CARL VON OSSIETZKY universität OLDENBURG

BUSINESS ENGINEERING
Carl von Ossietzky Universität Oldenburg

PG MOBILITY

ABSCHLUSSBERICHT DER PROJEKTGRUPPE

Teilnehmer:	Kolja Blohm	Mathias Ringe
	Tammo Buss	Florian Steinmeyer
	Hong-Son Dang	Christian van Göns
	Christian Denker	Sven Vorlauf
	Stefan Kai Kramer	Bastian Wagenfeld
	Arne Laverentz	Marcel Wilkens

Betreuer: Prof. Dr.-Ing. Axel Hahn, apl. Prof. Dr.-Ing. Jürgen Sauer, Dipl.-Wirt.-Inf. Tim Hoerstebrock

Carl-von-Ossietzky Universität Oldenburg
Department für Informatik
Abteilung Business Engineering

Datum: 10.04.2012

Kurzfassung

Rechnerbasierte Methoden sind heutzutage Standard für Problemlösungsansätze in der Verkehrsdomäne. Eine besonders vielversprechende Methode ist dabei die Simulation. In der Projektgruppe Mobility wurde in diesem Zusammenhang ein Framework zur Erstellung von agentenbasierten Verkehrssimulationen entwickelt.

Als Grundlage für dieses Framework dient das ebenfalls in der Projektgruppe entwickelte Multi-Agenten-Simulationsframework JASON, mit dem es möglich ist, nachrichtenbasierte Agenten-Simulationen in Java zu erstellen.

Aufbauend auf JASON erfolgte dann ein modellbasierter Ansatz zur Spezialisierung eines zweiten Frameworks auf die Verkehrsdomäne. Mit diesem können eigene Verkehrssimulationen erstellt werden. Hierzu werden Schnittstellen zur Modellbildung und zur Erweiterung der Agenten bereitgestellt.

Zur Validierung wurde in Zusammenarbeit mit der Reederei Frisia ein Szenario zur Einführung von LNG als alternative Energiequelle im Schiffsbereich erarbeitet. Die Umsetzung des Szenarios auf der Basis des Verkehrssimulationsframeworks zeigt eine der vielfältigen Nutzungsmöglichkeiten des Frameworks zur Analyse von komplexen Problemszenarien auf.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Formelverzeichnis

V. Abkürzungsverzeichnis

BIRT	Business Intelligence and Reporting Tools
CSS	Cascading Style Sheet
ECA	Emission Controlled Area
EMF	Eclipse Modeling Framework
FIPA	Foundation of Intelligent Physical Agents
GUI	Graphical User Interface
HFO	Schweröl (engl.: heavy fuel oil)
HTML	Hyper Text Markup Language
IEEE	Institute of Electrical and Electronics Engineers
IMO	International Maritime Organization
JADE	Java Agent Development Framework
JASON	Kompositum aus JADE und MASON
JDBC	Java Database Connectivity
JPA	Java Persistence API
LNG	Liquid Natural Gas
MAS	Multi-Agenten-System
MASON	Multi Agent Simulator of Neighborhoods (or Networks)
MDO	Marine Diesel Oil (deutsch: Marinedieselöl)
OSGi	Open Services Gateway initiative framework
PDF	Portable Document Format
RCP	Rich Client Application
SQL	Structured Query Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Einleitung

In diesem Kapitel wird ein einführender Überblick über die Projektgruppe Mobility gegeben, die im Rahmen des Masterstudiengangs Wirtschaftsinformatik in der Abteilung Business Engineering der Carl von Ossietzky Universität Oldenburg stattgefunden hat. Der Zeitraum der Projektgruppe war von Beginn des Sommersemesters 2011 bis zum Ende des Wintersemesters 2011/2012.

Im ersten Teil dieses Kapitels wird die Thematik und Aufgabe der Projektgruppe motiviert, bevor auf die damit verbundenen Problemstellungen und Lösungsansätze eingegangen wird. Darauf aufbauend werden im zweiten Abschnitt die angestrebten Ziele und Ergebnisse der Projektgruppe konkretisiert. Abschließend wird in diesem Kapitel ein Überblick über den gesamten Abschlussbericht gegeben, indem der Aufbau und die inhaltliche Struktur des Dokuments erläutert werden.

1.1 Motivation

Ein interessanter Gegenstand der Forschung sind Verkehrssysteme. Grundsätzlich besteht ein Interesse an der Untersuchung von innovativen Antriebskonzepten oder alternativen und nachhaltig-orientierten Treibstoffen in unterschiedlichen Domänen von Verkehrssystemen, was oftmals durch ein steigendes Umweltbewusstsein begründet ist. Ein Beispiel dafür ist die Forschung im Bereich der Elektromobilität. Dabei handelt es sich um ein Antriebskonzept, das elektrische Energie für die Fortbewegung von Kraftfahrzeugen nutzt.

Eine weitere interessante Domäne der Verkehrssysteme ist die Schifffahrt. Die Schifffahrt stellt als Verkehrsträger einen Großteil des weltweiten Gütertransports dar, dessen Wachstum in den letzten Jahrzehnten im Rahmen der Globalisierung enorm zugenommen hat (World Ocean Review 2010).

Schiffe verwenden dabei zu einem großen Teil den Treibstoff Schweröl (engl.: Heavy Fuel Oil, HFO), der mit vielen Schadstoffen behaftet ist (MAGALOC Projekt 2008). Auf die damit verbundene Umweltverschmutzung reagiert die International Maritime Organization (IMO) mit der Verschärfung von Umweltauflagen, die in naher Zukunft mit diesem Treibstoff nicht erfüllt werden können. Es müssen also Alternativen entwickelt werden, die Schweröl als primären Treibstoff ablösen. Als vielversprechend erweist sich dabei Liquid Natural Gas (LNG), das bereits für einzelne Schiffe für den Fährverkehr in Norwegen zum Einsatz kommt.

Die Einführung solcher Technologien erweist sich jedoch aus unterschiedlichen Gründen als komplex. Zum einen handelt es sich in beiden Fällen um eine langfristige Einführung, die

sich über mehrere Jahre oder Jahrzehnte erstreckt. Zum anderen müssen unterschiedliche Aspekte, Interessen und Sichtweisen bei der Etablierung berücksichtigt werden. So steht auf der einen Seite die Schaffung und der wirtschaftliche Betrieb einer möglichen Infrastruktur als wichtiger Punkt. Auf der anderen Seite müssen die Verkehrsteilnehmer (Fahrzeuge) mit der entsprechenden Technologie ausgerüstet werden.

Um ein solch komplexes Szenario untersuchen zu können, bieten sich Simulationen an. Durch eine Simulation können das dynamische Verhalten und die Interaktion von unterschiedlichen Teilnehmern des Verkehrssystems analysiert werden, ohne dabei wichtige Aspekte und Rahmenbedingungen auszublenden (Scherf 2010).

Im Bereich der Informationstechnik haben sich für Simulationen u. a. Multi-Agenten-Systeme etabliert, bei denen die wichtigsten Komponenten der Simulation durch Agenten abgebildet werden. Agenten besitzen dabei ein spezifiziertes und autonomes Verhalten, wodurch die dynamische Interaktion zwischen Teilnehmern in der Simulation realisiert werden kann.

Die Projektgruppe Mobility hat diesen Ansatz verfolgt, um ein Multi-Agenten-basiertes Simulationsframework für Verkehrssysteme zu entwickeln, welches sich auf unterschiedliche Szenarien verschiedener Verkehrsdomänen anwenden lässt.

1.2 Projektziele

Die grundsätzliche Voraussetzung für das Framework war die Verwendung eines Multi-Agenten-Systems als Basis der Simulation. Da sich während des Projekts herausgestellt hat, dass sich kein bestehendes Multi-Agenten-Framework für die Umsetzung eignet, ergab sich zunächst das Ziel, ein eigenes Multi-Agenten-Framework zu entwickeln, welches die von uns gestellten Anforderungen erfüllt.

Ein weiterer wichtiger Teil des Simulationsframeworks stellt das Verkehrsmodell dar, welches für die Anwendung auf Verkehrssysteme konzipiert wurde und als Abbildung einzelner Bestandteile von Verkehrssystemen entworfen wurde. Hierbei wurde ein abstrakter Ansatz verfolgt, um die Spezialisierung des Modells auf verschiedene Domänen zu ermöglichen.

Basierend auf dem Modell und unter Verwendung des Multi-Agenten-Frameworks sollte eine Erstellung einer allgemeinen Verkehrssimulation möglich sein. Für die bessere Nutzbarkeit sollte weiterhin eine grafische Oberfläche für die Eingabe von Simulationsdaten entwickelt werden.

Im Rahmen der Anwendung und der Evaluation sollte das Simulationsframework auf die spezifische Verkehrsdomäne der Schifffahrt angewendet werden. Hierbei wird die im vorherigen Abschnitt motivierte Nutzung von LNG als Treibstoff in einem Szenario für eine regionale Reederei angewendet.

Abschließend sollen die aus der Simulation erhaltenen Informationen zu Evaluationszwecken durch eine Reporting-Komponente in Form von Berichten dargestellt werden.

1.3 Aufbau des Dokuments

Im ersten Kapitel Grundlagen werden die wichtigsten Technologien und Frameworks behandelt, die für das Projekt relevant sind und für die Umsetzung des Simulationsframeworks verwendet wurden. Hierbei wird zunächst auf den Begriff der Multi-Agenten-Simulationen eingegangen, bevor eine Beschreibung der existierenden Multi-Agenten-Frameworks Multi Agent Simulator of Neighborhoods (MASON) und Java Agent Development (JADE) erfolgt. Weiterhin werden das Tool Eclipse und das Eclipse Modeling Framework (EMF) näher betrachtet.

Im Kapitel Anforderungsanalyse werden die funktionalen und nicht-funktionalen Anforderungen an das zu entwickelnde System erhoben und entsprechende Anwendungsfälle für das System erstellt und spezifiziert.

Darauf folgt der Entwurf des Frameworks, der auf der Anforderungsanalyse basiert. Hierbei werden die verfolgten Entwurfsziele erläutert, bevor die Systembeschreibung einen groben Überblick über die Gesamtarchitektur des Frameworks gibt. Die Konzepte der einzelnen Komponenten werden dann in den folgenden Abschnitten erklärt.

Ausgehend von dem Entwurf wird im Kapitel Implementierung die softwaretechnische Umsetzung des Frameworks beschrieben. Hier werden die konkreten Strukturen und Beziehungen innerhalb der Komponenten dargestellt und erläutert.

Im Kapitel Evaluation wird das Simulationsframework auf das konkrete Szenario aus der Schifffahrt angewendet. Zunächst erfolgen eine ausführliche Beschreibung des Szenarios sowie grundlegende Erklärungen bzgl. der Simulation im Bereich der Schifffahrt. Anschließend erfolgen die Beschreibung der Durchführung der Simulation und die darauf aufbauende Auswertung der Informationen aus der Simulation durch die Reporting-Komponente. Darauf folgt die Evaluation des gesamten Verkehrssimulationsframeworks, in der das Framework gegen die gestellten Anforderungen evaluiert wird.

Zum Abschluss dieses Berichts folgt eine Zusammenfassung, in der alle wichtigen Aspekte und Ergebnisse der Projektgruppe resümiert werden.

2 Grundlagen

Dieser Abschnitt beschreibt grundlegende Anwendungen, Frameworks und Technologien, die für die Entwicklung des Simulationsframeworks verwendet wurden.

2.1 Eclipse

Eclipse ist ein Open-Source Softwareentwicklungsprojekt, das 2001 von IBM ins Leben gerufen wurde und besonders in seiner Rolle als führendes Java Integrated Development Environment (IDE) bekannt ist (EclipseFoundation 2010). Das Projekt hat es sich zum Ziel gesetzt, eine integrierte Tool-Plattform bereitzustellen, mit der es möglich, ist IDEs zu entwickeln und zu erweitern. Hierzu wird eine Plug-In-Architektur verwendet, bei der die gesamte Funktionalität von Eclipse in einzelne Java Plug-Ins gekapselt ist. Die Plug-In-Verwaltung basiert dabei auf einer eigenen OSGi-Implementierung, wodurch Plug-Ins zur Laufzeit geladen werden können.

Neben der Entwicklung von IDEs kann die Plattform auch der Entwicklung einer Vielzahl von Anwendungsprogrammen dienen. Hierzu wurden die IDE spezifischen Elemente von den anwendungsunabhängigen Elementen getrennt. Die anwendungsunabhängigen Elemente bilden in ihrer Gesamtheit die Rich-Client-Platform (RCP), die es unter anderem ermöglicht Plug-Ins zu laden.

Die Entwicklung von Eclipse ist in mehrere Projekte unterteilt. Das Eclipse Projekt, das Tools-Projekt und das Technologie-Projekt stellen die drei Hauptprojekte dar. Das Tools Projekt beinhaltet dabei Werkzeuge wie bspw. das Eclipse Modeling Framework (EMF). Das Technologie Projekt umfasst Projekte, die sich in der Entwicklung befinden. Auf das Eclipse Projekt soll nun im Folgenden näher eingegangen werden (Sygo 2007, 5-21).

2.1.1 Eclipse Projekt

In Abbildung 1 ist der Kern des Eclipse Projektes, die Eclipse-Plattform dargestellt. Die Eclipse-Plattform ist die Grundlage für die Erstellung von eigenen Entwicklungsumgebungen und Anwendungsprogrammen. Sie kann durch das Hinzufügen von Plug-Ins um anwendungsspezifische Funktionalitäten erweitert werden. Zum Beispiel ermöglicht das Hinzufügen des Java Development Toolkits die Verwendung von Eclipse als Java-IDE. Ein weiteres Anwendungsgebiet wäre die Nutzung von Eclipse als Modellierungsumgebung durch das Hinzufügen des EMF Plug-Ins. Diese Art der Nutzung wird später noch weiter ausgeführt.

Charakteristisch für die Eclipse-Plattform ist, dass sie selbst aus mehreren Plug-Ins besteht, die von der Platform-Runtime geladen werden.

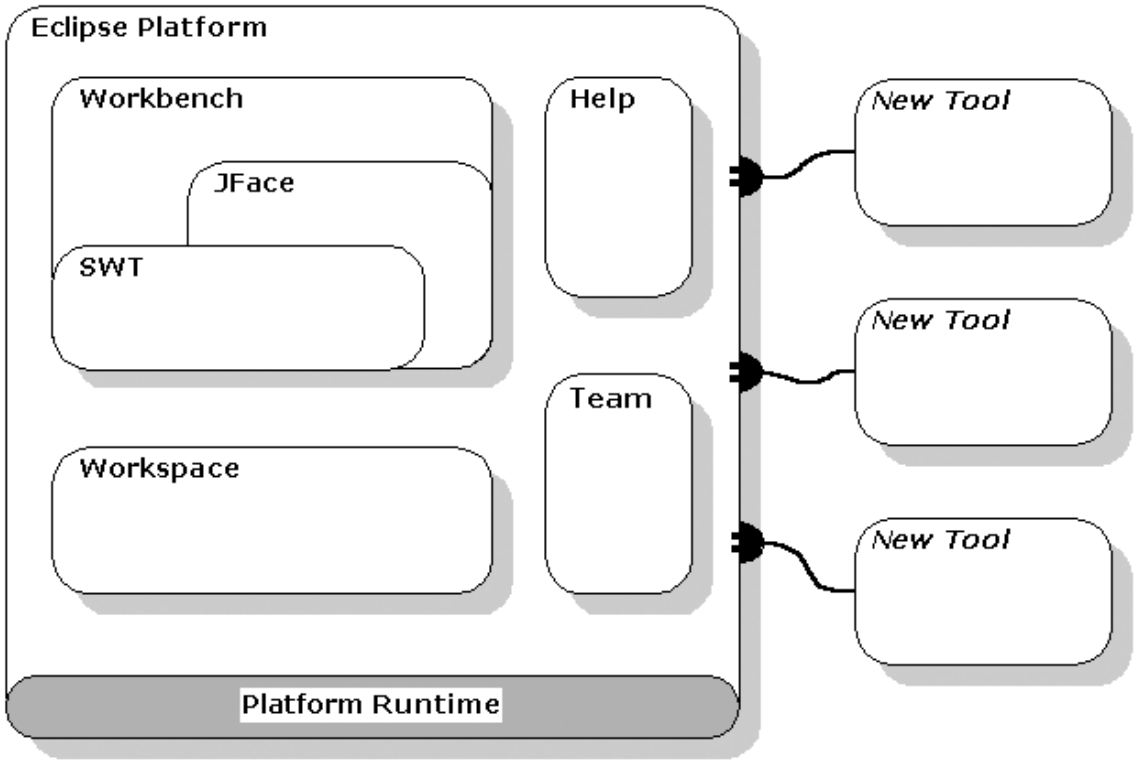

Abbildung 1: Eclipse Plattform

Im Folgenden werden die einzelnen Plug-Ins der Eclipse Plattform kurz beschrieben (Sygo 2007):

- Das Workbench-Plug-In enthält die grafisch Benutzeroberfläche. Es verwendet die Plug-Ins SWT und JFace, welche Bibliotheken für das Fenstersystem bereitstellen.
- Das Help-Plug-In enthält das Eclipse-Hilfesystem.
- Das Team-Plug-In stellt Funktionen zur Teamarbeit bei der Entwicklung bereit. Dazu gehört unter anderem ein integrierter CVS-Client.
- Das Workspace-Plug-In beinhaltet die Ressourcen-Verwaltung, d. h. es stellt die Funktionen zum Öffnen und Schließen, Überwachen von Dateiänderungen usw. bereit.
- Das Runtime-Plug-In enthält den Laufzeitkern der Eclipse-Plattform.

2.2 Multi-Agenten-Systeme / Simulation

Multi-Agenten-Systeme (MAS) gehören nach (Weiss 2000) zum Forschungsbereich der künstlichen Intelligenz. Die künstliche Intelligenz wird in mehreren gleichartigen oder unterschiedlichen Softwareagenten dargestellt.

Die Schlüsselfunktion der MAS ist die Beobachtung der Interaktion zwischen den Agenten, die ziel- oder aufgabenbasierte Koordination in kooperierenden und konkurrierenden Situationen durchführen.

Die Hauptcharakteristika bei Multi-Agenten-Systemen sind:

- Jeder Agent besitzt nur einen kleinen Teil der Informationen und ist im System in seinen Fähigkeiten eingeschränkt. Ein Agent ist Teil eines Kollektivs und muss mit anderen Agenten interagieren, um seine Aufgabe erfüllen zu können.
- Die Systemkontrolle ist verteilt. Kein Agent hat die volle Kontrolle über das System.
- Alle Daten sind dezentralisiert. Jeder Agent hat eine unterschiedliche Wissensbasis und nie einen kompletten Überblick. Weitere Informationen muss er durch Kooperation und Kommunikation mit anderen Agenten erhalten.
- Die Berechnung erfolgt asynchron. Es gibt keine übergeordnete Instanz, die Agenten koordiniert.

Aufbauend auf Multi-Agenten-Systemen gibt es Multi-Agenten-Simulationen, die von der Gesellschaft für Informatik e.V., wie folgt definiert werden:

> *Ein Multi-Agentenmodell ist ein Multi-Agentensystem in einer simulierten Umwelt und virtuellen Zeit, das ein reales Multi-Agentensystem nachbilden soll. Es besteht damit aus mehreren Agenten. Durch ihr Verhalten und ihre Interaktionen untereinander und mit ihrer Umwelt entstehen Muster und Verhalten auf einer höheren, aggregierten Ebene. Organisationsstrukturen können dabei ebenfalls Bestandteil eines Modells sein. Besonders interessant z. B. in sozialwissenschaftlichen oder biologischen Domänen sind Simulationen, bei denen die Entstehung von Organisationsstrukturen oder sozialen Netzwerken auf der Basis der Agenteninteraktionen erklärt bzw. untersucht werden. Nicht nur in solchen Anwendungsgebieten ist die Interaktion zwischen den Agenten ein zentraler Aspekt eines Multi-Agentenmodells.* (Gesellschaft für Informatik e.V. 2012)

Zur Erstellung einer Multi-Agenten-Simulation gibt es verschiedene Frameworks, die für die Umsetzung herangezogen werden können. Frameworks sind Programmiergerüste, die in der Softwareentwicklung Verwendung finden und dem Entwickler einen Rahmen bereitstellen, in dem er eine Anwendung entwickeln kann. Bekannte Frameworks für Multi-Agenten-Simulationen sind zum Beispiel JADE oder MASON, die in den folgenden Kapiteln betrachtet werden.

2.2.1 Multi Agent Simulator of Neighborhoods (or Networks)

Multi Agent Simulator of Neighborhoods (MASON) wurde an der George MASON University entwickelt und ist ein Werkzeug zur Erstellung von diskreten und stetigen Simulationen (George Mason University 2012).

