug. Anforderungs-ID(s)	5, 10
Abhängigkeiten	-

Tabelle A.9: Testfall SIM-DISP-POS, positiver *transmitAggregatedMinuteValues*-Test

ID	10
Name	SIM-DISP-NEG
Testszenario	Es sollen fehlerhafte Eingabedaten für den aus Matlab-Simulink überführten Algorithmus zur Auswertung der aggregierten Daten des Gesamtsystems im Dispatcher, sowie zur Erstellung von Regelungsanweisungen und Lastverlagerungsanforderungen getestet werden. Dazu werden in Matlab-Simulink generierte Testdatensätze von fehlerhaften bzw. unvollständigen Eingabedaten in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *transmitAggregatedMinuteValues*-Methode aufbereitet und ihr übergeben. Die Methode soll darauf mit einer Fehlermeldung abbrechen.
Test-Entität(en)	*transmitAggregatedMinuteValues*
Vorbedingungen	Aufbereitung der Daten
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	-
Nachbedingungen	Abbruch der Methode mit Fehlermeldung
Art	Komponententest
zug. Anforderungs-ID(s)	5, 10
Abhängigkeiten	-

Tabelle A.10: Testfall SIM-DISP-NEG, negativer *transmitAggregatedMinuteValues*-Test

ID	11
Name	SIM-ALG-INTEGRATION
Testszenario	Das Zusammenspiel der überführten Algorithmen für SubAggregator, Aggregator und Dispatcher soll geprüft werden. Dazu werden in Matlab-Simulink verifizierte Testdatensätze von Ein- und Ausgabedaten für den SubAggregator in Form von .MAT-Dateien eingelesen. Die Eingabedaten werden für die *transmitMinuteValues*-Methode aufbereitet und ihr übergeben. Mit den daraus folgenden Ergebnissen startet ein Durchlauf der Methoden *transmitSubAggregatedMinuteValues*, *transmitAggregatedMinuteValues* und *proposeSignals*. Die erwarteten Ausgabedaten werden dann mit den von *proposeSignals* ausgegebenen auf Gleichheit geprüft.
Test-Entität(en)	*transmitMinuteValues*, *transmitSubAggregatedMinuteValues*, *transmitAggregatedMinuteValues*, *proposeSignals*
Vorbedingungen	Aufbereitung der Daten, Aufbau der *Teststrecke*
Eingabedaten	Testdatensätze in Form von .MAT-Dateien
erwartete Ausgabedaten	Testdatensätze in Form von .MAT-Dateien
Nachbedingungen	Methoden ohne Fehlermeldung durchlaufen
Art	Integrationstest
zug. Anforderungs-ID(s)	2, 5, 10
Abhängigkeiten	SUBAGG-ALG1-POS, SUBAGG-ALG1-NEG, SUBAGG-ALG2-POS, SUBAGG-ALG2-NEG, SIM-AGG-POS, SIM-AGG-NEG, SIM-DISP-POS, SIM-DISP-NEG

Tabelle A.11: Testfall SIM-ALG-INTEGRATION, Integrationstest der Matlab-Simulink Algorithmen

ID	12
Name	SIM-MATLAB
Testszenario	Die Funktionalität des Simulations-Frameworks als ganzes soll verifiziert werden. Dazu wird mithilfe der Konfigurationsdatei ein Simulationsszenario erstellt, das dem in Matlab-Simulink aufgebauten Szenario entspricht. In den Modellen beider Systeme müssen dazu die stochastischen Größen auf feste Größen umgeschaltet werden. Es werden in Matlab-Simulink verifizierte Testdatensätze von Ausgabedaten des Dispatchers für einen Simulationslauf von einer Woche in Form von .MAT-Dateien erstellt. Diese erwarteten Ausgabedaten werden nach dem Simulations mit der vom Dispatcher-Agenten erzeugten .MAT-Datei auf Gleichheit geprüft.
Test-Entität(en)	Gesamtsystem
Vorbedingungen	stochastische Größen auf feste Größen umgestellt
Eingabedaten	Konfigurationsdatei mit Abbild des Matlab-Simulink Szenarios
erwartete Ausgabedaten	Testdatensatz in Form einer .MAT-Dateien
Nachbedingungen	Simulation ohne Fehlermeldung durchlaufen, JVM beendet
Art	Systemtest
zug. Anforderungs-ID(s)	1, 2, 3, 4, 5, 8, 9, 10
Abhängigkeiten	SIM-DLL-POS, SIM-DLL-NEG, SIM-ALG-INTEGRATION