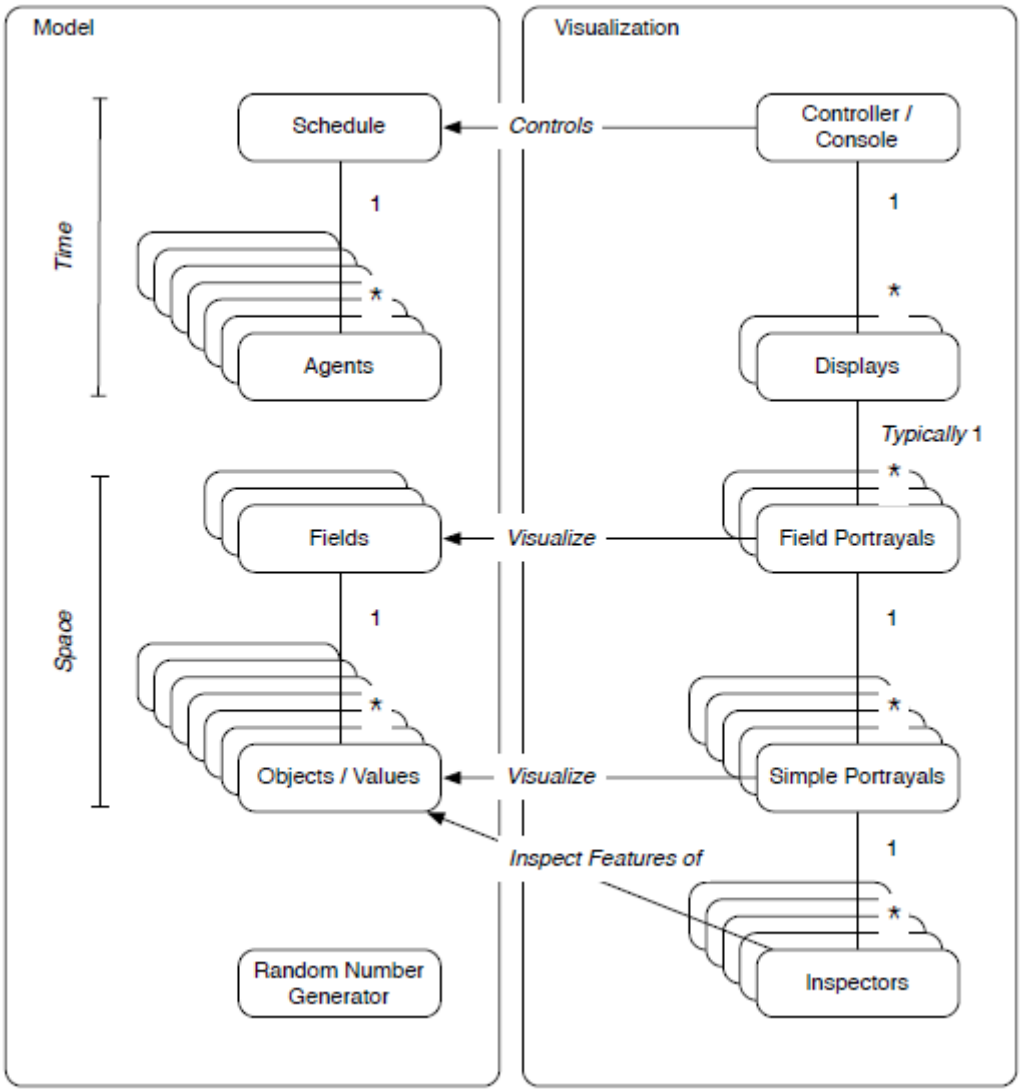

Abbildung 2: MASON Architektur (Department of Computer Science George Mason University 2011)

Die Architektur von MASON ist zweigeteilt in das Modell und die Visualisierung (siehe Abbildung 2). Diese Trennung ermöglicht die Durchführung der Simulation ohne eine Visualisierung.

Das Modell wird durch die Klasse *SimState,* welche die Hauptklasse der Simulation ist, dargestellt. In die SimState integriert ist der diskrete Event-Scheduler. Der Scheduler ist die Steuerungskomponente für die Organisation der zeitlichen Ausführung der Prozesse,

wodurch die Simulationszeit dargestellt wird. Ausführungen im Scheduler werden durch *Steps* im Interface *Steppable* bewerkstelligt.

Gleichzeitige Aktionen von Agenten werden in MASON nicht direkt unterstützt. Im Modell wird durch verankerte Objekte wie z. B*. network, field* etc. die Simulationsumwelt (*space*) dargestellt.

Die Visualisierung (console) ermöglicht eine Darstellung in 2D oder 3D. Durch eine Plug-In-Schnittstelle lassen sich weitere Visualisierungen einbetten. In der Visualisierung wird auch die grafische Oberfläche (GUI) generiert. Neben den Darstellungen werden durch Inspektoren Möglichkeiten zum Eingriff in die Simulation ermöglicht. Diese Trennung ermöglicht es dem Modell, je nach Bedarf, neue Visualisierungen zuzuweisen. Das Modell kann separat gespeichert und verwendet werden.

2.2.2 Java Agent Development

Java Agent Development (JADE) ist ein Java-basiertes, frei verfügbares Framework, das alle grundlegenden Anforderungen und eine Infrastruktur zur flexiblen Erstellung von Multi-Agenten-Systemen bereitstellt (Telecom Italia Lab 2012). Agenten stellen autonome Computersysteme dar, die ein definiertes Verhalten in einer Umwelt eigenständig nutzen können. Agenten in Jade agieren in Echtzeit. Die Kommunikation der Agenten erfolgt FIPA-konform und bildet den Kern von JADE. Es ist entsprechend der FIPA (Foundation for Intelligent Physical Agents) Standards entwickelt. Die FIPA (IEEE 2012) ist ein Standardisierungsgremium und in den Berufsverband Institute of Electrical and Electronics Engineers (IEEE) integriert. Das Ziel ist die Entwicklung einer Kommunikationsbasis für heterogene, interagierende Agentensysteme. Die Kommunikation und die Planung von Interaktionen zwischen Agenten erfolgt hierbei über Nachtrichten. Die Interaktion der Agenten wird durch built-in-Verhalten, sog. Behaviours gesteuert. Behaviours beinhalten die Logik der Agenten. (Wesche 2004)

Mit der Agentenplattform werden das Agenten Management System, der Directory Facilitator Agent und das Message Transport System gestartet. Dies sind systemseitige Komponenten für die Umsetzung und Durchführung der Simulation.

Das Agent Management System ist die Hauptinstanz in dem System. Sie stellt u. a. sicher, dass Agenten nicht die gleiche Bezeichnung haben. Agenten können durch sie erstellt oder beendet werden. Über den Directory Facilitator Agent erhalten bestehende Agenten Informationen, welcher Agent für welche Aufgabe im System zuständig ist. Dieser Agent ist vergleichbar mit einem Register. Das Message Transport System übernimmt FIPA-konform die Nachrichtenübermittlung zwischen den Agenten.

In JADE sind verschiedene Tools zur Plattformverwaltung und Anwendungsentwicklung implementiert. Diese Tools sind als Agenten realisiert, die im Auftrag des Anwenders in die Simulation eingreifen. Das sind:

- Remote Monitoring Agent
- Dummy Agent
- Introspector Agent
- Sniffer Agent

Der Remote Monitoring Agent ermöglicht dem Anwender die Steuerung und Überwachung der lokalen oder verteilten Abläufe auf der Agentenplattform. Dieser Agent besitzt eine grafische Oberfläche und kann die gesamte Plattform oder einzelne Agenten erstellen oder beenden.

Mit dem Dummy Agent wird der Nachrichtenaustausch zwischen den Agenten mittels der Agent Communication Language (ACL) Nachrichten getestet. Er besitzt die Möglichkeit einzelne Agent – Agent Kommunikationen zu Analysezwecken aufzuzeichnen.

Der Introspector Agent kontrolliert und überwacht die Agentenzustände und deren Nachrichtenverhalten. Des Weiteren kann dieser Agent die Geschwindigkeit der einzelnen Agenten reduzieren.

Der Sniffer Agent verfolgt die Message-Nutzung und kann diese zur Analyse speichern und/oder grafisch aufarbeiten.

2.3 Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) ist ein Teilprojekt des Eclipse Modeling Projects, das sich mit der modellgetriebenen Softwareentwicklung innerhalb von Eclipse beschäftigt (Steinberg, Budinsky und Merks 2008). Laut der Eclipse Homepage handelt es sich bei EMF um ein Modellierungsframework, das die Möglichkeit bietet, „effizienten, korrekten und leicht anpassbaren Java Code“ (EclipseFoundation 2010) aus strukturierten Datenmodellen zu generieren. Für die Beschreibung von Modellen beinhaltet EMF ein eigenes Metamodell. Neben dem Metamodell verwendet EMF zusätzlich noch ein Generatormodell zur Konfiguration der Codegenerierung.

2.3.1 Metamodell

Die zwei von EMF verwendeten Modelle werden in diesem Abschnitt kurz eingeführt. Zuerst wird dazu das zugrunde liegende Metamodell und im Anschluss daran das Generator-Modell erläutert.

2.3.2 Ecore-Modell

Unter einem Modell wird in EMF im Wesentlichen der Teil der Unified Modeling Language (UML) verstanden, der zur Beschreibung von Klassendiagrammen dient. EMF konzentriert sich lediglich auf die Modellierung von Klassen, da eine vollständige Abbildung der UML eine sehr anspruchsvolle Aufgabe darstellt (Steinberg, Budinsky und Merks 2008). Zur Darstellung von Modellen beinhaltet EMF mit dem Ecore-Modell ein eigenes Metamodell. Eine stark vereinfachte Sichtweise auf das Ecore-Metamodell wird durch Abbildung 2 gegeben und die einzelnen Elemente werden im Folgenden kurz erläutert.

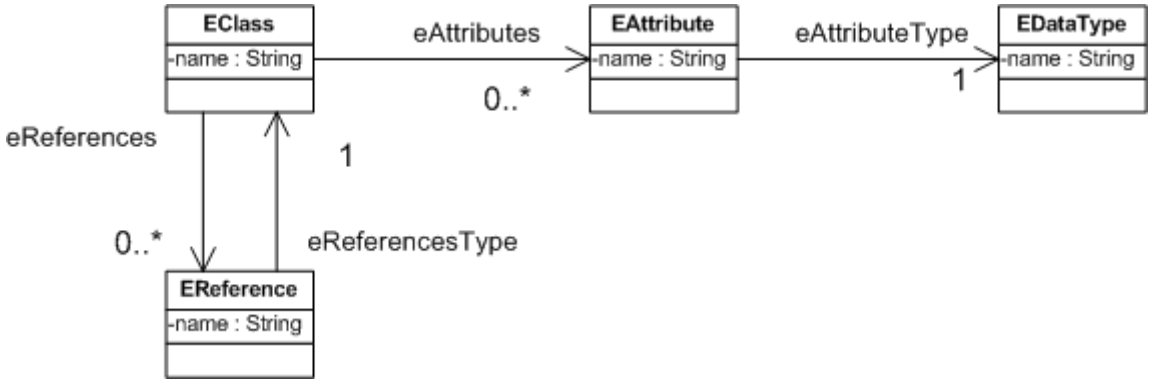

Abbildung 3: Ecore-Metamodell (Steinberg, Budinsky und Merks 2008)

Die EClass repräsentiert eine Klassendefinition. Sie besitzt einen Namen und kann Attribute und Referenzen beinhalten. EAttributes repräsentieren die modellierten Attribute. Diese haben einen Namen und einen Typ, welcher durch einen EDataType abgebildet wird. Es gibt EDataTypes für die gängigen Java-Datentypen wie zum Beispiel Integer oder Double. Es können aber auch benutzerdefinierte Datentypen erstellt werden. Die EReference stellt ein Ende einer Assoziation zwischen Klassen dar. Mit EMF ist es möglich, verschiedene Arten von Referenzen zu modellieren. Einige Beispiele dafür sind unidirektionale und bidirektionale Verbindungen sowie Kompositionen als spezielle Form der Assoziation (Vogel 2011).

Weiterführende Informationen zu den Referenzen sowie eine Darstellung des gesamten Ecore-Modells findet sich in unter anderem in (Steinberg, et al., 2009, Bacvanski, et al., 2005).

2.3.2.1 Generator-Modell

Zusätzlich zu dem bereits vorgestellten Ecore-Modell wird für die Codeerzeugung das Generator-Modell (GenModel) benötigt. Bei dem Generator-Modell handelt es sich um einen Wrapper für das Ecore-Modell, welches zusätzliche Konfigurationsmöglichkeiten für die Codegenerierung zur Verfügung stellt. So wird bspw. ein EAttribute aus dem Ecore-Modell auf ein GenFeature im GenModel abgebildet (Bacvanski und Graff 2005). Mit dem GenFeature ist es bspw. möglich, für jedes modellierte Attribut zu bestimmen, ob bei Änderungen Nachrichten, im Sinne des Observer-Patterns, an registrierte Listener verschickt werden sol-

len. Die Konfigurationsmöglichkeiten sind dabei nicht auf Attributebene beschränkt, sondern können auch auf Klassen- oder Paketebene zum Einsatz kommen. Hier existieren noch viele weitere Einstellungsmöglichkeiten, die der entsprechenden Literatur entnommen werden können. Zur Bearbeitung des Generator-Modells bietet EMF eine grafische Oberfläche an, in welcher die entsprechenden Einstellungen vorgenommen werden können.

2.3.3 Generierter Code

Der generierte Code kann vier verschiedene Projekte umfassen: das EMF.Model, EMF.Edit, EMF.Editor und EMF.Test. Im Folgenden werden diese einzelnen Projekte (ausgenommen EMF.Test) kurz erläutert, da diese für die Entwicklung des Verkehrssimulationsframeworks relevant sind.

2.3.3.1 EMF.Model

Das EMF.Model Projekt umfasst die Implementierung des erstellten Modells. Zu jeder modellierten Klasse werden ein Java-Interface und die dazugehörige Klasse angelegt, die das Interface implementiert. Dieses Vorgehen ermöglicht die Mehrfachvererbung in EMF. Klassen erben hierbei grundsätzlich vom EObject, welches das Äquivalent zum Java-Object darstellt. Mittels des EObject werden eine eigene Reflection-API sowie eine Persistence-API angeboten. EMF enthält ein leistungsstarkes Framework, um Modelle zu persistieren. Unter anderem besteht die Möglichkeit, Modelle als XML oder XML Metadata Interchange (XMI) zu persistieren. EMF ist nicht auf die zwei genannten Arten beschränkt, sondern unterstützt jegliche Art von Speicherung (Steinberg, Budinsky und Merks 2008). Ein Beispiel hierfür wäre die Anbindung von Datenbanken. Hierzu kann das Framework Teneo genutzt werden (Rummler und Voigt 2009). Jedes EObject ist außerdem ein Notifier im Sinne des Observer-Patterns und kann somit andere Objekte über Änderungen an Attributen informieren. Die Setter der generierten Klassen implementieren bereits die Notifikationsfunktionalität, wodurch modellierte Klassen als Observable benutzt werden können. Des Weiteren wird zur Objekterzeugung das Factory-Pattern eingesetzt, indem eine Factory-Klasse generiert wird.

2.3.3.2 EMF.Edit und EMF.Editor

Das EMF.Edit Projekt ist der Ausgangspunkt für die Entwicklung von eigenen Editoren für das erstellte Modell. Die generierten Klassen sollen aber noch an die eigenen Bedürfnisse angepasst werden. Der EMF.Edit Code ist dabei in den von der UI abhängigen und den von der UI unabhängigen Code aufgeteilt. Der UI abhängige Code nutzt SWT und JFace, um eine Oberfläche zur Bearbeitung des Modells zu erstellen. Das EMF.Editor Projekt stellt einen fertigen Eclipse-Editor zum Bearbeiten des zugrunde liegenden Modells bereit. Neben der Anpassung des Edit Projekts sollte auch dieser Code an die jeweiligen Bedürfnisse des

Nutzers angepasst werden, da er lediglich als Ausgangspunkt betrachtet werden sollte (Steinberg, Budinsky und Merks 2008).

2.3.4 Arbeitsweise

Im Folgenden wird eine vereinfachte Sichtweise auf den Prozess zur Arbeit mit EMF dargestellt. Das Arbeiten mit EMF folgt einem klar strukturierten und aufeinander aufbauenden Prozess. In einem ersten Schritt wird das Ecore-Modell erzeugt. Wie bereits erwähnt, bietet EMF hierzu mehrere Möglichkeiten. Auf Basis des Modells wird dann durch einen in EMF integrierten Generator das Generator-Modell erzeugt. Dieses dient als Grundlage für die EMF-Codegeneratoren und ermöglicht es die Code-Generierung zu konfigurieren. Mithilfe des Generator-Modells kann der zum Modell gehörende Java-Code erzeugt werden. Der durch EMF erzeugte Quellcode ist dafür ausgelegt, an die Bedürfnisse des Nutzers angepasst zu werden. Diese Änderungen können sowohl auf Ebene des erzeugten Codes als auch auf Modellebene vorgenommen werden. Bei Änderungen auf Modellebene werden, auch bei einer Neuerzeugung des Codes, selbst geschriebene Codeabschnitte nicht überschrieben. Die Unterscheidung zwischen generiertem und selbst erstelltem Code erfolgt bei EMF über Annotationen. Generierte Abschnitte erhalten die @generated-Annotation, anhand derer der Codegenerator entscheidet, ob Abschnitte überschrieben werden oder nicht. Selbst geschriebene Abschnitte werden vor der Überschreibung geschützt in dem die Annotation durch das Hinzufügen eines NOTs ungültig gemacht wird. Die grundlegende Arbeitsweise von EMF wird anhand der Abbildung 4 verdeutlicht.

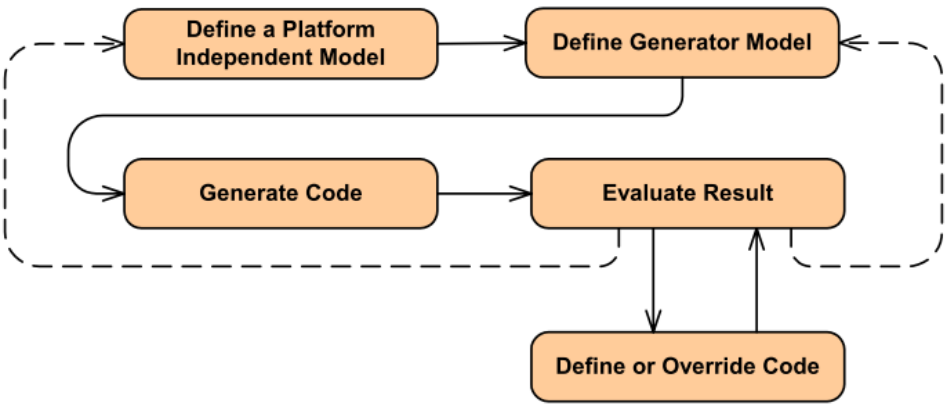

Abbildung 4: Arbeitsweise von EMF

3 Projektplanung

Dieses Kapitel beschreibt die Projektorganisation der Projektgruppe Mobility, welche im Masterstudium einen praktischen Anteil über zwei Semester darstellt. Im ersten Abschnitt wird die Vorgehensweise der Projektgruppe beschrieben. Der Abschnitt Projektphasen zeigt die terminliche Einordnung der einzelnen Phasen, die im Zeitraum des Projekts durchlaufen wurden. Abschließend werden im Abschnitt Projektbeteiligte die Teilnehmer vorgestellt und die Aufteilung in Kleingruppen dargestellt.

3.1 Vorgehensmodell

Für die Projektgruppe Mobility kommen sowohl klassische als auch agile Vorgehensmodelle infrage. Als klassische Vorgehensmodelle existiert u. a. das V-Modell, welches das Wasserfallmodell um eine Berücksichtigung der Qualitätssicherung, durch kontinuierliche Verifikation und Validierung erweitert und somit ein paralleles Entwickeln und Testen vorsieht.

Agile Vorgehensmodelle sorgen für ein dynamisches und flexibles Management. Vorgeschlagen wurde dafür u. a. eXtreme Programming, welches vor allem durch die Definition von Testfällen vor der Implementierung, Programmierung in Paaren, sowie dem Festhalten der Anforderungen in Form von User-Stories geprägt ist. Scrum ist ebenfalls ein iteratives Vorgehensmodell. Durch kurze Zyklen, sog. Sprints und tägliche Treffen stellt es ein äußerst effizientes, allerdings auch recht aufwendiges Vorgehen dar. Die unterschiedlichen Vorgehensmodelle haben sowohl Vor- als auch Nachteile.

Die gewählte Vorgehensweise soll in kleinen bis mittelgroßen Projekten gut anwendbar sein und vor allem die Arbeit in Kleingruppen durch das sog. Pair-Programming sowie die klar definierten Zyklen gut unterstützen. Diese Anforderungen finden sich verstärkt innerhalb des agilen Projektmanagements wieder. Ohne eine konkrete Form der agilen Methoden, wie bspw. eXtreme Programming oder Scrum, anzuwenden, orientiert sich die Vorgehensweise der Projektgruppe an einigen Grundsätzen der Agilität. Dabei wurde das Projekt wie folgt organisiert:

Wöchentlich werden zwei Treffen abgehalten, bei denen alle Projektmitglieder anwesend sind. Das erste Treffen dient der Rücksprache mit den Betreuern des Projektes. Alle Koordinierungen bzgl. des Projektes werden hier besprochen. Das Ziel hierbei ist der Informationsaustausch sowie die Bewältigung und Lösung von Problemen, die während der Projektphasen entstehen. Außerdem werden die nächsten Schritte werden besprochen und Arbeitspa-

kete verteilt. Das Treffen stellt somit die Schnittstelle zwischen der universitären Betreuung und der Projektgruppe dar.

Das zweite Treffen ist ein Arbeitstreffen. Die Kleingruppen von drei bis vier Personen konnten hier ihre Arbeitspakete bearbeiten. Durch die Nähe zu den anderen Gruppen konnte eine gute Kommunikation entwickelt werden, die kurze Lösungswege bei Fragen und Problemen bot.

3.2 Projektphasen

Grundsätzlich gliedert sich der Zeitraum der Projektgruppe in fünf Phasen. Mit dem Start der Projektgruppe erfolgte zunächst eine Seminarphase, in der relevante Thematiken und Technologien, die für das Projekt wichtig sind, in Form von Präsentationen und schriftlichen Ausarbeitungen aufbereitet wurden. In der Ideenfindungsphase stand die Erarbeitung eines geeigneten Lösungsansatzes für das gestellte Problem sowie eine Prüfung der technischen Machbarkeit im Vordergrund, was als Ausgangspunkt für die Anforderungsanalyse der dritten Projektphase diente. An dieser Stelle wurden die zu erfüllenden Anforderungen an das Verkehrssimulationsframework herausgearbeitet und aufeinander abgestimmt. Das Ergebnis war hierbei eine Leistungsbeschreibung, in der alle wichtigen Aspekte und Entscheidungen der Anforderungsanalyse festgehalten wurden. Die zeitliche umfangreichste Phase stellt die Entwurfs- und Implementierungsphase dar, welche sich vom September 2011 bis zum Februar 2012 erstreckte. In dieser Phase stand zunächst der Entwurf der einzelnen Komponenten des Frameworks an, bevor mit der Implementierung des Frameworks begonnen wurde.

In der Endphase des Projekts wurden zum einen Optimierungen an dem Framework (insbesondere im Rahmen der Evaluation) vorgenommen sowie abschließende Dokumentationsaufgaben (Dokumentation des Sourcecodes) durchgeführt. Weiterhin wurde in dieser Phase der Projektabschlussbericht fertiggestellt und die abschließende Präsentation der Projektgruppe vorbereitet. In der Abbildung 5 ist die zeitliche Abfolge der einzelnen Projektphasen in Form eines Gantt-Diagramms dargestellt.

ID	Phase	2011									2012		
		Apr	Mai	Jun	Jul	Aug	Sep	Okt	Nov	Dez	Jan	Feb	Mrz
1	Seminarphase												
2	Ideenfindungsphase												
3	Anforderungsphase												
4	Entwurfs- und Implementierungsphase												
5	Endphase												

Abbildung 5: Projektphasenplan

3.3 Projektbeteiligte

Die Projektgruppe setzt sich aus zwölf Studenten des Studiengangs Master of Science Wirtschaftsinformatik der Carl von Ossietzky Universität Oldenburg zusammen. Aufgrund der verhältnismäßig hohen Teilnehmerzahl der Projektgruppe war es an dieser Stelle wichtig, die Projektgruppe in mehrere Teilgruppen mit unterschiedlichen Aufgabenschwerpunkten zu untergliedern.

Tabelle 1: Mitglieder der Projektgruppe

	Arne Laverentz		Bastian Wagenfeld
	Christian Denker		Florian Steinmeyer

Christian van Göns

Kolja Blohm

Marcel Wilkens

Mathias Ringe

Hong-Son Dang

Tammo Buss

Sven Vorlauf

Stefan Kramer

Zu Beginn der Ideenphase entstanden drei Arbeitsbereiche (siehe Abbildung 6). Die Datenbeschaffung hatte dabei die Aufgabe, alle Daten und Informationen, die für Simulationen im Bereich der Verkehrssysteme (insbesondere der Schifffahrt) relevant sind, zu beschaffen und aufzubereiten, um somit eine möglichst realitätsnahe Abbildung der Aspekte innerhalb der Simulation zu erzielen. Im Rahmen der Modellierung wurde ein abstraktes Modell eines Verkehrssystems konzipiert, das als Ausgangspunkt für den Entwurf des Frameworks diente. Weiterhin wurde parallel dazu in der Teilgruppe Architektur der Aufbau des gesamten Systems spezifiziert, aus dem die Aufteilung in mehrere Teilkomponenten folgte.

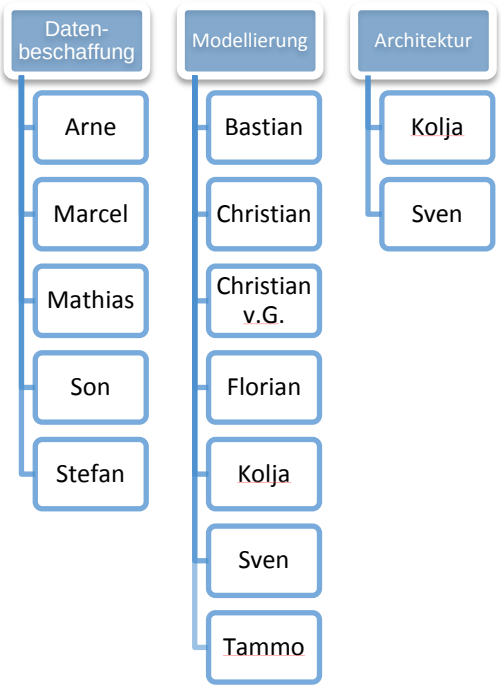

Abbildung 6: Arbeitsgruppen

Bei der Erarbeitung der Anforderung wurde ersichtlich, dass die Aufteilung neu strukturiert werden muss. Aus der Modellierung und Architektur entwickelten sich die Arbeitsgruppen Reporting, Datenmodell, Simulation und Editor. Diese Struktur hat bis zum Ende der Projektgruppe bestand.

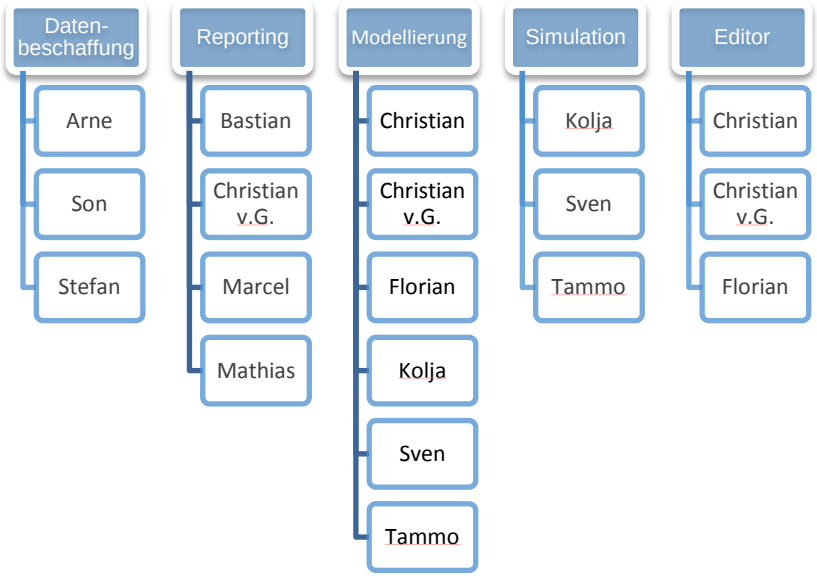

Abbildung 7: Neu strukturierte Arbeitsgruppen

4 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an das zu entwickelnde System beschrieben. Aus der Vorgabe, ein agentenbasiertes Framework zu entwickeln, folgen zunächst die gestellten Anforderungen an ein Multi-Agenten-Framework. Hinsichtlich der Erfüllung dieser Anforderungen werden JADE und MASON analysiert. Da weder JADE noch MASON alle gestellten Anforderungen hinreichend erfüllen, erfolgt hiernach eine Aufstellung von Anforderungen an ein eigenes Multi-Agenten-Framework.

Zudem soll die Verkehrssimulation in ein Datenmodel und in eine Agentensimulation aufgeteilt werden. Das Datenmodell dient der Abbildung der Verkehrskomponenten. Die Agentensimulation beinhaltet Agenten, die auf dem Modell arbeiten und die Logik der Simulation bereitstellen.

Die nicht-funktionalen Anforderungen werden an das gesamte zu entwickelnde System gestellt. Abschließend werden im letzten Abschnitt Anwendungsfälle für die Verwendung des Frameworks entworfen und spezifiziert.

4.1 Anforderungen und Analyse von Multi-Agenten-Frameworks

Dieser Abschnitt beschreibt die wichtigsten Kriterien bei der Auswahl eines Agenten-Frameworks, das als Basis für die Verkehrssimulation genutzt wird:

- **Eigene Simulationszeit**: Agenten sollen eine Aktion zu einer bestimmten Simulationszeit ausführen können.
- **Interaktion/Reaktion**: Agenten sollen mittels Nachrichten miteinander kommunizieren können und auf Anfragen anderer Agenten reagieren können.
- **Gleichzeitige Aktionen**: Agenten sollen zu einem Simulationszeitpunkt mehrere Aktionen ausführen können.

Im Folgenden werden die in Abschnitt 2.2 beschriebenen Frameworks JADE und MASON auf die Erfüllung der drei Hauptanforderungen untersucht. MASON wurde der Projektgruppe zum Beginn durch die Projektleitung vorgeschlagen und wurde bereits in einem ähnlichen Projekt verwendet. Alternativ dazu wurde JADE mit in die Analyse aufgenommen, da bereits Vorkenntnisse vorhanden waren und es grundsätzlich als geeignet hinsichtlich der Anforderungen betrachtet wurde.

Tabelle 2: Analyse von MASON und JADE

Kriterium	**MASON**	**JADE**
Eigene Simulationszeit	Scheduler	Keine Simulationszeit, nur Echtzeit
Interaktion/Reaktionen	SimState und Methodenaufrufe	Nachrichten und Protokolle (FIPA)
Gleichzeitige Aktionen	Nicht direkt unterstützt	Realisiert durch Behaviours

In der obigen Tabelle ist der Vergleich der beiden Frameworks bzgl. der genannten Kriterien dargestellt.[1] Hinsichtlich der Simulationszeit erfüllt MASON das Kriterium durch den integrierten Scheduler. In JADE erfolgt die Simulation hingegen nur in Echtzeit und es wird keine Simulationszeit unterstützt. Die Interaktion bzw. Reaktion zwischen Agenten erfolgt in JADE durch FIPA-konforme Nachrichten und Protokolle, wodurch dieses Kriterium erfüllt ist. In MASON ist dies durch SimStates und Methodenaufrufe umgesetzt – es werden keine Nachrichten verwendet. Gleichzeitige Aktionen von Agenten werden in MASON nicht direkt unterstützt. In JADE hingegen wird hierzu das Konzept der Behaviours verwendet.

Zusammenfassend lässt sich daraus schließen, dass MASON insbesondere durch den Scheduler gute Ansätze aufweist, andererseits nicht alle Kriterien (insbesondere die gleichzeitigen Aktionen) erfüllt. Im Gegensatz dazu erfüllt JADE alle Kriterien bis auf die eigene Simulationszeit.

In Anbetracht dieses Sachverhalts unterstützt keines der beiden Frameworks die genannten Kriterien. Aufgrund dessen ist eine Eigenentwicklung nötig, die es ermöglicht, alle drei Anforderungen zu erfüllen. Diese Eigenentwicklung beinhaltet einen an das MASON-Framework angelehnten Scheduler und erweitert dieses Konzept um Nachrichten und Behaviours, wie es bei JADE der Fall ist. Da es sowohl Funktionalitäten von JADE als auch von MASON aufweist, heißt das erstellte Multi-Agenten-Framework JASON.

4.2 Funktionale Anforderungen

In diesem Abschnitt werden die funktionalen Anforderungen an das Verkehrssimulations-Framework sowie an JASON erhoben. Angelehnt an die Aufteilung im einleitenden Teil, werden die Anforderungen an JASON, das Verkehrsmodell und die darauf basierende Verkehrssimulation separat betrachtet.

[1] Spezifische Begriffe bzgl. der Frameworks wie z. B. Scheduler oder SimStates sind im Kapitel Grundlagen erläutert.

4.2.1 Funktionale Anforderungen JASON

Im Folgenden sind die Anforderungen an JASON genannt, die sich aus den Funktionalitäten von JADE und MASON sowie eigenen Anforderungen ergeben.

1. Das Framework soll die Möglichkeit bieten Multi-Agenten-Simulationen zu entwickeln.
   1.1. Agenten sollen in der Simulation agieren können.
      1.1.1. Das Framework soll gleichzeitige Aktionen unterstützen.
   1.2. Agenten sollen Dienste anbieten können.
      1.2.1. Die angebotenen Dienste sollen in der Simulation registriert werden können.
   1.3. Agenten sollen Dienste anderer Agenten nutzen können.
      1.3.1. Ein Agent soll angebotene Dienste aus der Simulation erfragen können.
2. Das Framework soll eine eigene Simulationszeit enthalten.
   2.1. Agenten sollen Aktionen zu bestimmten Simulationszeiten einplanen können.
3. Agenten einer Simulation sollen miteinander interagieren können.
   3.1. Agenten sollen über ein Nachrichtensystem kommunizieren können.
      3.1.1. Agenten sollen Nachrichten empfangen können.
      3.1.2. Agenten sollen Nachrichten senden können.
   3.2. Das Nachrichtensystem soll alle Agenten kennen.

4.2.2 Funktionale Anforderungen Verkehrsmodell

Das Verkehrsmodell soll ein Modell zur Abbildung von Verkehrssystemen sein.

1. Das Modell soll eine Verkehrsinfrastruktur abbilden.
   1.1. Die Verkehrsinfrastruktur soll ein Streckennetz abbilden.
      1.1.1. Strecken können aus Teilstrecken bestehen.
      1.1.2. Das Streckennetz soll Infrastrukturstandorte verbinden können.
   1.2. Die Verkehrsinfrastruktur soll aus Infrastrukturpunkten bestehen.
      1.2.1. Infrastrukturpunkte sollen Tankstationen beinhalten können.
         1.2.1.1. Eine Tankstation soll Tanks haben.
      1.2.2. Infrastrukturpunkte können Verladestationen beinhalten können.
         1.2.2.1. Eine Verladestation kann einen Stauraum haben können.
2. Es sollen verschiedene Ladungen betrachtet werden können
   2.1. Ladungen sollen ein Gewicht haben.
   2.2. Ladungen sollen ein Volumen haben.
3. Es sollen verschiedene Treibstoffe betrachtet werden können.
   3.1. Ein Treibstoff soll unterschiedliche Merkmale haben.
      3.1.1. Ein Treibstoff soll einen Kohlenstoffdioxidanteil haben.
      3.1.2. Ein Treibstoff soll einen Schwefelanteil haben.
      3.1.3. Ein Treibstoff soll einen Stickstoffanteil haben.

3.1.4. Ein Treibstoff soll einen Brennwert haben.

3.1.5. Ein Treibstoff soll ein Gewicht haben.

3.1.6. Ein Treibstoff soll ein Volumen haben.

4. Das Modell soll Fahrzeuge abbilden können.

4.1. Ein Fahrzeug kann einen Stauraum haben können.

4.1.1. Ein Fahrzeug kann im Stauraum Ladung haben.

4.2. Ein Fahrzeug soll einen Tank haben.

4.3. Ein Fahrzeug soll Motoren haben.

4.3.1. Ein Motor soll Leistung für die Fortbewegung bereitstellen können.

4.3.2. Ein Motor soll für einen bestimmten Treibstoff ausgelegt sein.

4.3.3. Ein Motor soll Treibstoff verbrauchen können.

4.3.4. Ein Motor soll Emissionen verursachen können.

4.4. Der Treibstoff soll durch Tankstationen zur Verfügung gestellt werden.

4.5. Ein Fahrzeug soll seine eigenen Leistungsmerkmale kennen.

4.5.1. Ein Fahrzeug soll eine aktuelle Geschwindigkeit haben.

4.5.2. Ein Fahrzeug soll die aktuelle Leistung seiner Motoren kennen.

4.5.3. Ein Fahrzeug soll die maximale Leistung seiner Motoren kennen.

4.6. Ein Fahrzeug soll einen Fahrplan haben können.

4.6.1. Ein Fahrplan soll aus Einträgen bestehen.

4.6.1.1. Ein Eintrag soll einen Ort haben.

4.6.1.2. Ein Eintrag soll einen Zeitpunkt haben.

4.6.1.3. Ein Eintrag soll auszuführende Aktionen haben.

4.6.1.4. Ein Eintrag soll wiederholt ausführbar sein.

4.7. Es sollen verschiedene Treibstoffe betrachtet werden können.

4.7.1. Ein Treibstoff soll unterschiedliche Merkmale haben.

4.7.1.1. Ein Treibstoff soll einen Kohlenstoffdioxidanteil haben.

4.7.1.2. Ein Treibstoff soll einen Schwefelanteil haben.

4.7.1.3. Ein Treibstoff soll einen Stickstoffanteil haben.

4.2.3 Funktionale Anforderungen Verkehrssimulation

Es soll ein Framework zur Simulation von Verkehrssystemen entwickelt werden.

1. Die Simulation soll auf dem Verkehrsmodell basieren.
2. Das Framework soll als Erweiterung von JASON implementiert werden.

2.1. Agenten sollen Standorte im Verkehrssystem besitzen können.

2.1.1. Die Standorte sollen in der Simulation registriert werden können.

2.2. Agenten sollen Standorte anderer Agenten ermitteln können.

2.2.1. Ein Agent soll Standorte aus der Simulation erfragen können.

2.3. Infrastrukturpunkte sollen von Infrastrukturagenten verwaltet werden.
2.3.1. Infrastrukturagenten sollen Treibstoffe und Frachtgut be- und entladen können.
2.3.2. Infrastrukturagenten sollen Ladevorgänge verhandeln können.
2.3.2.1. Infrastrukturagenten sollen die Verladegeschwindigkeit verhandeln können.
2.3.2.2. Infrastrukturagenten sollen die Menge der Ladung verhandeln können.
2.4. Fahrzeuge sollen von Fahrzeugagenten gesteuert werden.
2.4.1. Fahrzeugagenten sollen ein Fahrzeug bewegen können.
2.4.1.1. Fahrzeugagenten sollten die kürzeste Route des Fahrzeugs planen können.
2.4.1.2. Fahrzeugagenten sollen die benötigte Motorleistung für eine Route vorgeben können.
2.4.1.3. Fahrzeugagenten sollen eine Route abfahren können.
2.4.1.4. Fahrzeugagenten sollen während der Fahrt Treibstoff verbrauchen können.
2.4.2. Fahrzeugagenten sollen einen Fahrplan abarbeiten können.
2.4.3. Fahrzeugagenten sollen Treibstoff tanken können.
2.4.4. Fahrzeugagenten sollen Frachtgut be- und entladen können.

4.2.4 Funktionale Anforderungen Editor

Es soll ein Editor ins Framework integriert werden, mit dem Simulationen angelegt und gestartet werden können.

1. Der Editor soll in der Lage sein, Daten zu speichern.
   1.1. Daten sollen in einem XML-Format gespeichert werden.
2. Der Editor soll in der Lage seine Daten zu laden.
   2.1. Der Editor soll zuvor gespeicherte Daten laden können.
3. Der Editor soll geladene Daten verarbeiten können.
   3.1. Die in den Editor geladenen Daten sollen modifizierbar sein.
   3.2. Die modifizierten Daten sollen wieder gespeichert werden können.
4. Der Editor soll eine Simulation starten können.
   4.1. Die in den Editor eingegebenen Daten sollen dazu verwendet werden können eine Simulation zu starten.
      4.1.1. Die eingegebenen Daten sollen vom Editor an die Simulation übergeben werden.
   4.2. Der Editor soll die gleiche Simulation mehrmals ausführen können.

4.3 Nicht-funktionale Anforderungen

In diesem Abschnitt werden die nicht-funktionalen Anforderungen beschrieben, die an das Framework gestellt werden.

1. Das Framework soll in Java implementiert werden.
   1.1. Die Implementierung soll in englischer Sprache erfolgen.
2. Das Verkehrsmodell soll mit EMF realisiert werden.
   2.1. Die Namensgebung des Verkehrsmodells soll in englischer Sprache erfolgen.
3. Zu allen Klassen sollen Kommentare existieren.
   3.1. Die Kommentare sollen als JavaDoc vorliegen.
   3.2. Die Kommentare sollen in englischer Sprache vorliegen.
4. Der Editor muss zuverlässig und stabil laufen.
5. Die GUI des Editors soll übersichtlich sein.
   5.1. Die GUI soll eine einfache Eingabe der Daten gewährleisten.
   5.2. Die GUI solle intuitiv bedienbar sein.
6. Für den Editor sollte genügend Hauptspeicher zur Verfügung stehen.

4.4 Anwendungsfälle

Dieser Abschnitt behandelt die Anwendungsfälle, die durch Interaktion mit dem Framework auftreten. Diese stellen keine vollständige Sicht auf die Funktionalität des zu entwerfenden Framework dar, sondern zeigen grundlegende Anwendungsszenarien aus Sicht des Akteurs, einem Benutzer des Frameworks. Ein Akteur ist dabei ein im Umgang mit dem Framework vertrauter Experte.

Eine schematische Ansicht der Anwendungsfälle in Form eines UML-Anwendungsfall-diagramms wird in Abbildung 8 illustriert.

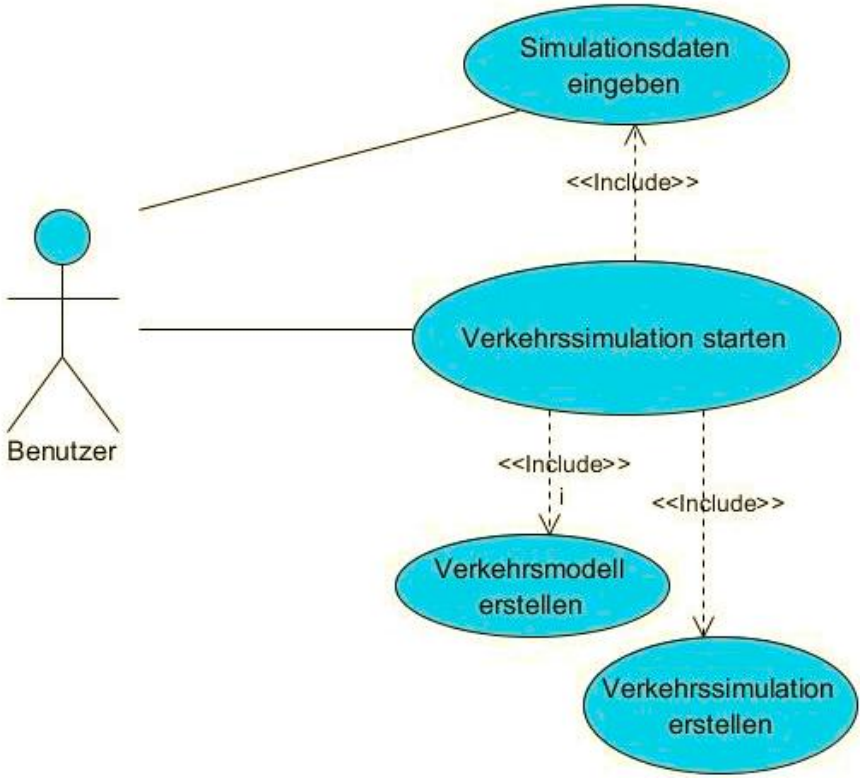

Abbildung 8: Anwendungsfälle des Systems

Nachfolgend werden die in der Abbildung 8 dargestellten Anwendungsfälle im Einzelnen beschrieben. Der Übersicht halber erfolgt die Beschreibung in tabellarischer Form. Der Tabellenaufbau gestaltet sich wie folgt:

- **Name:** Unter diesem Punkt ist der Name des betrachteten Anwendungsfalls angegeben.
- **Akteure:** An dieser Stelle werden die Ressourcen angegeben, die an die Ausführung des betreffenden Anwendungsfalls beteiligt sind. In diesem Fall ist der Akteur des zu entwerfenden Systems, ein im Umgang mit dem System vertrauter Experte.
- **Beschreibung:** In dieser Zeile erfolgt eine Grobbeschreibung des Anwendungsfalls.
- **Standardablauf:** An diesem Punkt wird ausführlich beschrieben, was bei der Durchführung des Anwendungsfalls passiert.
- **Anfangsbedingungen:** Dieser Punkt gibt an, welche Bedingungen gelten müssen, damit der Anwendungsfall ausgeführt werden kann.
- **Abschlussbedingungen:** An dieser Stelle wird angegeben, welche Bedingungen nach der Durchführung des Anwendungsfalls gelten müssen.
- **Qualitätsanforderungen:** Aus diesem Eintrag in der Tabelle geht hervor, welche Qualitätsanforderungen an die Funktionalitäten gestellt werden.