Tabelle A.12: Testfall SIM-MATLAB, Verifikation des Simulations-Frameworks

ID	13
Name	SIM-VALID-20K
Testszenario	Die Korrektheit des Simulations-Frameworks für eine große Anzahl von Haushalts-Modellen soll validiert werden. Dazu wird mithilfe der Konfigurationsdatei ein Simulationsszenario mit 20.000 Haushalt-Modellen erstellt. Die Daten der am Ende des Simulationslaufes erzeugten .MAT-Datei werden von einem Team aus Experten einem Plausibilitäts-Check unterzogen.
Test-Entität(en)	Gesamtsystem
Vorbedingungen	-
Eingabedaten	Konfigurationsdatei mit 20.000 Haushalt-Agenten
erwartete Ausgabedaten	Plausibilitäts-Check durch Experten-Team
Nachbedingungen	Simulation ohne Fehlermeldung durchlaufen, JVM beendet
Art	Systemtest
zug. Anforderungs-ID(s)	1, 2, 3, 4, 5, 7, 8, 9, 10, 13
Abhängigkeiten	SIM-MATLAB

Tabelle A.13: Testfall SIM-VALID-20K, Validierung des Simulations-Frameworks für 20.000 Haushalts-Modelle

ID	14
Name	SIM-PERF-VERGLEICH
Testszenario	Die Performanz des Simulations-Frameworks soll mit der von Matlab-Simulink verglichen werden. Dazu wird mithilfe der Konfigurationsdatei ein Simulationsszenario erstellt, das dem in Matlab-Simulink aufgebauten Szenario entspricht. Dann wird die Zeit für beide Systeme mit gleichen Simulationsszenarien von 1, 7, 14, 30,5 und 149 simulierten Tagen gemessen.
Test-Entität(en)	Gesamtsystem
Vorbedingungen	Nutzung gleicher Hardware
Eingabedaten	Konfigurationsdateien mit Abbildern des Matlab-Simulink Szenarios für 1 / 7 / 14 / 30,5 / 149 Simulationstage
erwartete Ausgabedaten	Zeitmessung
Nachbedingungen	Simulation ohne Fehlermeldung durchlaufen, JVM beendet
Art	Performanztest
zug. Anforderungs-ID(s)	1, 2, 3, 4, 5, 8, 9, 10, 14
Abhängigkeiten	SIM-MATLAB

Tabelle A.14: Testfall SIM-PERF-VERGLEICH, Performanzvergleich Simulations-Framework vs. Matlab-Simulink

ID	15
Name	SIM-PERF-SKAL
Testszenario	Die Performanz und Skalierbarkeit des Simulations-Frameworks soll getestet werden. Dazu werden mithilfe der Konfigurationsdateien Simulationsszenarien mit zwei verschiedenen DLL-Typen und jeweils 5.000, 10.000, 20.000 sowie 100.000 Haushalt-Agenten erstellt. Für alle Szenarien wird der Simulationszeitraum von jeweils 7 und 14 simulierten Tagen gesetzt und die Zeit gemessen.
Test-Entität(en)	Gesamtsystem
Vorbedingungen	-
Eingabedaten	Konfigurationsdateien für Misch-Szenarien von zwei DLL-Typen mit 5.000 / 10.000 / 20.000 / 100.000 Haushalt-Agenten und jeweils 7 Simulationstagen
erwartete Ausgabedaten	Zeitmessung
Nachbedingungen	Simulation ohne Fehlermeldung durchlaufen, JVM beendet
Art	Performanztest
zug. Anforderungs-ID(s)	1, 2, 3, 4, 5, 8, 9, 10, 13, 14
Abhängigkeiten	SIM-MATLAB

Tabelle A.15: Testfall SIM-PERF-SKAL, Performanz- und Skalierbarkeits-Test des Simulations-Frameworks

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 26. November 2014 Gregor Balthasar