Die Spezifikationen der Anwendungsfälle sehen wie folgt aus:

Tabelle 3: Erstellung eines domänenspezifischen Datenmodells

Name	**AF1 Erstellung eines domänenspezifischen Datenmodells**
Akteure	Benutzer
Beschreibung	Zur Abbildung domänenspezifischer Objekte (wie Schiffe, Tankstationen, etc.) für die Verkehrssimulation, muss der Benutzer ein konkretes Datenmodell erstellen. Dazu wird das abstrakte Verkehrsmodell des Frameworks durch ein konkretes domänenspezifisches Datenmodell spezialisiert.
Standardablauf	1. Der Benutzer erweitert das abstrakte Verkehrsmodell des Frameworks um konkrete domänenspezifische Attribute. a. Die Erweiterung kann codebasiert oder modellgetrieben erfolgen. 2. Bei einer modellgetriebenen Erweiterung verwendet der Benutzer EMF. 3. Über eine grafische Benutzeroberfläche, die EMF zur Verfügung stellt, führt der Benutzer die notwendigen Erweiterungen durch.
Anfangsbedingungen	Keine
Abschlussbedingungen	Es liegt ein konkretes domänenspezifisches Verkehrsmodell vor.
Qualitätsanforderungen	Es müssen alle Objekte und Attribute für den angestrebten Simulationszweck abgebildet werden können.

Tabelle 4: Erstellung einer domänenspezifischen Verkehrssimulation

Name	**AF2 Erstellung einer domänenspezifischen Verkehrssimulation**
Akteure	Benutzer
Beschreibung	Um Simulationen durchführen zu können, muss der Benutzer eine konkrete domänenspezifische Verkehrssimulation erstellen. Die Verkehrssimulation baut dabei auf dem konkreten domänenspezifischen Datenmodell auf.
Standardablauf	1. Der Benutzer erweitert die abstrakte Verkehrssimulation um konkrete domänenspezifische Funktionalitäten. 2. Die Erweiterung erfolgt programmiertechnisch: Der Benutzer

	erweitert die Basis-Funktionalitäten des Frameworks um konkrete domänenspezifische Funktionalitäten.
Anfangsbedingungen	Es wurde ein konkretes domänenspezifisches Datenmodell erstellt.
Abschlussbedingungen	Es liegt ein konkretes domänenspezifisches Verkehrsmodell vor.
Qualitätsanforderungen	Es müssen alle erforderlichen Funktionalitäten für einen bestimmten Simulationszweck bereitstehen.

Tabelle 5: Eingabe der Simulationsdaten in das Simulationssystem

Name	**AF3 Eingabe der Simulationsdaten in das Simulationssystem**
Akteure	Benutzer
Beschreibung	Das domänenspezifische Datenmodell und die Verkehrssimulation bilden das Simulationssystem. Für die Simulation werden Eingabedaten benötigt. Diese werden von dem Benutzer über einen auf EMF basierenden Editor in das Simulationssystem eingegeben.
Standardablauf	1. Der Benutzer sammelt Daten, die für die Simulation notwendig sind. 2. Der Benutzer überführt die Daten in ein von der Simulation nutzbares Format. 3. Über den Editor gibt der Benutzer die Daten in das System ein. Der Editor validiert die eingegebenen Daten.
Anfangsbedingungen	• Es liegen ein domänenspezifisches Datenmodell und eine domänenspezifische Verkehrssimulation vor, die ausführbar sind. • Die für die Simulation erforderlichen Daten können ermittelt werden.
Abschlussbedingungen	• Die Daten liegen in einem vom System lesbaren Format vor.
Qualitätsanforderungen	• Die Daten sollen realitätsnah sein, um nutzbare Ergebnisse zu erhalten.

Tabelle 6: Simulation starten

Name	**AF4 Simulation starten**
Akteure	Benutzer

Beschreibung	Nachdem die erforderlichen Simulationsdaten eingegeben sind, wird die Simulation gestartet. Das Starten erfolgt über den Editor.
Standardablauf	1. Der Benutzer startet die Simulation über den Editor. 2. Nach einer zuvor definierten Simulationszeit wird die Simulation beendet. 3. Die Simulation liefert Ergebnisse. Der Benutzer kann diese Ergebnisse als Unterstützung für seine Entscheidung verwenden.
Anfangsbedingungen	• Es liegt ein lauffähiges Simulationssystem vor. • Die Daten liegen in einem vom System lesbaren Format vor.
Abschlussbedingungen	• Es liegen Simulationsergebnisse vor.
Qualitätsanforderungen	• Die Ergebnisse sollen für einen bestimmten Zweck nutzbar sein.

5 Entwurf

Nachdem in dem vorherigen Kapitel die funktionalen und nicht-funktionalen Anforderungen formuliert wurden, widmet sich dieses Kapitel den Entwurfszielen und den grundlegenden Entwurfsgedanken des Verkehrssimulationsframeworks. Das Kapitel beginnt mit der Beschreibung der allgemeinen Entwurfsziele des Frameworks. Diese Ziele dienen als Referenz bei der Entwurfserstellung. Im Anschluss daran wird eine Übersicht über die Teilkomponenten des Frameworks gegeben. Dabei werden die grundlegenden Bausteine des Frameworks und ihre Zusammenhänge dargestellt. Abschließend folgen Abschnitte, in denen die Konzepte der einzelnen Teilkomponenten des Frameworks detailliert erläutert werden.

5.1 Entwurfsziele

Ziel des Projektes ist es, ein Framework zu entwickeln, welches die Möglichkeit bietet, Multi-Agenten-Simulationen zu entwickeln. Im Folgenden sind wichtige Aspekte (in Anlehnung an (Leitner 2007)) beschrieben und erläutert, die für den Entwurf und der darauf folgenden Implementierung eine Rolle spielen.

5.1.1 Modularität

Das Framework soll modular aufgebaut sein. Es soll eine einfache Erweiterung von zusätzlichen Funktionalitäten ermöglichen. Dazu gehören zum einen eine unkomplizierte Einbindung von neuer Logik und zum anderen die Erweiterung vom bestehenden allgemeinen Verkehrssimulationsframework zur domänenspezifischen Verkehrssimulation. Die einfache Erweiterbarkeit soll sicherstellen, dass sich die Funktionalität des Frameworks an zukünftige Anforderungen anpassen lassen kann.

5.1.2 Wiederverwendbarkeit

Da es das Ziel des Frameworks ist, als Grundlage für eine realitätsnahe Simulation zu dienen, ist es wichtig darauf zu achten, dass es leicht zu benutzen ist und keine genauen Kenntnisse über die Implementierung vorausgesetzt werden. Auch eine flexible Nutzung des Frameworks soll möglich sein. Bspw. soll es in eine Anwendung eingebunden werden können, welche nur die Funktionalitäten zur Ermittlung der Reisezeiten der Verkehrsteilnehmer auf einer Verkehrsinfrastruktur benutzt. In diesem Fall könnte z. B. auf festgelegte Funktionen zum Tanken der Verkehrsteilnehmer verzichtet werden. Da die Funktionalität einer breiten Anwenderschicht zugänglich gemacht werden soll, soll die Implementierung möglichst

unabhängig von dem Betriebssystem und der Architektur des vom Benutzer verwendeten Rechensystems sein.

5.1.3 Zuverlässigkeit

Da das Framework zur realitätsnahen Simulation von Verkehrssystemen eingesetzt werden soll und die Simulationsergebnisse als Entscheidungsunterstützung dienen sollen, muss es die erforderlichen Funktionalitäten zuverlässig breitstellen. Bei der Simulation soll frühzeitig die Korrektheit überprüft werden, damit keine fehlerhaften Ergebnisse erzeugt werden. Wenn dennoch ein Fehler auftreten sollte, muss der Benutzer durch aussagekräftige Fehlermeldungen über die Art und den Grund des Fehlers informiert werden.

5.1.4 Wartbarkeit

Alle Funktionen und Schnittstellen des Frameworks sollen ausführlich dokumentiert werden. Zusätzlich wird der Quellcode verständlich und gemäß Code-Konventionen einheitlich gehalten. Der Quellcode wird offen verfügbar gemacht, um eine Weiterentwicklung des Systems auch nach Ende der Projektgruppe und von projektgruppenfremden Entwicklern zu ermöglichen.

Durch die vollständige Dokumentation des Quellcodes soll es einfach möglich sein, sich in das System einzuarbeiten und dessen Wartung zu gewährleisten.

5.1.5 Leistung

Bei dem Entwurf des Frameworks soll darauf geachtet werden, dass die Simulation auf einem durchschnittlichen Heimrechner (2GHz-Prozessor mit 2GB-Arbeitsspeicher) performant auszuführen ist und in einer angemessenen Zeit durchlaufen werden kann. Deshalb soll das Framework so entworfen werden, dass beim Simulationslauf unnötige Zugriffe auf den Arbeitsspeicher und die Datenquellen vermieden werden.

5.2 Systembeschreibung

Das Verkehrssimulationsframework besteht aus vier Hauptkomponenten, die im Folgenden näher erläutert werden. Zunächst wird ein Überblick über die Komponenten und deren Zusammenhänge gegeben. Anschließend folgt die Beschreibung der einzelnen Komponenten.

5.2.1 Überblick

Die Hauptkomponenten des Frameworks sind in zwei Architekturschichten unterteilt. Auf der ersten Schicht befindet sich das eigenentwickelte Multi-Agenten-Framework JASON, das

notwendige Funktionalitäten für die Entwicklung von nachrichtenbasierten Multi-Agenten-Simulationen bereitstellt. JASON ist dabei so konzipiert, dass es als Grundlage für die Erstellung verschiedener Simulationen benutzt werden kann. JASON wird auf der zweiten Schicht erweitert, um das Framework auf eine Simulation von Verkehrssystemen anwenden zu können. Hierbei sind entsprechende Akteure des Verkehrssystems als Agenten realisiert, wie z. B. Fahrzeugagenten oder Infrastrukturagenten. Die für eine Verkehrssimulation notwendigen Objekte der Realität (z. B. Fahrzeuge, Infrastrukturstandorte, etc.) werden im Verkehrsmodell abgebildet. Auf diesen modellierten Verkehrsmodell-Objekten arbeiten die durch JASON bereitgestellten Agenten. Die Eingabe und die Verarbeitung der benötigten Simulationsdaten werden von der Komponente Editor unterstützt. Die zweite Schicht bietet damit eine Basis für die Simulation von Verkehrssystemen.

In Abbildung 9 ist eine Übersicht des gesamten Verkehrssimulationsframeworks mitsamt den einzelnen Schichten dargestellt.

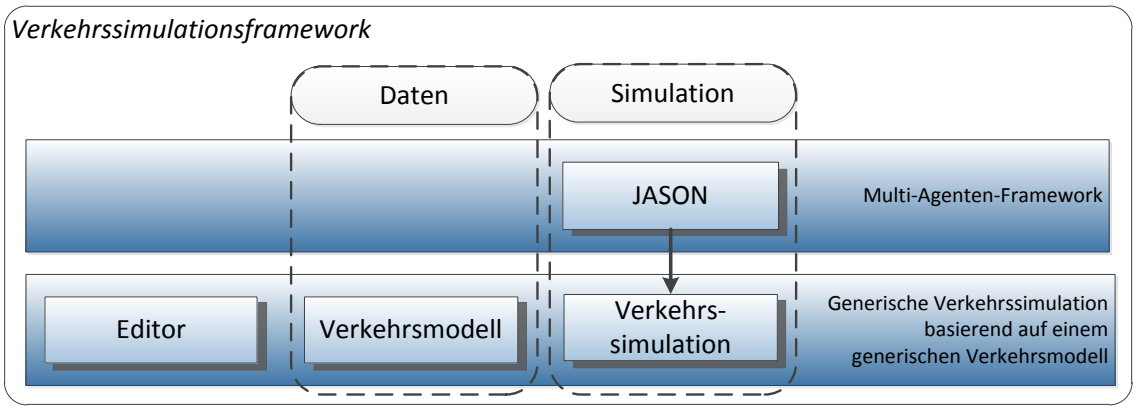

Abbildung 9: Überblick über die Hauptkomponenten

5.2.2 Multi-Agenten-Framework: JASON

JASON ist ein eigenentwickeltes Framework, welches die Möglichkeit bietet, Multi-Agenten-Simulationen zu entwickeln. Es beinhaltet die Basis, um Agenten zu erstellen, die jeweils eine Logik in Form von spezifischen Aktionen besitzen. Die Agenten können über Nachrichten miteinander kommunizieren und interagieren. Die Kommunikation findet über ein internes Nachrichtensystem statt. JASON verfügt zudem über einen Scheduler, der das Einplanen der Agenten-Aktionen zur Ausführung zu einem bestimmten Simulationszeitpunkt ermöglicht. Die Implementierung von JASON ist abstrakt gehalten, sodass eine Verwendung für jede Art von Multi-Agenten-Simulation möglich ist.

5.2.3 Verkehrssimulation

Die Verkehrssimulation stellt eine Erweiterung des JASON-Frameworks dar und bildet eine generische Simulation, die sich auf verschiedene Domänen von Verkehrssystemen anwenden lässt. Im Allgemeinen werden Infrastrukturpunkte und Fahrzeuge durch entsprechende Agenten des JASON-Frameworks realisiert. Die allgemeine Verkehrssimulation basiert auf dem Verkehrsmodell und bildet somit die Grundlage, um eine domänenspezifische Simulation, z. B. im Bereich der Schifffahrt, erstellen zu können.

5.2.4 Verkehrsmodell

Alle notwendigen Objekte eines Verkehrssystems werden durch das Verkehrsmodell abgebildet. Das Modell ist dabei weitestgehend abstrakt, wodurch es durch entsprechende Anpassungen und Erweiterungen zur Abbildung domänenspezifischer Verkehrssysteme anwendbar ist. So ermöglicht das Modell eine Abbildung unterschiedlicher Fahrzeugtypen (z. B. Schiffe, Landfahrzeuge oder Flugzeuge), Infrastrukturpunkte (z. B. Bunker, Häfen, Tankstellen oder Flughäfen) sowie ein graphenbasiertes Verkehrsnetzwerk, womit entsprechende Streckennetze in der Simulation realisiert werden. Das Verkehrsmodell wird mit Hilfe des von EMF erstellt, dessen Vorteile in Abschnitt 2.3 beschrieben wurden. Dadurch ist die Erstellung von Modellen über Programmcode nicht mehr notwendig und die Erstellung von Szenarien für eine Simulation wird vereinfacht.

5.2.5 Editor

Zur Unterstützung der Eingabe und Verarbeitung von Simulationsdaten stellt die Komponente Editor verschiedene Funktionalitäten bereit. So können die Simulationsdaten über eine grafische Oberfläche eingepflegt werden. Um die Eingabe für den Benutzer zu vereinfachen und Redundanzen während der Erstellung einer Konfiguration zu vermeiden, bietet diese Komponente außerdem eine Möglichkeit automatisiert Kopien bestimmter Bestandteile einer Simulation anzufertigen. Bspw. können mehrere Instanzen eines Fahrzeugtyps aus einer Vorlage erstellt werden. Die Überprüfung der eingegebenen Daten übernehmen die Validatoren. Diese überprüfen die Attribute auf deren Gültigkeit.

5.3 Entwurf des Multi-Agenten-Frameworks JASON

Dieser Abschnitt befasst sich mit dem Entwurf des Multi-Agenten-Frameworks JASON. Aus den in der Analyse aufgestellten funktionalen Anforderungen wird in diesem Abschnitt ein Entwurf für eine mögliche Implementierung von JASON erstellt. Abbildung 10 stellt einen möglichen Aufbau des Multi-Agenten-Frameworks dar.

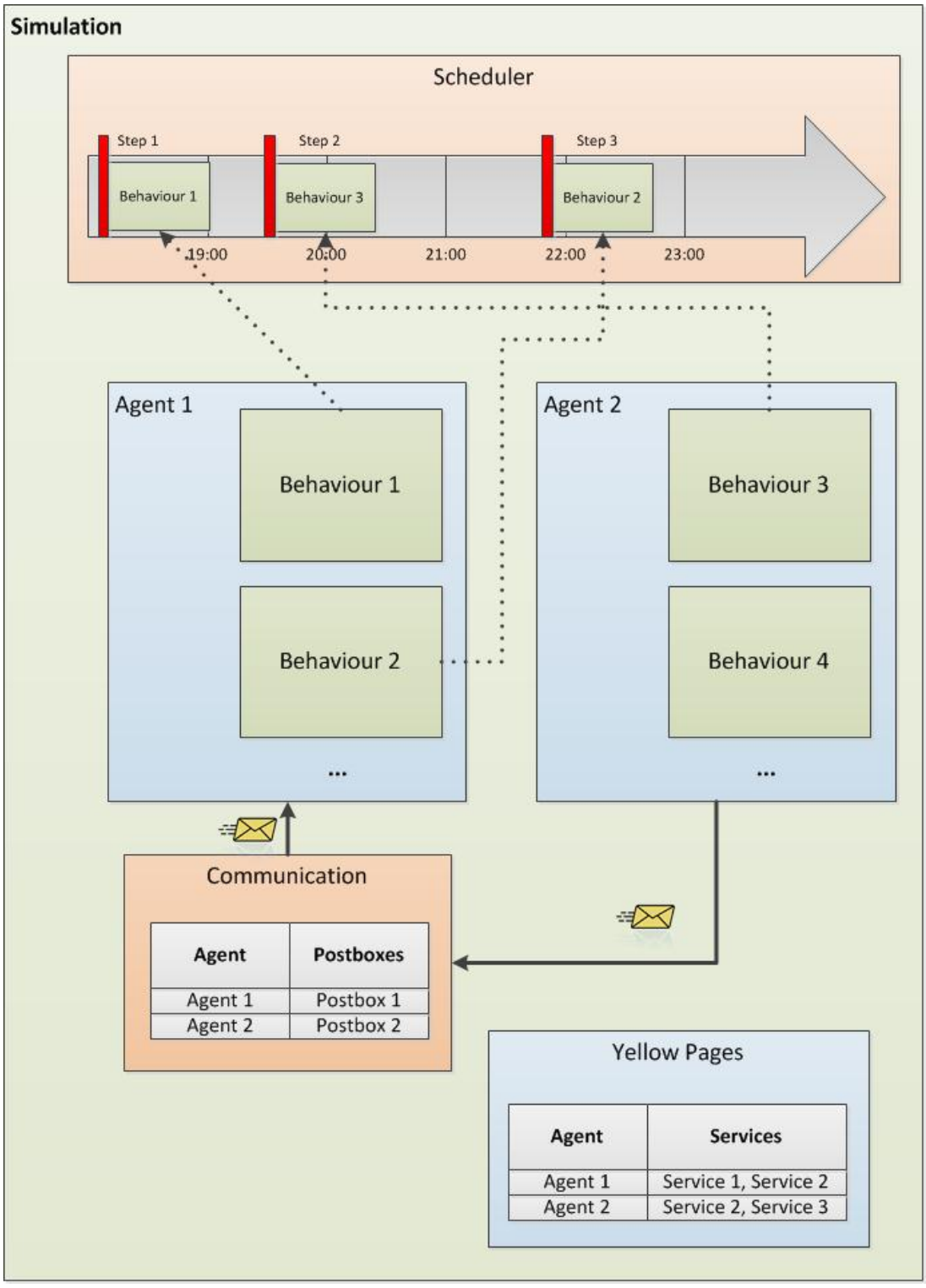

Abbildung 10: Entwurf des Multi-Agenten-Frameworks JASON

Im Folgenden werden die in der Grafik aufgeführten Komponenten erläutert.

Die *Simulation* beinhaltet alle Agenten der Simulation und den Scheduler, der die Aktionen der Agenten zur eingeplanten Simulationszeit ausführt. Eine Kommunikationskomponente ermöglicht den Agenten, innerhalb der Simulation Nachrichten auszutauschen. Außerdem

besitzt die Simulation eine eigene Komponente, um die von den Agenten angebotenen Dienste zu verwalten.

Agenten werden in Anlehnung an JADE umgesetzt. Agenten führen keine Aktionen aus, sondern agieren über sog. Behaviours. *Behaviours* ermöglichen es Agenten, innerhalb der Simulation zu handeln. Hierzu können sie diese Behaviours in den Scheduler einfügen. Der Scheduler führt die Aktionen dann zur spezifizierten Simulationszeit aus. Damit Agenten mit anderen Agenten interagieren können, können sie Nachrichten austauschen. Über *Yellow Pages* können Agenten anderen Agenten ihre Dienste bekannt machen bzw. diese erfragen.

Der *Scheduler* ist für die Verwaltung der Simulationszeit zuständig. Die Umsetzung des Schedulers erfolgt in Anlehnung an MASON. Das bedeutet, dass Aktionen in Schritten ausgeführt werden. Alle Behaviours, die zu einer Simulationszeit eingeplant sind, werden zusammen gestartet. Wenn diese abgeschlossen sind, wird die Simulationszeit bis zum nächsten Zeitpunkt übersprungen, an dem weitere Behaviours eingeplant sind. Diese werden im nächsten Schritt der Simulation ausgeführt. Bei der Ausführung der Simulation ist es möglich, Nachrichten sowohl im aktuellen als auch in einem späteren Schritt zu empfangen. Das heißt, ein Agent kann im aktuellen Schritt auf eine Nachricht warten, die erst in einem späteren Schritt gesendet und auch erst dann empfangen wird. Um dies zu ermöglichen, ist dem Scheduler zu jederzeit der aktuelle Ausführungsstatus der Behaviours bekannt.

5.4 Entwurf des Verkehrsmodells

Der in diesem Abschnitt zu beschreibende Entwurf eines Verkehrsmodells wird als Grundlage für die Modellierung bzw. Implementierung spezifischer Verkehrssysteme erstellt. Abbildung 11 bildet eine vereinfachte Sichtweise auf das zu entwickelnde Verkehrsmodell ab, welche nur die grundlegenden Bestandteile und deren Beziehungen abbildet. Nachfolgend werden dabei als Güter sowohl Waren Treibstoffe als auch Personen definiert.

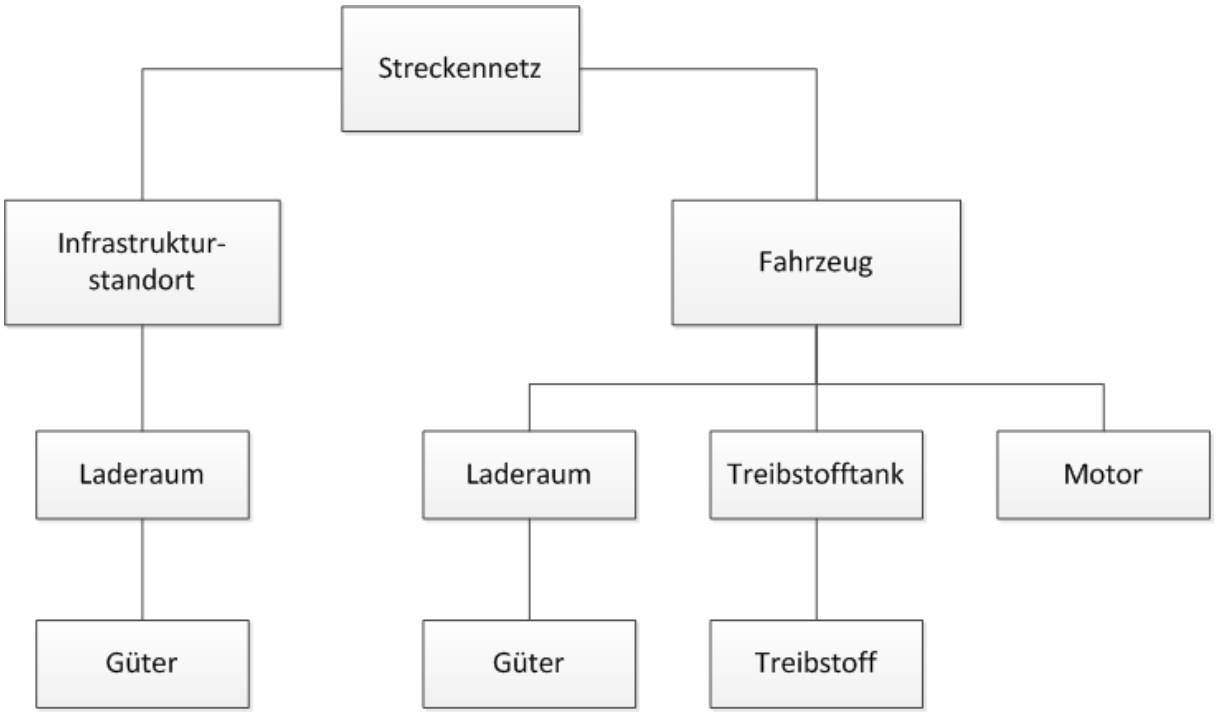

Abbildung 11: Verkehrsmodell Entwurf

5.4.1 Infrastrukturstandort

Ein Infrastrukturstandort stellt eine Entität innerhalb eines Verkehrssystems dar, die bspw. den Austausch von Gütern ermöglicht. Zu diesem Zweck besitzen diese einen Laderaum, der Güter aufnehmen und abgeben kann. Ein Infrastrukturstandort besitzt mehrere Verladestellen, wie z. B. Zapfsäulen. Verladestellen können individuell von Verkehrsteilnehmern durch vorherige Reservierung belegt werden und bieten Funktionalitäten an, um Güter aufnehmen und abgeben zu können. Des Weiteren ist es möglich, für die Verladestellen festzulegen, wie viele Güter pro Zeiteinheit von dieser ver- bzw. entladen werden können.

Neben der Reservierung von Verladestellen ist es möglich, eine Reservierung eines entsprechenden Gutes vorzunehmen, um den Transfer der Güter zu garantieren.

5.4.2 Fahrzeug

Fahrzeuge bilden die Verkehrsteilnehmer eines Verkehrssystems ab, die sich in diesem bewegen und Güter transportieren können. In Analogie zu Infrastrukturstandorten verfügen sie dazu ebenfalls über entsprechende Laderäume und Verladestellen. Darüber hinaus sind sie in der Lage, Güter zu bestimmten Zeiten an bestimmten Orten abzuholen bzw. anzuliefern. Solche Aktionen können sich regelmäßig wiederholen. Hierfür wird die Möglichkeit geschaffen, Wiederholungszeiträume festzulegen, um bspw. eine Aktion mehrmals täglich oder alle drei Wochen ausführen zu lassen.

Des Weiteren sind Fahrzeuge in der Lage Treibstoffe zu verbrauchen, um sich zwischen Streckenpunkten auf dem Streckennetz zu bewegen. Hierzu verfügen sie neben einem Laderaum auch über einen Treibstofftank. Zum Zwecke der Fortbewegung besitzen Fahrzeuge Motoren, welche die notwendige Leistung für das Fahren zur Verfügung stellen und dabei Treibstoffverbrauchen. Treibstoffe weisen dabei Eigenschaften, wie CO2-Schadstoffausstoß pro Einheit, auf um die durch den Treibstoffverbrauch erzeugten Emissionen, abzubilden.

5.4.3 Streckennetz

Das Streckennetz besteht aus einer Menge von Standorten, die durch Strecken miteinander verbunden sind. Dieses bildet somit die Verkehrsinfrastruktur eines Verkehrssystems ab, auf der sich die in Abschnitt 5.4.1 beschriebenen Infrastrukturstandorte befinden. Des Weiteren können sich die in Abschnitt 5.4.2 eingeführten Fahrzeuge auf den Strecken bewegen.

Spezifische Streckeneigenschaften, wie z. B. eine Länge oder eine Geschwindigkeitsbegrenzung können ebenfalls abgebildet werden.

5.5 Entwurf der Verkehrssimulation

In diesem Abschnitt wird der Entwurf der Verkehrssimulation beschrieben. Die Verkehrssimulation ist eine Simulation auf dem in Abschnitt 5.4 beschriebenen Verkehrsmodell. Grundsätzlich kann die Verkehrssimulation als eine Erweiterung von JASON angesehen werden, wobei es für das Streckennetz, das Fahrzeug und den Infrastrukturstandort des Modells entsprechende Agenten von JASON gibt.

In der folgenden Darstellung sind die wichtigsten Entitäten der Verkehrssimulation vereinfacht dargestellt.

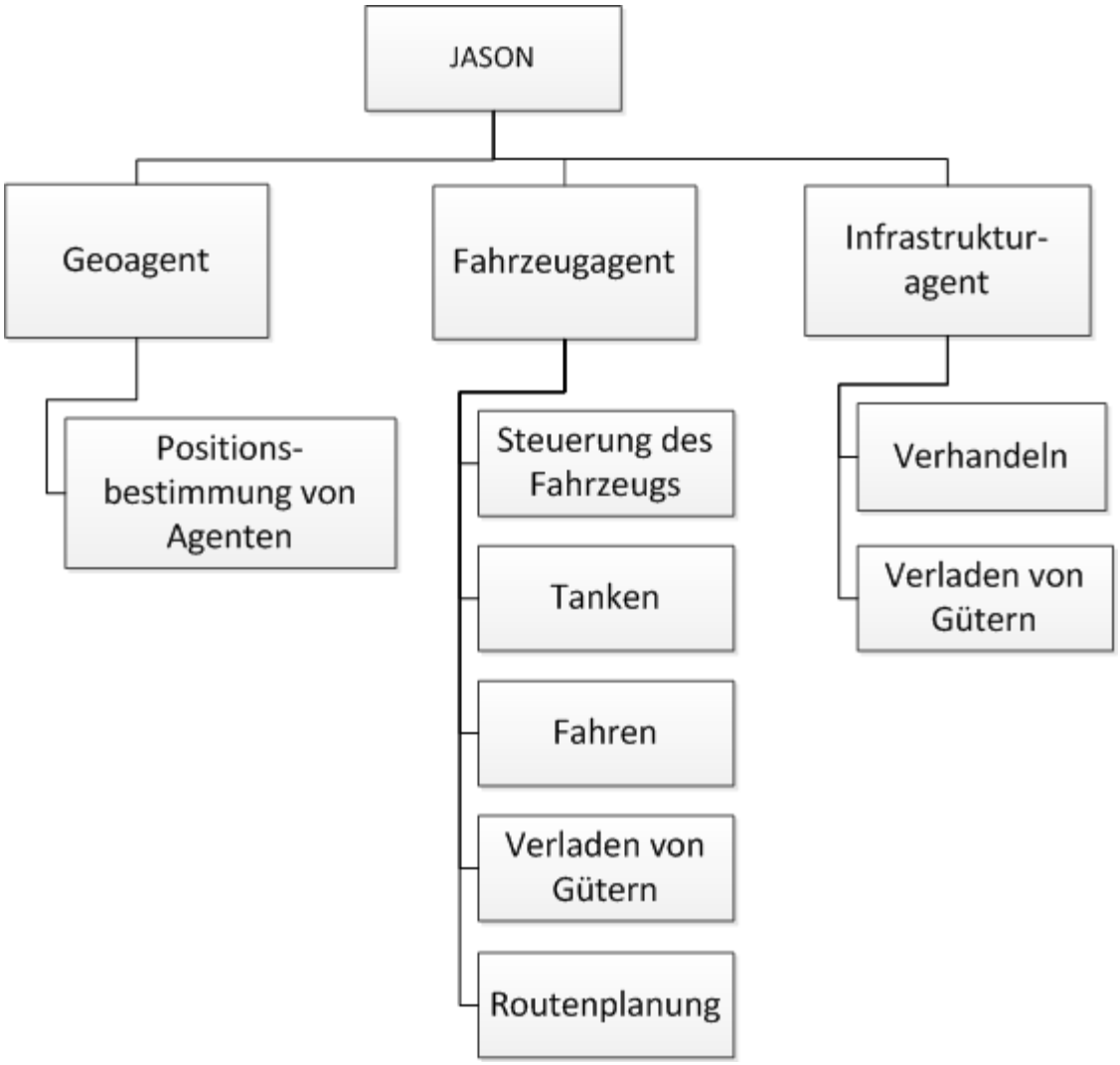

Abbildung 12: Verkehrssimulation Entwurf

5.5.1 Fahrzeugagent

Fahrzeugagenten stellen ein Fahrzeug mit dessen Logik innerhalb der Simulation dar. Ein Fahrzeugagent bewegt ein Fahrzeug innerhalb des Verkehrssystems und verfolgt dabei das Ziel, verschieden Aufgaben zu erfüllen. Dafür sind in der Logik verschiedene Aktionen für das Tanken, Fahren und Verladen von Gütern hinterlegt. Weiterhin umfassen die Aufgaben eines Agenten die Routenplanung innerhalb des Streckennetzes. Dazu gehören u. a. die Planung der optimalen Strecke zwischen zwei Knotenpunkten und die Ermittlung der dafür benötigten Motorleistung.

5.5.2 Infrastrukturagent

Ein Infrastrukturagent ist innerhalb der Simulation für die Verwaltung von Infrastrukturstandorten und die Bereitstellung von Services für andere Agenten im System zuständig. Der Infrastrukturagent besitzt ebenfalls eine Logik, mit der die unterschiedlichen Aktionen des

Agenten gesteuert werden. Hierzu zählt das Be- und Entladen von Gütern und Treibstoffen. Für diese Aktionen ist es notwendig, den Zeitpunkt, die Verladestelle und die Menge der jeweiligen Güter verhandeln zu können. Die entsprechenden Funktionen für das Verhandeln sind ebenso in der Logik des Infrastrukturagenten festgehalten.

5.5.3 Geoagent

Der Geoagent repräsentiert einen Agenten, der für die Verwaltung des Streckennetzes verantwortlich ist. Zudem besitzt er Informationen über die Standorte der Agenten innerhalb des Streckennetzes und über die Services, die diese Agenten anbieten. Hierdurch wird ermöglicht, die Position von Agenten abzufragen, die einen bestimmten Service anbieten (z. B. ein Infrastrukturagent, der einen bestimmten Treibstoff anbietet).

5.6 Entwurf des Editors

Neben dem Entwurf des Multi-Agenten-Frameworks JASON, des für eine Verkehrssimulation allgemeingültigen Modells und der darauf aufbauenden Verkehrssimulation, soll es einen Editor geben, der die Eingabe und Verarbeitung von Daten für eine Verkehrssimulation unterstützt.

Der Editor ist ein Plug-In-Framework mit grafischer Benutzeroberfläche, das zum Erstellen und zum Starten einer Verkehrssimulation dienen soll. Das Erstellen einer Simulation soll durch das Anlegen einer Simulationskonfiguration stattfinden. Diese Konfiguration muss alle für eine spezifische Simulation benötigten Daten enthalten.

Da die Simulation und das korrespondierende Simulationsmodell domänenspezifisch sein werden, ist auch der Editor an die domänenspezifischen Ausprägungen anzupassen. Dies bedeutet ebenfalls, dass die Benutzeroberflächen zum Bearbeiten einer Konfiguration an die Domäne und die jeweilige Simulationsausprägung anpassbar sein muss. Um dies zu ermöglichen, muss der Editor ein eigenes Metamodell (*EditorModel*) bekommen, das zur Beschreibung der einzelnen Daten einer Simulation dient. Der Editor selbst muss die Daten des *EditorModel* verarbeiten, serialisieren und visualisieren können. Abbildung 13 zeigt die Zusammenhänge des Editors auf Frameworkebene.

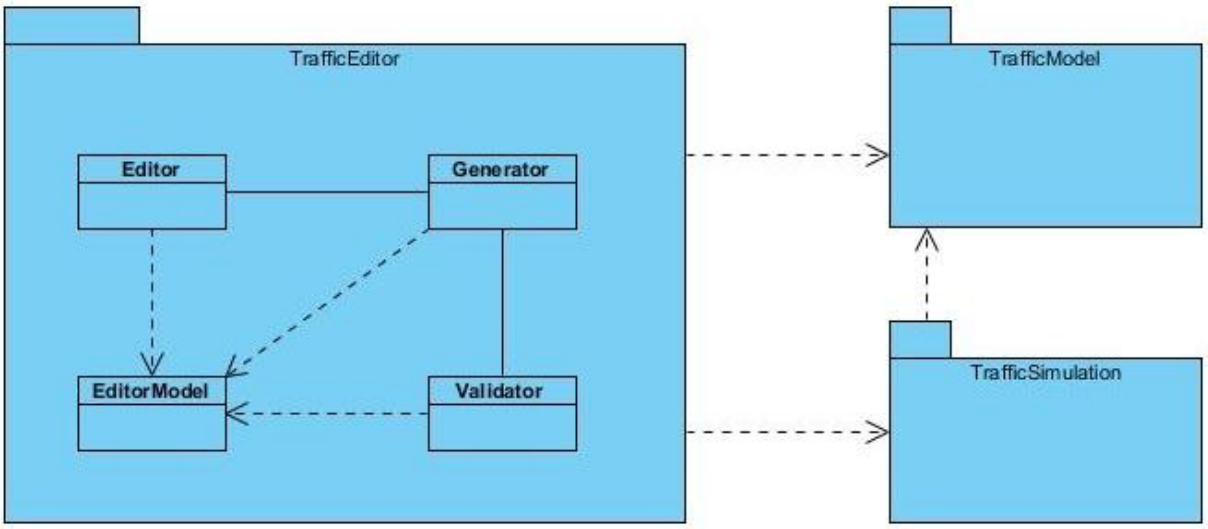

Abbildung 13: Zusammenhang des Editors auf Frameworkebene

Um die Eingabe für den Benutzer zu vereinfachen und Redundanzen während der Erstellung einer Konfiguration zu vermeiden, wird es die Teilkomponente Generator (siehe Abbildung 13) geben. Der Generator wird es ermöglichen, automatisiert Kopien bestimmter Bestandteile einer Simulationskonfiguration anzufertigen. Bspw. soll es möglich sein, mit ihm mehrere Instanzen eines Fahrzeugtyps aus einer Vorlage zu erstellen. Die Bestandteile der Simulationskonfiguration sollen vor der Simulationsausführung auf syntaktische Korrektheit und Konsistenz geprüft werden.

6 Implementierung

Dieses Kapitel behandelt die konkrete Umsetzung und Implementierung der Entwurfsentscheidungen aus dem vorherigen Kapitel. Entsprechend der Architektur des Frameworks und auf Basis des Entwurfs des Verkehrssimulationsframeworks erfolgt auch hier die Unterteilung der Abschnitte in JASON, Verkehrsmodell, Verkehrssimulation und Editor.

Grundsätzlich werden für die Erläuterung unterschiedliche Diagrammtypen herangezogen. UML-Klassendiagramme dienen hierbei der strukturellen Darstellung der einzelnen Komponenten und deren Beziehungen zueinander. Weiterhin werden an geeigneten Stellen Prozessablaufdiagramme sowie UML-Sequenzdiagramme verwendet, um interne Prozessabläufe des Frameworks deutlich zu machen (z. B. das Nachrichtensystem in JASON). Bei der Beschreibung der Editor-Komponente werden zusätzlich Screenshots der grafischen Oberfläche benutzt.

6.1 Implementierung des Multi-Agenten-Frameworks JASON

Dieser Abschnitt befasst sich mit der Implementierung des Multi-Agenten-Frameworks JASON. Er baut auf dem Entwurf auf und stellt detailliert die grundlegenden Bausteine von JASON dar.

Eine Simulation beruht in JASON auf der Klasse *Simulation*, die in Abschnitt 6.1.1 beschrieben wird. Eine Simulation beinhaltet Agenten (siehe Abschnitt 6.1.2), die über *Behaviours* (siehe Abschnitt 6.1.3) Aktionen ausführen. Der zeitliche Ablauf dieser Aktionen wird innerhalb einer Simulation durch den *Scheduler* (siehe Abschnitt 6.1.4) realisiert. Die Kommunikation zwischen den Agenten basiert auf Nachrichten und wird in Abschnitt 6.1.5 beschrieben. Über Nachrichten ist es möglich, dass Agenten ihre Dienste anderen Agenten anbieten. Dienste werden in Abschnitt 6.1.6 beschrieben.

6.1.1 Simulation

Simulation ist die grundlegende Klasse, die es ermöglicht benutzerspezifische Simulationen anzulegen. Um eine Simulationszeit zu realisieren, hat die Simulation einen definierten Start- und Endzeitpunkt. Der Durchlauf durch die Simulationszeit wird über den *Scheduler* realisiert, auf den in Abschnitt 6.1.4 näher eingegangen wird. Abbildung 14 zeigt die Klasse *Simulation* mit den wichtigsten Methoden.

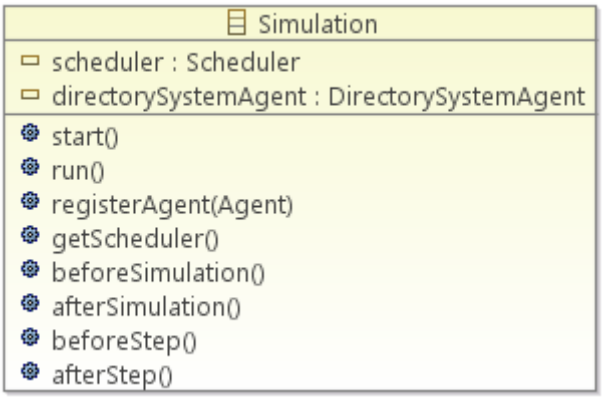

Abbildung 14: Klasse Simulation

Agenten können sich über die *registerAgent*-Methode registrieren, sodass sie innerhalb der Simulation über das Nachrichtensystem kommunizieren können. Agenten werden nach der Registrierung in der Simulation initialisiert.

Registrierte Agenten können über die *getScheduler*-Methode der *Simulation* auf den *Scheduler* zugreifen, um ihre *Behaviours* einzutragen, sodass sie zu einem bestimmten Simulationszeitpunkt ausgeführt werden. Solche Simulationszeitpunkte, zu denen ein oder mehrere *Behaviours* eingeplant sind, werden als Steps bezeichnet. Die Simulation führt die Steps des *Schedulers* über die *run*-Methode aus, um so den Fortschritt der Simulationszeit zu simulieren. Über die Methoden *beforeStep* und *afterStep* ist es möglich, vor und nach der Ausführung eines Steps eigene Logik einzufügen.

Eine Simulation beginnt bei Aufruf der *start*-Methode. Die Simulation wird so lange ausgeführt, bis entweder der definierte Endzeitpunkt erreicht wird, oder keine weiteren *Behaviours* für die zukünftige Simulationszeit eingeplant sind. Ähnlich wie bei der Ausführung eines Steps ist es über die Methoden *beforeSimulation* und *afterSimulation* möglich, die Simulation um zusätzliche Logik, wie bspw. den Aufbau einer Datenbankverbindung und das Speichern der Simulationsergebnisse, zu erweitern.

Außerdem beinhaltet jede Simulation einen *DirectorySystemAgent*, der beim Simulationsstart aufgerufen wird. Während der Simulationslaufzeit dient er als eine Art „Yellow Pages“, bei der Agenten ihre Dienste anbieten und Dienste anderer Agenten erfragen können. Eine genauere Beschreibung des *DirectorySystemAgents* folgt in Abschnitt 6.1.9.

6.1.2 Agent

Der *Agent* dient dazu, Akteure innerhalb einer Simulation zu erstellen. Agenten selbst weisen keine eigene Handlungslogik auf, sondern handeln über *Behaviours*. Agenten bieten aber Grundfunktionalitäten, die von den *Behaviours* verwendet werden, um das Verhalten eines

Agenten umzusetzen. Außerdem können *Behaviours* ihren Agenten dazu nutzen, gemeinsam verwendete Daten abzulegen. Über den Agenten können *Behaviours* zur Ausführung in der Simulationszeit eingeplant werden. Ein Agent kann mit anderen Agenten derselben Simulation kommunizieren. Hierzu kann er Nachrichten senden und empfangen. Zur Identifikation eines Agenten verfügt dieser über einen eindeutigen Namen und einen eigenen „Postkasten“ (Klasse: *MessageBox*). Nähere Informationen über den Nachrichtenaustausch sind in Abschnitt 6.1.5 zu finden.

Agenten können anderen Agenten Dienste anbieten und besitzen dafür eine Beschreibung (Klasse: *AgentDescription*). Auf die *AgentDescription* wird im Kontext der Dienste in Abschnitt 6.1.6 eingegangen.

Wenn sich ein Agent an einer Simulation registriert, wird er initialisiert. Diese Initialisierung beinhaltet eine *setup*-Methode, die dazu genutzt werden soll, initiale *Behaviours* einzuplanen.

6.1.3 Behaviour

Behaviour ist die Basisklasse, um ein Verhalten eines Agenten zu erstellen. *Behaviours* können in den *Scheduler* eingefügt und somit zu bestimmten Simulationszeitpunkten ausgeführt werden. Die auszuführende Logik wird in der *action*-Methode implementiert, die jedes Mal aufgerufen wird, wenn ein *Behaviour* vom *Scheduler* gestartet wird.

Jedes *Behaviour* läuft während der Ausführung der *action*-Methode in einem eigenen Thread. Der Grund hierfür ist es, die Möglichkeit zu schaffen, die Ausführung eines *Behaviours* beim Warten auf Nachrichten solange zu unterbrechen, bis eine passende Nachricht eingetroffen ist, oder eine bestimmte Simulationszeit abgelaufen ist (Timeout). Das Prinzip des Wartens auf Nachrichten lehnt sich an die Implementierung in JADE an. Bei der Ausführung eines *Behaviours* wird garantiert, dass nur eines der *Behaviours* zurzeit aktiv ist. Somit ist es trotz der Verwendung von Threads nicht notwendig, dass die *Behaviours* eines Agenten auf gemeinsam genutzte Daten des Agenten synchronisiert werden. Solange mindestens ein *Behaviour* zu einer Simulationszeit läuft, ist garantiert, dass die Simulationszeit nicht weiterläuft.

Behaviours können sich oder andere *Behaviours* über den Agenten zur Ausführung einplanen. Somit kann ein fortlaufendes Handeln eines Agenten gewährt werden. Außerdem ist es so möglich, die Dauer einer Aktion zu simulieren, indem sich ein *Behaviour* nach dem Start einer Aktion zu einem späteren Simulationszeitpunkt erneut einplant, um die Aktion zu beenden.

6.1.4 Scheduler

Der *Scheduler* ist in der Simulation dafür zuständig, dass *Behaviours* zu bestimmten Simulationszeiten ausgeführt werden. Agenten können hierzu ihre *Behaviours* über die *schedule*- bzw. *scheduleIn*-Methode zu einem definierten Zeitpunkt zur Ausführung einplanen. Für die Verwaltung der eingeplanten *Behaviours* nutzt der *Scheduler* eine Prioritätswarteschlange, in der die *Behaviours* nach aufsteigender Ausführungszeit sortiert sind.

Das Durchlaufen der Simulationszeit erfolgt schrittweise und wird durch die *Simulation* angestoßen. Das bedeutet, dass alle *Behaviours* mit demselben Ausführungszeitpunkt in einem Schritt der Simulation ausgeführt werden. Hierzu ermittelt der *Scheduler* anhand der Prioritätswarteschlange den nächsten Simulationszeitpunkt, zu dem *Behaviours* eingeplant sind, und startet alle *Behaviours* dieses Zeitpunkts. Zeitpunkte, zu denen keine Behaviours eingeplant sind, werden somit nicht ausgeführt und übersprungen. Ein Schritt der Simulation wird erst beendet, wenn kein *Behaviour* der aktuellen Simulationszeit mehr aktiv ist. Dies geschieht, um zu vermeiden, dass *Behaviours* aus der vergangenen Simulationszeit zusammen mit *Behaviours* der Zukunft ausgeführt werden. Da jedes *Behaviour* in einem eigenen Thread gestartet wird, existiert mit dem *ProcessingStatus* die Möglichkeit, den aktuellen Ausführungsstatus eines *Behaviours* zu ermitteln, um festzustellen, wann der nächste Simulationsschritt gestartet werden kann. Die *step*-Methode zur Ausführung eines Simulationsschrittes ist im Folgenden dargestellt. Der Übersichtlichkeit halber stellt der abgebildete Quellcode eine Vereinfachung des Originals dar.

```
1.  public void step() {
2.      time = schedule.peek().getTime();
3.      startCurrentBehaviours();
4.      ProcessingStatus status = ProcessingStatus.RUNNING;
5.      while (status == ProcessingStatus.RUNNING) {
6.          schedule.wait();
7.          status = updateStatus();
8.          if (status != ProcessingStatus.RUNNING) {
9.              if (startCurrentBehaviours()) {
10.                 status = ProcessingStatus.RUNNING;
11.             }
12.         }
13.     }
14. }
```

Abbildung 15: step-Methode des Schedulers

Beim Start eines *Behaviours* wird dieses in ein *BehaviourRunnable* geschachtelt, welches den *ProcessingStatus* des *Behaviours* verwaltet. Dem *BehaviourRunnable* wird ein Thread aus einem Threadpool zugewiesen. Dieses ruft dann die *action*-Methode des *Behaviours* auf. Vor der Ausführung des *Behaviours* wird der *ProcessingStatus* auf *RUNNING* gesetzt. Fertige *Behaviours* erhalten den Status *DONE*. Sollte ein *Behaviour* auf den Empfang einer Nachricht warten, wird der Status auf *WAITING* gesetzt.

Bei einer Statusänderung wird der *Scheduler* benachrichtigt und prüft dann den Status aller derzeit ausgeführten *Behaviours*, um festzustellen, ob der Simulationsschritt beendet ist und die Simulationszeit weiterlaufen kann. Solange noch mindestens ein *Behaviour* den Status *RUNNING* hat, kann die Simulationszeit nicht weitergeführt werden.

Der *WAITING*-Status begründet sich dadurch, dass der Thread eines wartenden *Behaviours* im Gegensatz zu einem fertigen *Behaviour* noch läuft, die Simulationszeit aber weitergeführt werden soll. Ansonsten wäre es nicht möglich, auf Nachrichten zu warten, die erst zu einem späteren Simulationszeitpunkt versendet werden, ohne die Ausführung der Simulation zu blockieren. Um jedoch die Wartezeit auf Nachrichten einzuschränken, gibt es die Möglichkeit, beim Empfang ein Timeout zu spezifizieren.

Die *receive*-Methode des Agenten, in der der *ProcessingStatus* auf *WAITING* gesetzt wird, ist im Folgenden in vereinfachter Form dargestellt.

```
1.  public Message receive(MessageTemplate messageTemplate, long timeout) {
2.      …
3.      Message message = messageBox.search(messageTemplate);
4.      if (message != null) {
5.          return message;
6.      }
7.      while (!timedOut() && (message == null) {
8.          behaviourRunnable.changeStatus(ProcessingStatus.WAITING);
9.          getMessageBox().wait();
10.         message = messageBox.search(messageTemplate);
11.     }
12.     return message;
13. }
```

Abbildung 16: receive-Methode eines Agenten

6.1.5 Kommunikation

Die Kommunikation in JASON wird über Nachrichten (Klasse: *Message*) realisiert. Für den Empfang von Nachrichten verfügt jeder Agent über einen „Postkasten“ (Klasse: *Message-*

Box). Das Senden von Nachrichten erfolgt über ein Nachrichtensystem (Klasse: *MessageSystem*), das jeden Agenten und seine *MessageBox* kennt und die Nachrichten zustellt. Für die Zustellung werden die eindeutigen Agentennamen verwendet. Beim Empfangen von Nachrichten nutzt ein Agent *MessageTemplates*, um nur diejenigen Nachrichten der *MessageBox* zu entnehmen, die im Weiteren verarbeitet werden sollen. Im Folgenden werden die für die Kommunikation relevanten Bestandteile erläutert.

Nachrichten werden in JASON durch *Messages* repräsentiert. Jede *Message* enthält einen Sender und einen oder mehrere Empfänger, wobei diese die Agentennamen sind. Der eigentliche Inhalt der Nachricht ist ein einfaches Java-Objekt. Das ermöglicht das Senden verschiedener Inhalte. Weiterhin weist jede Nachricht ein ID-Feld auf, durch das jede Nachricht einer Konversation zugeordnet werden kann. Außerdem ist es möglich über entsprechende Felder Antworten auf eine Nachricht dieser Nachricht zuzuordnen sowie ein Timeout für Antworten vorzugeben. Über das *Protocol*-Feld wird das für die Kommunikation verwendete Protokoll angegeben. Das *Performative*-Feld gibt den aktuellen Stand des Kommunikationsverlaufs in Anlehnung an das FIPA-Performative an. Sollte die Nachricht in Anfrage auf einen Dienst verwendet werden, kann der *ServiceName* im dafür vorgesehenen Feld angegeben werden.

Die *MessageBox* beinhaltet die Liste aller empfangenen Nachrichten eines Agenten und bietet eine Funktion, um aus der Liste eine Nachricht zu entnehmen, die einem *MessageTemplate* entspricht. Das *MessageTemplate* ermöglicht das Filtern von Nachrichten. Vorgefertigte *MessageTemplates* können Nachrichten hinsichtlich der Erfüllung bestimmter Kriterien überprüfen, bspw. ob ein bestimmtes Protokoll verwendet wird, oder ob die Nachricht eine Antwort auf eine zuvor gesendete Nachricht ist. Außerdem existieren Templates, die logischen Ausdrücke darstellen, um so komplexere Filter zu erstellen.

Bei der Registrierung an der Simulation registrieren sich Agenten auch beim *MessageSystem*. Dadurch kennt das *MessageSystem* die *MessageBoxes* jedes registrierten Agenten. Beim Senden einer Nachricht ruft der Agent die *send*-Methode des Nachrichtensystems auf. Hier wird die Nachricht jedem der Empfänger zugestellt, indem eine Kopie der Nachricht in die *MessageBox* gelegt wird. Das Kopieren der Nachricht wird vorgenommen, um Seiteneffekte zu vermeiden, wenn mehrere Empfänger auf derselben Nachricht arbeiten. Außerdem wird jedes auf Nachrichten wartende *Behaviour* eines Agenten benachrichtigt, damit es überprüfen kann, ob die Nachricht von Interesse ist oder ob weiter auf Nachrichten gewartet werden muss. Der zuvor beschriebene Nachrichtenaustausch zwischen Agenten ist in Abbildung 17 dargestellt.

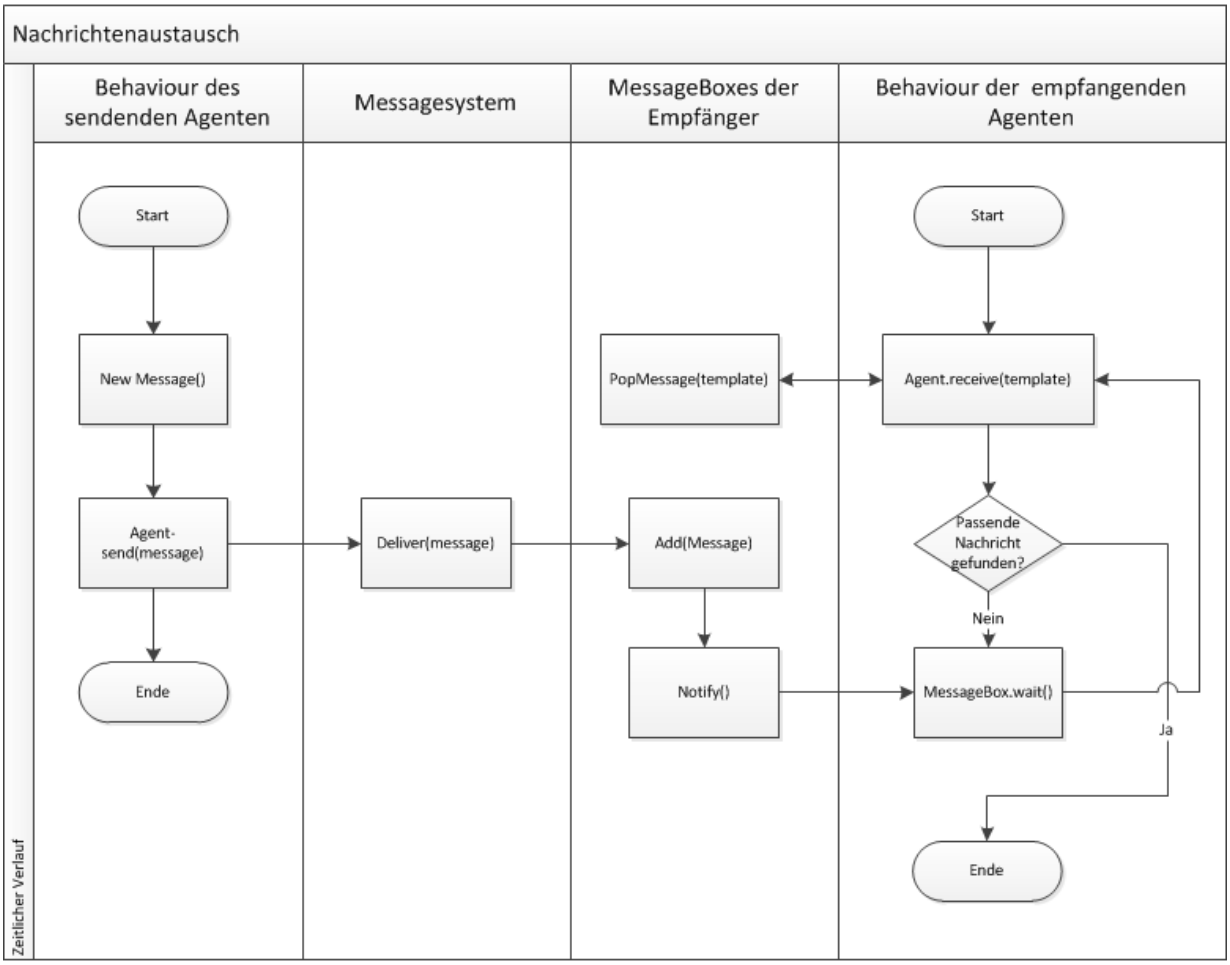

Abbildung 17: Nachrichtenaustausch

6.1.6 Dienste

Innerhalb der Simulation können Agenten anderen Agenten Dienste (*Services*) anbieten. Services dienen dazu, die Interaktion zwischen Agenten zu erleichtern. Angebotene *Services* werden anderen Agenten über den *DirectorySystemAgent* bekannt gemacht.

6.1.7 AgentDescription

Jeder Agent verfügt über eine Beschreibung (Klasse: *AgentDescription)*, die er beim *DirectorySystemAgent* registriert. Die *AgentDescription* beinhaltet den Namen des Agenten und Informationen über dessen angebotene Dienste in Form von *ServiceDescriptions*. Mithilfe der *AgentDescription* ist es anderen Agenten möglich, zu erfahren, welche Dienste der Agent anbietet.

Die *ServiceDesciption* beschreibt einen vom Agenten angebotenen Dienst. Sie beinhaltet den Namen des Dienstes sowie das zur Kommunikation verwendete Protokoll und eine Liste weiterer beschreibender Eigenschaften.

6.1.8 Service

Der den *Service*-anfragende Agent wird im Folgenden als Initiator bezeichnet, der *Service*-anbietende Agent als Anbieter. Ein *Service* besteht aus einer *ServiceDescription*, einem *MessageTemplate* und einer Methode zur Erstellung des *ServiceHandlerBehaviours*.

Angebotene *Services* eines Agenten werden von dessen *ServiceBehaviour* verwaltet. Das *ServiceBehaviour* ist dafür zuständig, die initialen Nachrichten aller Serviceanfragen zu empfangen und entsprechend den angebotenen *Services* zuzuweisen. Zum Empfangen dieser initialen Nachricht wird das im *Service* definierte *MessageTemplate* genutzt. Für jede initiale Nachricht wird dann ein neues *ServiceHandlerBehaviour* erstellt und in den *Scheduler* eingefügt. Hierzu nutzt das *ServiceBehaviour* die *createNewServiceHandlerBehaviour*-Methode des *Services*, für den die Anfrage bestimmt ist. Das *ServiceHandlerBehaviour* hat die Aufgabe diese Serviceanfrage zu bearbeiten. Hierunter fallen neben der Bearbeitung der initialen Nachricht auch alle weiteren Schritte der Kommunikation mit dem Initiator.

Durch die *addService*-Methode des Agenten können neue *Services* hinzugefügt werden. Diese werden dann dem *DirectorySystemAgent* bekannt gemacht und durch das *ServiceBehaviour* verwaltet. Analog dazu existiert eine *removeService*-Methode zum Entfernen eines *Services*.

Für den Initiator gibt es als Gegenstück zum *ServiceHandlerBehaviour* mit dem *ServiceInitiatorBehaviour* eine Klasse, die ein Gerüst zum Stellen von Serviceanfragen bietet.

Aufbauend auf dem *Service*, *ServiceInitiatorBehaviour* und *ServiceHandlerBehaviour* werden drei Protokolle bereitgestellt. Jedes dieser Protokolle ermöglicht es *Services* zu implementieren, die sich an einen im Protokoll vorgegebenen Kommunikationsfluss zwischen Initiator und Anbieter halten. Diese drei Protokolle sind:

- **Inform-Protokoll**: Dieses Protokoll wird dazu verwendet, damit Initiator einen oder mehrere Anbieter über ein Ereignis informieren kann.
- **Request-Protokoll:** Das Request-Protokoll ist angelehnt an das entsprechende FIPA-Protokoll. Es erlaubt einem Initiator eine Anfrage an einen oder mehreren Anbieter zu stellen. Dieser kann die Anfrage bearbeiten, zurückweisen oder eine Fehlermeldung zurückgeben.
- **Contract-Net-Protokoll:** Auch das Contract-Net-Protokoll ist an das entsprechende FIPA-Protokoll angelehnt. Das Protokoll gibt den Ablauf einer Verhandlung vor. Ein Initiator bittet einen oder mehrere Anbieter, ihm ein Angebot auf eine Anfrage zu machen. Der Anbieter kann ein Angebot zurücksenden, oder die Teilnahme an der Verhandlung ablehnen. Sobald der Initiator die Antworten aller Anbieter erhalten hat, kann er ein oder mehrere Angebote annehmen oder zurückweisen. Die Anbieter, de-

ren Angebote angenommen wurden, bearbeiten die Anfrage, oder geben eine Fehlermeldung zurück.

Um diese Protokolle umzusetzen, müssen *Service*, *ServiceInitiatorBehaviour* und *ServiceHandlerBehaviour* der Protokolle überschrieben werden.

6.1.9 DirectorySystemAgent

Eine Simulation hat einen *DirectorySystemAgent*, durch den eine Verwaltung der angebotenen Dienste aller Agenten vorgenommen wird. Hierzu wurde eine agentenbasierte Lösung gewählt, damit das vorhandene Kommunikationssystem für das Registrieren und Anfragen von *Services* genutzt werden kann.

Jeder Agent registriert seine *AgentDescription* und damit seine angebotenen Dienste beim *DirectorySystemAgent*. Dieser verwaltet alle registrierten *AgentDescriptions* in seiner *KnowledgeBase*. Der *DirectorySystemAgent* stellt hierzu den *RegisterAgentDescriptionService* bereit. Für das Registrieren oder Aktualisieren seiner *AgentDescription* kann das *RegisterAgentDescriptionInitiatorBehaviour* verwendet werden.

Außerdem ist es möglich, die *KnowledgeBase* des *DirectorySystemAgents* zu durchsuchen. Zu diesem Zweck bietet der *DirectorySystemAgent* den *RequestAgentDescriptionService* an. Interessierte Agenten können dem *DirectorySystemAgent* hierzu eine *AgentDescription* schicken, die als Vorlage für die Suche dient. Als Antwort erhalten sie all die Beschreibungen der Agenten, die die Kriterien dieser Vorlage erfüllen.

6.2 Implementierung Verkehrsmodell

Im folgenden Abschnitt wird auf die Implementierung des Verkehrsmodells eingegangen, welches ein Modell für Verkehrssimulationen auf Basis von JASON darstellt. Das Verkehrsmodell wird auf Basis des in Abschnitt 5.4 beschriebenen Entwurfs implementiert. Es wird als Grundlage für die Abbildung domänenspezifischer Verkehrssysteme modelliert. Alle Verkehrsmodell-Objekte erweitern ein *Root*-Objekt und erhalten hierdurch einen Namen sowie eine ID, sodass eine eindeutige Identifizierung aller Modellobjekte ermöglicht wird.

Die Modellierung der notwendigen Objekte wird mittels EMF realisiert und im Folgenden näher beschrieben.

6.2.1 Streckennetz

Die Klasse *Network* repräsentiert das Streckennetz, auf dem sich die Verkehrsteilnehmer bewegen können. Wie in Abbildung 18 dargestellt besteht das Streckennetz aus einer Menge von Strecken und Streckenpunkten sowie unterstützenden Funktionalitäten.

Streckenpunkte (Klasse: *TrackPoint*) bilden konkrete Standorte im Netzwerk ab. Diese Streckenpunkte besitzen aufgrund der Abstrahierung keine geografischen Informationen, da die Standortbestimmung in einem domänenspezifischen Modell sowohl über Längen- und Breitengrade als auch über x-y-Koordinaten denkbar wäre und bei Bedarf entsprechend erweitert werden kann.

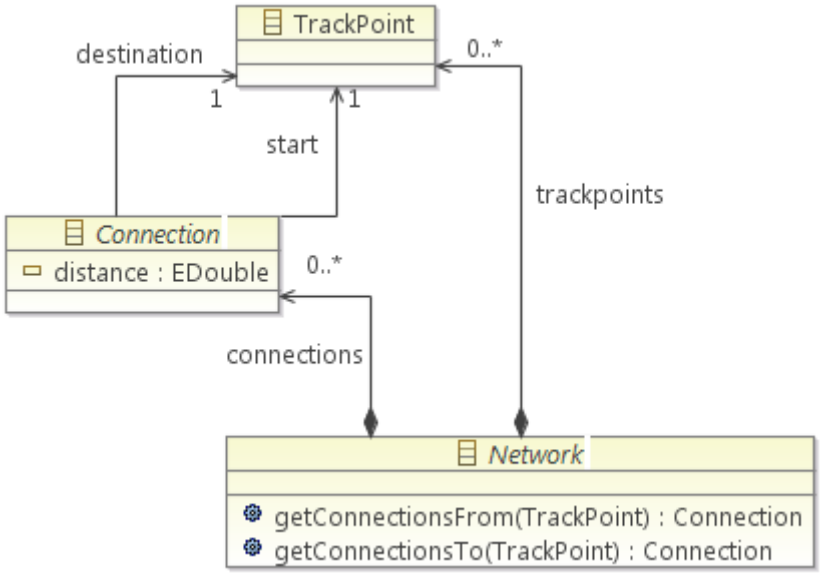

Abbildung 18: Verkehrsmodell Streckennetz

Neben den Streckenpunkten besteht das Streckennetz aus einer Menge von Strecken (Klasse: *Connection*) einer bestimmten Länge in Kilometer. Diese verbinden jeweils zwei dieser Streckenpunkte miteinander. *Connections* sind als eigene Klasse modelliert, da dies im Gegensatz zu einer Adjazenzmatrix auch mehrere direkte Verbindungen zwischen zwei Streckenpunkten ermöglicht. Strecken können neben der Länge auch weitere Eigenschaften, wie Geschwindigkeitsbegrenzungen, erhalten. Um die direkten Verbindungen von bzw. zu einem Streckenpunkt ermitteln zu können, bietet die Klasse *Network* die beiden Methoden *getConnectionsFrom* und *getConnectionsTo*.

6.2.2 Güteraustausch

In einem Verkehrssystem müssen sowohl Verkehrsteilnehmer (Klasse: *Vehicle*) als auch Infrastrukturstandorte (Klasse: *InfrastructurePoint*) einen Güteraustausch unterstützen. Aus diesem Grund sind beide Entitäten über die Klasse *LoadOwner* generalisiert. Diese besitzt zum Zwecke des Güteraustausches einen aktuellen Standort im Streckennetz. Analog zur Aufnahme von Gütern können Verkehrsteilnehmer auch Treibstoffe aufnehmen. Zur Vermeidung der redundanten Implementierung der benötigten Logik werden die Funktionalitäten zur Verwaltung von Laderäumen und zum Austausch von Gütern durch den in Abbildung 19

dargestellten *LoadManager* bereitgestellt. Hierdurch ist es möglich, dass für die Verwaltung des Treibstoffs eine zweite, als *FuelManager* bezeichnete Instanz der Klasse *LoadManager* verwendet wird.

In den folgenden Abschnitten wird genauer auf die im *LoadManager* enthaltenen Funktionalitäten eingegangen.

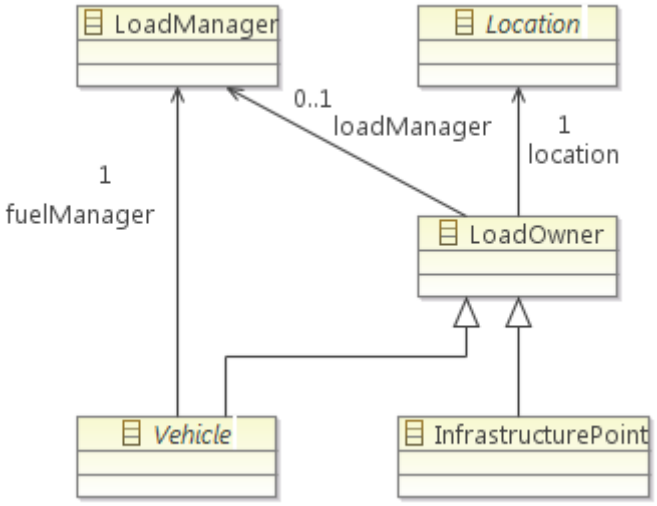

Abbildung 19: Verkehrsmodell Güteraustausch

6.2.3 Güterverwaltung

Um Güter austauschen zu können, müssen diese definiert werden, wofür die Klasse *LoadType* aus Abbildung 20 vorgesehen ist. Damit die Charakteristika eines Gutes abgebildet werden können, besitzen *LoadTypes* Eigenschaften wie Volumen in Kubikmeter und Gewicht in Kilogramm pro Einheit.

Eine Spezialisierung von *LoadTypes* sind *FuelTypes*, welche die Treibstoffe eines Verkehrssystems darstellen. Diese *FuelTypes* besitzen neben den Attributen des *LoadTypes* weitere Eigenschaften für den Brennwert in Kilowattstunden sowie den Ausstoß von CO_2, NO_x und SO_x in Gramm pro Einheit. Diese Werte können genutzt werden, um den Verbrauch und die Emissionen eines Verkehrsteilnehmers zu bestimmen.

Zur Lagerung von Gütern wird die Klasse *LoadSpace* modelliert. Ein *LoadSpace* kann eine Menge eines einzigen LoadTypes aufnehmen. Zum Hinzufügen bzw. Entfernen von Gütern werden vom *LoadSpace* die Methoden *add* und *remove* bereitgestellt. Des Weiteren wird über die Methoden *getMaxAmountToAdd* und *getMaxAmountToRemove* die Funktionalität bereitgestellt, um festzustellen, welche Mengen hinzugefügt bzw. entnommen werden können. Der *LoadSpace* besitzt eine unbegrenzte Kapazität, weshalb die Spezialisierung *AmountLimitedLoadSpace* einen Lagerraum mit einer maximalen Kapazität bereitstellt. Zur

Lagerung von Treibstoffen kann ein *FuelTank* benutzt werden, der ebenfalls ein *AmountLimitedLoadSpace* ist. Analog hierzu wird der *VolumeLimitedLoadSpace* modelliert, welcher anstatt einer maximalen Kapazität ein maximales Volumen bereitstellt. Neben den zuvor dargestellten *LoadSpaces* wird der *InfiniteContentLoadSpace* erstellt, um bspw. Tankstellen abbilden zu können, die über einen unbegrenzten Vorrat an Treibstoffen verfügen.

Wie bereits in Abschnitt 6.2.2 beschrieben, existiert für die Verwaltung von Gütern die Klasse *LoadManager*. Ein *LoadManager* kann mehrere *LoadSpaces* verwalten und somit auch mit unterschiedlichen Gütern umgehen. Über die Methode *getSupportedTypes* besteht die Möglichkeit, die von den *LoadSpaces* unterstützen Güter zu erfragen. Darüber hinaus bietet der *LoadManager* über die Methoden *add* und *remove* Funktionalitäten zum Hinzufügen und Entfernen der Güter, wobei auf die entsprechenden Methoden, der zum *LoadType* kompatiblen *LoadSpaces* delegiert wird. Das gleiche Prinzip kommt bei den Methoden *getMaxAmountToAdd* und *getMaxAmountToRemove* zur Anwendung. Eine spezialisierte Form des *LoadManagers* ist der *WeightLimitedLoadManager,* bei dem die Summe aller verwalteten *LoadSpaces* eine definierte Grenze nicht überschreiten darf.

Neben den *LoadSpaces* verwaltet der *LoadManager* auch die in Abschnitt 5.4.1 beschriebenen Verladestellen (Klasse: *Terminal*). Über die Klasse *SupportedLoads* erhalten die *Terminals* eine maximale Durchsatzrate der unterschiedlichen *LoadTypes* in Einheiten pro Minute. Damit zum Verladen eines bestimmten *LoadTypes* passende *Terminals* gefunden werden können, bietet der *LoadManager* die Methoden *getOutputTerminals* und *getInputTerminals* an.

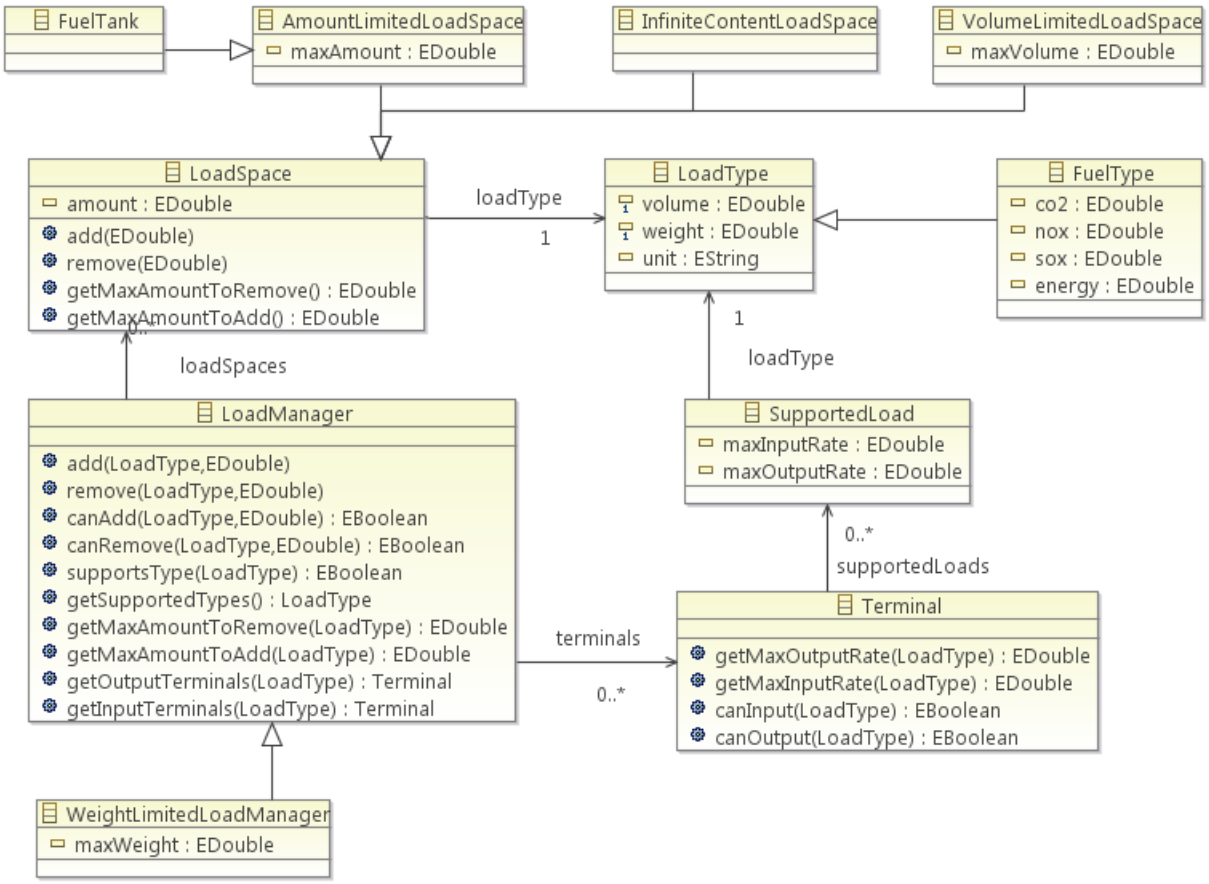

Abbildung 20: Verkehrsmodell LoadManager

6.2.4 Reservierung

In JASON wird Logik nicht kontinuierlich, sondern nur zu bestimmten Simulationszeitpunkten ausgeführt. Deshalb wird auch beim in Abschnitt 6.2.3 beschriebenen Güteraustausch nicht kontinuierlich der Inhalt der *LoadSpaces* verändert, sondern nur zu Beginn und Ende des Austauschs. Zwischen den beiden Simulationszeitpunkten können weitere Güteraustauschvorgänge beginnen bzw. enden und somit den Inhalt der *LoadSpaces* verändern. Zur Sicherstellung, dass jeder begonnene Güteraustauschvorgang erfolgreich abgeschlossen werden kann, werden Reservierungen (Klasse: *LoadReservation*) genutzt. Durch die Methoden *reserveAdd* und *reserveRemove* des *LoadManagers* werden *LoadReservations* für bestimmte Mengen eines *LoadTypes* erstellt. Sie gewähren den Verkehrsteilnehmern einen exklusiven Zugriff auf die reservierte Menge und garantieren somit die Durchführbarkeit des Güteraustausches über die gesamte Dauer. Der exklusive Zugriff wird dadurch sichergestellt, dass in den Methoden *getMaxAmountToAdd* und *getMaxAmountToRemove* des *LoadManagers* nicht nur der aktuelle Inhalt der *LoadSpaces* berücksichtigt wird, sondern auch die reservierten Mengen. Des Weiteren bietet die *execute*-Methode, welche auf die im *LoadManager* implementierte *executeReserveration*-Methode verweist, die Funktionalität, die reservierte Menge durch den *LoadManager* hinzuzufügen bzw. zu entfernen.

Durch die beschriebenen Funktionalitäten ist es möglich, Güter zu Beginn eines Austauschs über die *reserve*-Methoden zu reservieren und erst am Ende durch die *execute*-Methode einem LoadSpace hinzuzufügen bzw. zu entnehmen. Um eine Reservierung aufzuheben, ohne dabei den Inhalt der *LoadSpaces* zu ändern, existieren entsprechende *unreserve*-Methoden im *LoadManager* und in der *LoadReservation*.

Wie bereits in Abschnitt 5.4.1 beschrieben, muss eine Möglichkeit existieren, Verladestellen (Klasse: *Terminal*) für einen Güteraustausch exklusiv zu reservieren. Zur Umsetzung der Reservierung gibt es analog zu den bereits eingeführten *LoadReserverations* entsprechende *TerminalReservations*. Diese *TerminalReservations* können durch entsprechende *unreserve*-Methoden aufgehoben werden. Darüber hinaus ist es möglich, vor der Reservierung eines *Terminals*, mit der Methode *isReserved* zu überprüfen, ob ein bestimmtes *Terminal* belegt ist.

Für den Austausch von Gütern werden sowohl *LoadReservation* als auch *TerminalReservation* benötigt. Diese beiden Entitäten wurden nicht zusammengefasst, da es für den Verbrauch von Treibstoffen notwendig ist, diesen ohne eine Verladestelle zu reservieren. Die zuvor beschriebenen Zusammenhänge werden durch Abbildung 21 verdeutlicht.

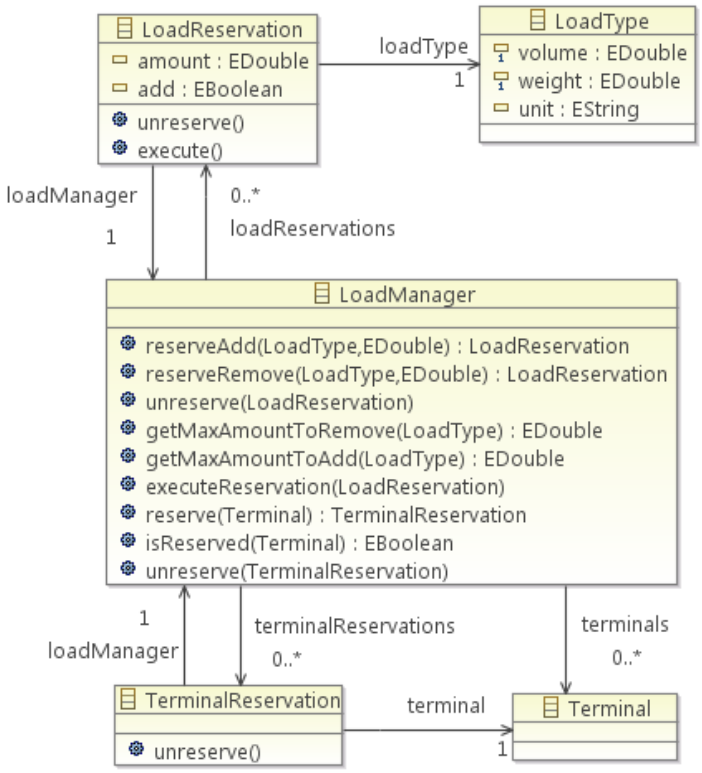

Abbildung 21: Verkehrsmodell Reservierung

6.2.5 Fahrzeug

Fahrzeuge (Klasse: *Vehicle*) bilden die in Abschnitt 6.2.2 eingeführten Verkehrsteilnehmer ab. Zum Zwecke des Treibstoffverbrauchs können Fahrzeuge über *Engines* verfügen, welche die notwendige Leistung zur Fortbewegung zur Verfügung stellen. Diese Leistung wird aus der Umwandlung der in Abschnitt 6.2.3 beschriebenen *FuelTypes* gewonnen, wobei jede *Engine* jeweils nur einen *FuelType* unterstützt.

Der Verbrauch einer *Engine* wird über die Methode *getFuelConsumption* berechnet. Sie nutzt hierfür den Brennwert des *FuelTypes* und die benötigte Eingangsleistung der *Engine* in Kilowatt. Die benötigte Eingangsleistung kann aus dem Wirkungsgrad der *Engine* und der Leistung in Kilowatt, die von der *Engine* zur Verfügung gestellt wird, berechnet werden. Diese Berechnung wird durch die Methode *getNeededPowerInput* durchgeführt. In unterschiedlichen Verkehrssystemen existieren verschiedene Einflussfaktoren, wie z. B. das Alter oder die Laufleistung einer *Engine*, die sich auf die Leistungskurve auswirken können. Aus diesem Grund ist die Methode *getNeededPowerInput* in domänenspezifischen Modellen auszuprägen und stellt somit einen Erweiterungspunkt des Verkehrsmodells dar. Der Verbrauch ist somit abhängig von den jeweiligen *Engines* des *Vehicles*. Des Weiteren hat eine *Engine* eine maximale Leistung in Kilowatt, die ebenfalls von den Einflussfaktoren abhängt und deshalb im domänenspezifischen Modell implementiert werden muss.

Die im vorherigen Textabschnitt beschriebenen *Engines* werden von den *Vehicles* zur Fortbewegung in dem Verkehrssystem genutzt. Die Ermittlung der Geschwindigkeit in Kilometer pro Stunde, die mit einer bestimmten Motorleistung erreicht wird, erfolgt über die Methode *getSpeed*. Als Umkehrfunktion hierzu berechnet die Methode *getPower* die Leistung in Kilowatt, die für eine bestimmte Geschwindigkeit benötigt wird. Der Zusammenhang zwischen Leistung und Geschwindigkeit hängt von verschiedenen Einflussfaktoren ab. Analog zu den Methoden der *Engine* stellen die Methoden *getSpeed* und *getPower* Erweiterungspunkte des Verkehrsmodells dar und müssen somit ebenfalls in einem domänenspezifischen Modell implementiert werden.

Die bei der Bewegung der Fahrzeuge durch die *Engines* verbrauchten *FuelTypes* werden durch einen *LoadManager* verwaltet. Der Zusammenhang zwischen *Vehicle*, *Engine* und *LoadManager* ist in Abbildung 22 dargestellt.

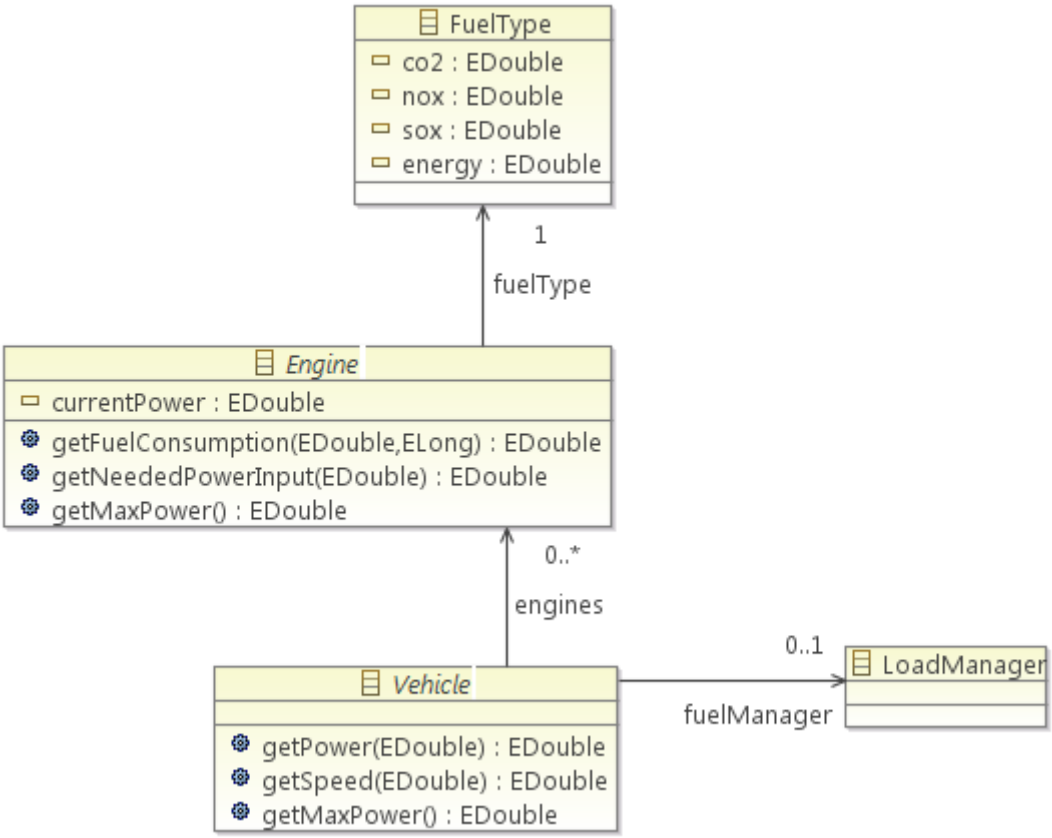

Abbildung 22: Verkehrsmodell Fahrzeug

6.2.6 Fahrplan

Durch die in den vorherigen Abschnitten beschriebenen Funktionalitäten sind *Vehicles* in der Lage, Güter zwischen Infrastrukturstandorten zu transportieren. Bei einem Transport müssen Güter zu bestimmten Zeiten an bestimmten Orten abgeholt bzw. geliefert werden. Mit den in Abbildung 23 dargestellten Klassen wurde ein Fahrplan modelliert, der dieses ermöglicht.

Der Fahrplan (Klasse: *TimeTable*) besteht aus einer Menge von Einträgen (Klasse: *TimeTableEntry*). Jeder dieser Einträge definiert, welche Aktionen zu welcher Zeit, an welchem Ort ausgeführt werden sollen. Da die für einen Güteraustausch notwendigen Parameter sich in domänenspezifischen Simulationen unterscheiden können, und auch andere Aktionen denkbar sind, gibt es für diese keine implementierte Klasse, sondern nur das Interface *Action*.

Neben diesen Einträgen für einmalige Aktionen existieren noch Einträge für sich regelmäßig wiederholende Aktionen (Klasse: *RepeatedTimeTableEntry*). Den Unterschied zwischen den beiden Eintragsarten in der Implementierung besteht darin, dass sich der *RepeatedTimeTableEntry* nach der Ausführung selbst mit seinem nächsten Ausführungszeitpunkt wieder in den *TimeTable* einträgt. Zur Festlegung der Ausführungszeitpunkte wurde die *CalendarIntervalUnit* modelliert, welche es ermöglicht, verschiedene Wiederholungszeiträume festzulegen.

Durch die *CalendarIntervalUnit* und das Attribut *interval* ist es z. B. möglich, einen Eintrag alle zwei Wochen wiederholen zu lassen. Diese Modellierung erlaubt u. a. die Abbildung des öffentlichen Personennahverkehrs (ÖPNV).

Der Fahrplan beinhaltet nur die an bestimmten Orten auszuführenden Aktionen und nicht die Fahrten zwischen diesen Orten, da diese implizit zwischen den Einträgen stattfinden müssen. Durch diese Modellierung wird u. a. das Einfügen eines neuen Eintrags zwischen zwei Vorhandene erleichtert, da andernfalls die Fahrten angepasst werden müssten.

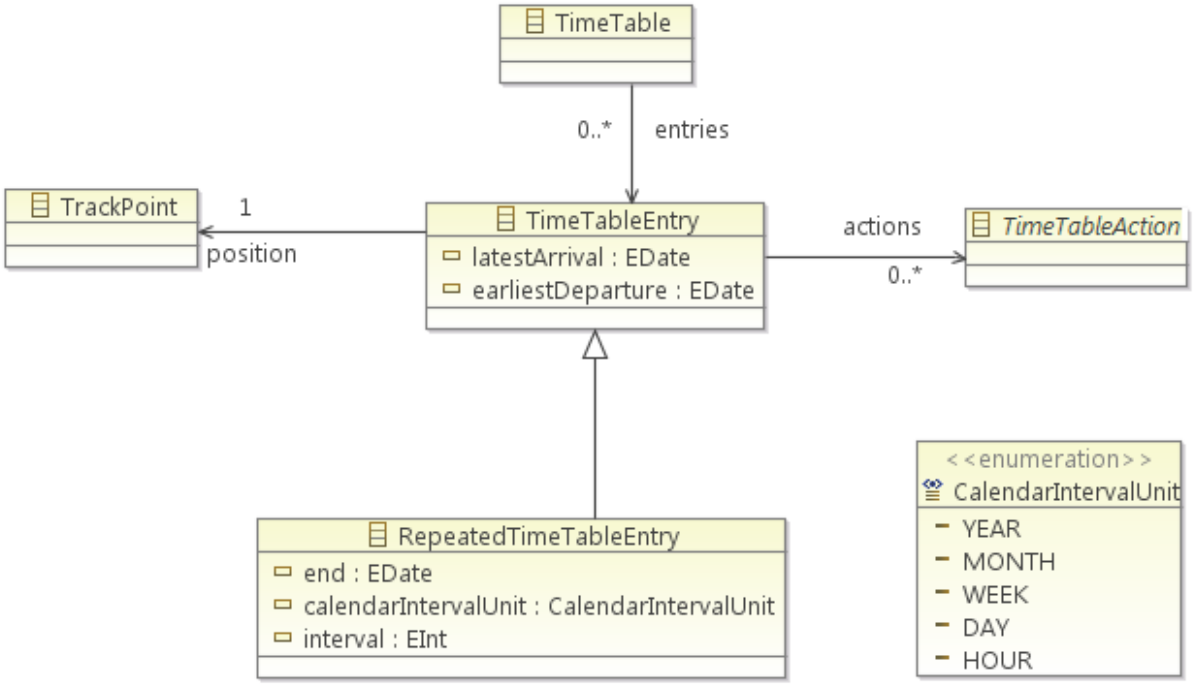

Abbildung 23: Verkehrsmodell Fahrplan

6.3 Implementierung der Verkehrssimulation

Dieser Abschnitt behandelt die Implementierung der Verkehrssimulation. Die Beschreibung baut dabei auf dem in Abschnitt 5.5 dargestellten Entwurf der Verkehrssimulation auf.

Das Multi-Agenten-Simulationsframework wurde um verkehrsspezifische Funktionalitäten erweitert, wodurch Simulationen von Verkehrssystemen verschiedener Domänen ermöglicht werden. Erweitert werden dabei die Simulation, Agenten, Behaviours und Services der JASON-Plattform.

Für das Simulieren des Verhaltens und der Interaktion der Verkehrsteilnehmer gibt es entsprechende Verkehrsagenten, wie z. B. die Fahrzeugagenten und die Infrastrukturstandortagenten. Diese Agenten verfügen über entsprechende Logik in Form von *Behaviours* und *Services*, um vorgegebene Aktionen auszuführen. Der *TrafficAgent* (siehe Abschnitt 6.3.2) ist

dabei eine Superklasse für alle Agententypen in einer Verkehrssimulation. Verkehrsteilnehmer sollen weiterhin die Möglichkeit besitzen, Güter auszutauschen. Aus diesem Grund ist der *LoadOwnerAgent* (siehe Abschnitt 6.3.3) implementiert. Fahrzeugagenten (siehe Abschnitt 6.3.5*)* sind Entitäten, die sich innerhalb eines Verkehrssystems bewegen. Der *InfrastructureAgent* (siehe Abschnitt 1.3.1.1) repräsentiert eine Entität, die *Services* für andere Agenten der Simulation bietet. Dabei handelt es sich hauptsächlich um Services für den Austausch von Gütern (Laden und Entladen). Der *Systemagent* (siehe Abschnitt 6.3.7) verwaltet das Netzwerk der Verkehrssimulation und bietet Dienste für die Positionsbestimmung von Agenten an. Während der Simulationslaufzeit dient er als eine Art „Fellow Pagers“, bei der die *TrafficAgenten* ihre Services registrieren und die Services anderer Agenten abfragen können.

In der Abbildung 24 ist die grundlegende Struktur der Verkehrssimulation aufgezeigt, welche in den folgenden Unterabschnitten detailliert behandelt wird.

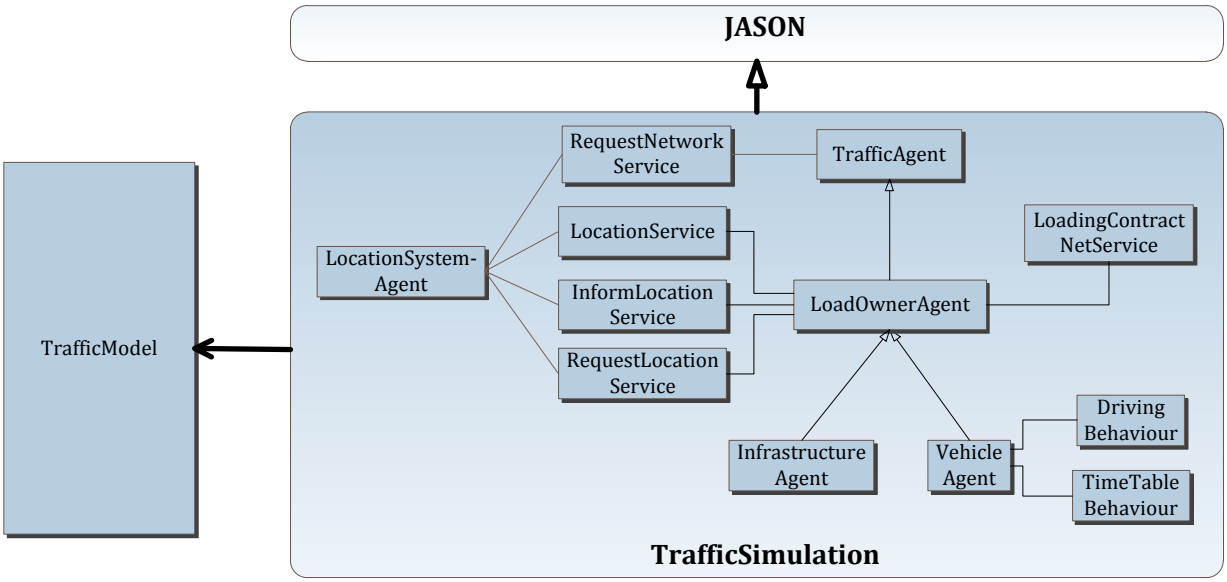

Abbildung 24: Übersicht der Verkehrssimulation

6.3.1 Verkehrssimulation

Die Verkehrssimulationsklasse (Klasse: *TrafficSimulation*) ist eine abstrakte Implementierung der Simulationsklasse aus JASON (Klasse: *Simulation*), die eine grundlegende Klasse für die Initialisierung und Ausführung von Verkehrssimulationen darstellt. In einer Verkehrssimulation lassen sich das Verhalten und die Interaktion von Verkehrsteilnehmern und Infrastrukturstandorten innerhalb einer bestimmten Simulationszeit, die durch einen Start- und Endzeitpunkt definiert ist, simulieren.

Zusätzlich hat eine Verkehrssimulation ein Netzwerk (Klasse: *Network*), auf dem sich die Verkehrsteilnehmer bewegen können. Das *Network* wird von einem Lokalisierungssystemagenten (Klasse: *LocationSystemAgent*) verwaltet. Der dazugehörige *LocationSystemAgent* wird über die *createDirectorySystemAgent*-Methode erzeugt und ersetzt damit den *DirectorySystemAgent* aus JASON.

6.3.2 TrafficAgent

Die abstrakte Klasse *TrafficAgent* ist eine Implementierung der Agentenklasse (Klasse: *Agent*) von JASON und bildet die Superklasse für alle Agenten in einer Verkehrssimulation. Der *TrafficAgent* ist dadurch charakterisiert, dass er durch ein *Network* erweitert wird, auf dem er sich bewegen und seine Aktionen durchführen kann.

Das *Network* der Simulation wird in der Initialisierung des *TrafficAgents* über den *RequestNetworkService* vom *LocationSystemAgent* angefordert.

6.3.3 LoadOwnerAgent

Die Klasse *LoadOwnerAgent* ist eine Erweiterung des *TrafficAgents*, die Logik zur Verwaltung von *LoadOwnern* (siehe Abschnitt 6.2.2) bietet. Da im Modell die Entitäten *Vehicle* und *InfrastructurePoint* zu einem *LoadOwner* generalisiert sind, stellt der *LoadOwnerAgent* analog die Superklasse für die Agentenklassen *VehicleAgent* und *InfrastructurePointAgent* dar.

Der *LoadOwnerAgent* besitzt eine Verarbeitungslogik für die Bestimmung und Bereitstellung des aktuellen Standorts der von ihm gesteuerten Entität. Standortinformationen sind für den Güteraustausch wichtig, da dieser nur an Standorten abgewickelt werden kann, an dem sich zwei *LoadOwner* zur selben Zeit befinden. Einen Überblick über alle Standortpositionen der *LoadOwner* im Netzwerk hat dabei der *LocationSystemAgent*. Dazu müssen die Verkehrsagenten ihre aktuelle Position bei ihm registrieren. Hierfür besitzt der *LoadOwnerAgent* entsprechende Logik. Diese Logik wird über den *InformLocationService* bereitgestellt. Hierüber können Agenten ihre Position an den *LocationSystemAgent* weiterleiten.

Für den Güteraustausch verwendet der *LoadOwnerAgent* den *LoadingContractNetService*, der den Austausch von Ladungen zwischen den *LoadOwnerAgenten* regelt. Dieser Service wird vom *LoadOwnerAgent* in der Initialisierung für alle *LoadTypes* registriert, die von ihrem *LoadManager* unterstützt werden.

6.3.4 LoadingContractNetService

Der *LoadingContractNetService* ist eine Implementierung des *ContractNetService* von JASON und bietet einen Service für den Austausch von Gütern zwischen *LoadOwnerAgents*.

Dieser Service ist dabei an das FIPA-Protokoll angelehnt. Er regelt den Austausch von Gütern durch die Vorgabe eines Ablaufs beim Güteraustausch. Der Ablauf eines Güteraustausches ist wie folgt:

Ein *Service*-anfragender *LoadOwnerAgent* fordert bei einem oder mehreren *Service*-anbietenden *LoadOwnerAgents* ein Angebot für einen Güteraustausch an. Der anbietende Agent kann ein Angebot zurücksenden, oder die Teilnahme an der Verhandlung ablehnen. Sobald der anfragende Agent die Antworten aller Anbieter erhalten hat, kann er ein Angebot annehmen und alle nicht angenommenen Angebote zurückweisen.

Abbildung 25 stellt die Interaktion zwischen den *LoadOwnerAgents* bei dem Güteraustausch dar.

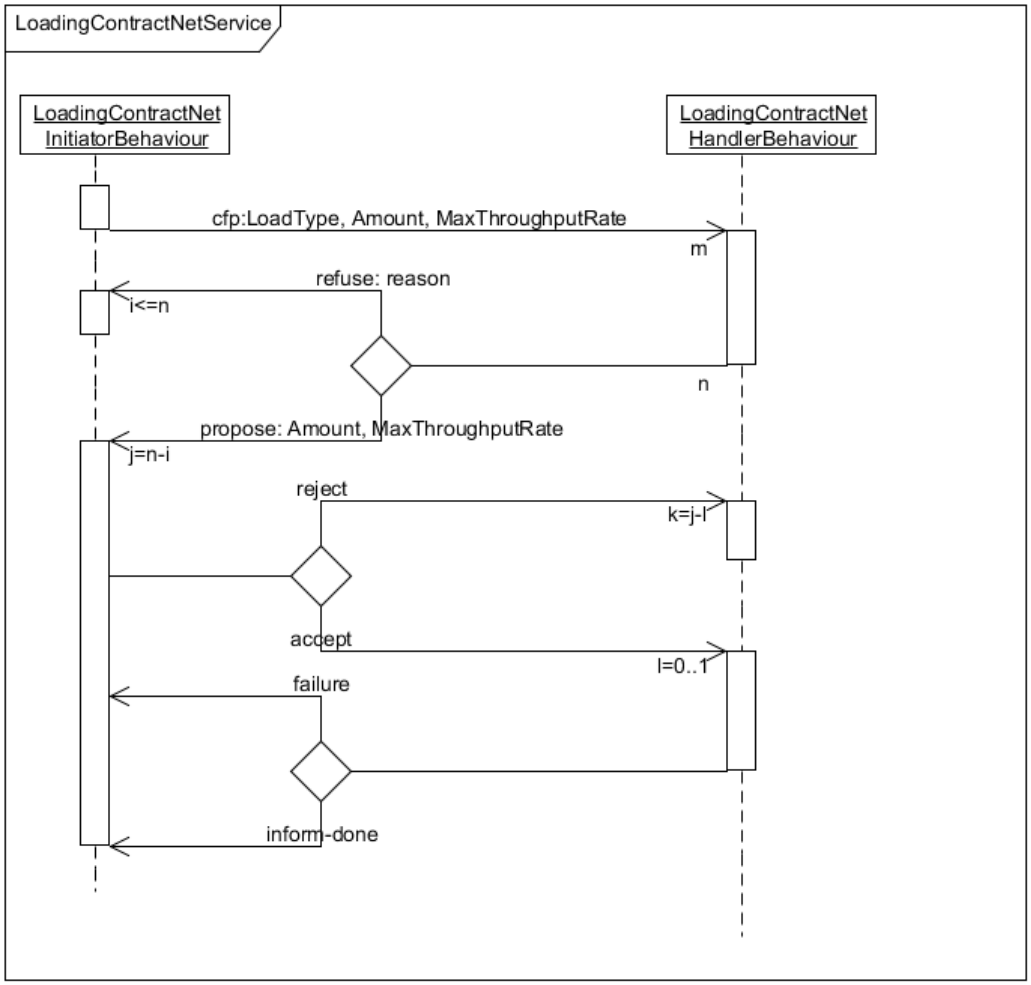

Abbildung 25: Ablauf eines Güteraustausches

Das *LoadingContractNetInitiatorBehaviour* bietet dem *LoadOwnerAgent* entsprechende Logik für die Aktionen auf der Service-anfragenden Seite an. Der LoadOwnerAgent verwendet diese Behaviours, um einen Austausch von Gütern zwischen zwei *LoadOwnerAgents* zu initiieren. Dazu reserviert er zunächst eine Verladestelle (Klasse: *Terminal*) für die Transakti-

on und eine Menge der auszutauschenden Güter (Klasse. *LoadType*). Die Reservierung selbst erfolgt über den *LoadManager* des jeweiligen *LoadOwnerAgents*, der entsprechende Reservierungsfunktionen bereitstellt. Nachdem die Reservierung erfolgreich durchgeführt wurde, erzeugt der Service-anfragende *LoadOwnerAgent* über die *createInitiationMessage*-Methode eine *CALL_FOR_PROPOSE*-Nachricht, mit der entsprechenden *LoadType*, die Menge und die maximale Durchsatzrate des Güteraustauschs mitgeteilt werden. Diese Nachricht sendet er an beliebig viele Service-anbietende *LoadOwnerAgenten*. Sobald der anfragende Agent die Antworten aller Anbieter erhalten hat, kann er ein Angebot über die *createProposeReplies*-Methode annehmen und andere Angebote zurückweisen.

Die benötigte Logik zur Bearbeitung der Anfrage stellt das *LoadingContractNetHandler-Behaviour* bereit. Der Service-anbietende *LoadOwnerAgent* verwendet dieses Behaviour, um auf die Service-Anfrage reagieren zu können. Wenn eine Service-Anfrage ankommt, überprüft der Agent, ob er diese Anfrage erwidern kann. Dazu überprüft er über die *getTerminal-Reservation*-Methode, ob ein Terminal frei ist und reserviert dieses, bzw. bricht die Verhandlung mit einer entsprechenden Nachricht ab. Ist ein Terminal frei, aber die Durchsatzrate entspricht nicht der Anfrage, kann ein Gegenangebot, mit den entsprechenden Durchsatzraten, abgegeben werden. Zudem wird durch die *getLoadReservation*-Methode überprüft, ob der angeforderte *LoadType* in ausreichender Menge verfügbar ist. Auch hier kann ein Gegenangebot mit einer anderen Menge abgegeben werden. Über die Methode *createCfpReply* erzeugt der Agent eine Nachricht zur Beantwortung der Service-Anfrage. Dabei kann er eine Ablehnung (*REFUSE*) senden, falls er diesen Service nicht bieten kann, bzw. möchte, oder eine Zustimmung (*PROPOSE*), sofern er den Güteraustausch bereitstellen kann bzw. möchte. Sobald der Service-anfragende *LoadOwnerAgent* das Angebot annimmt, beginnt der Anbieter mit dem Güteraustausch. Dazu werden, nach der für den Austausch benötigten Zeitspanne, die angebotenen Güter aus dem *LoadManager* entfernt und der anfragende Agent über den erfolgreichen Austausch informiert. Dieser führt seine Reservierung entsprechend auf seinem *LoadManager* aus.

6.3.5 VehicleAgent

Die abstrakte Klasse *VehicleAgent* ist eine Spezialisierung des *LoadOwnerAgents* und stellt einen Agenten dar, der in der Verkehrssimulation ein Fahrzeug (Klasse: *Vehicle*) steuert.

*VehicleAgent*s verfügen über ein *DrivingBehaviour*, um sich zwischen Streckenpunkten (Klasse: *TrackPoint*s) nach einer vorgegebenen Route zu bewegen. Weiterhin können *VehicleAgent*s durch Implementierung des *HasTimeTable*-Interfaces über einen Fahrplan (Klasse: *TimeTable*) verfügen. Die *TimeTable*s werden hierbei durch das *TimeTableBehaviour* abge-

arbeitet. Der Zusammenhang zwischen dem *VehicleAgent* und den *Behaviours*, die im Folgenden erklärt werden, ist in der folgenden Abbildung dargestellt:

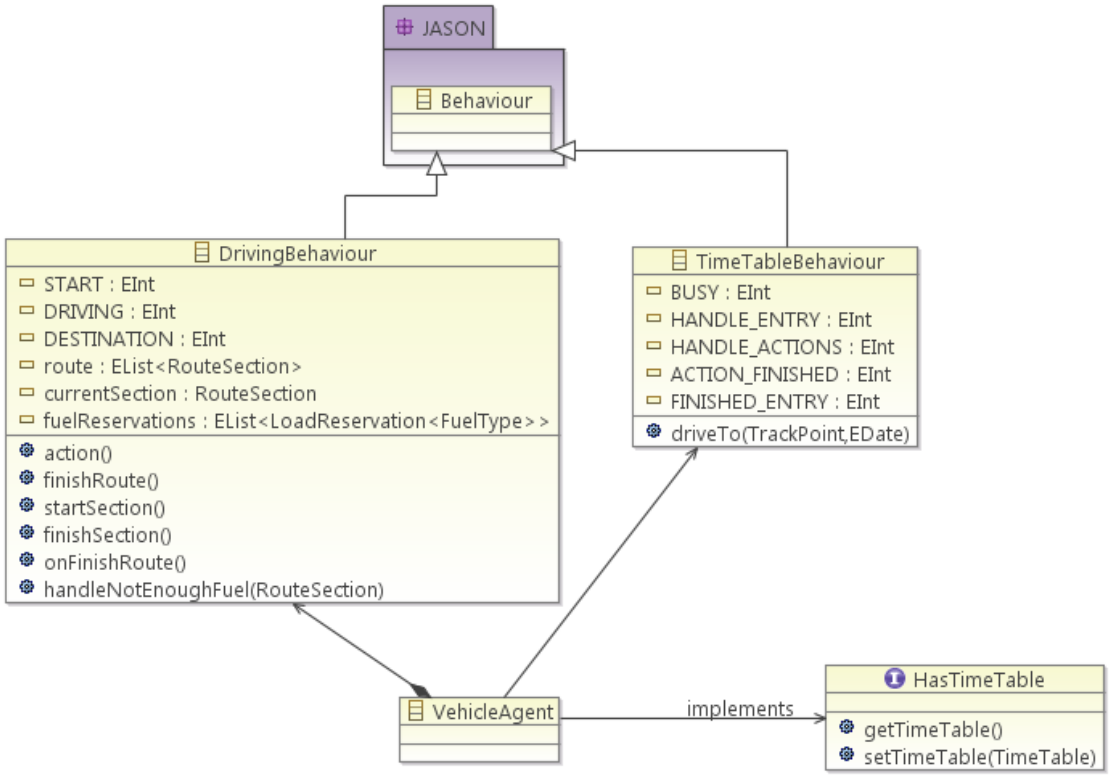

Abbildung 26: VehicleAgent und dessen Behaviours

1.3.1.1 DrivingBehaviour

Die abstrakte Klasse *DrivingBehaviour* dient als Superklasse für Behaviours, die Fahrzeuge (Klasse: *Vehicle*s), mit Hilfe von vorgegebenen Routen, zwischen verschiedenen Standorten bewegen können. Eine Route besteht dabei aus mehreren Streckenabschnitten (Klasse: *RouteSection*).

Für den Ablauf des *DrivingBehaviours* bestehen hierbei drei definierte Zustände: *START*, *DRIVING* und *DESTINATION*. Die Route wird als eine Liste von *RouteSection*s abgebildet, wobei die *RouteSection*s nacheinander abgefahren werden.

Hierbei ist es wichtig, den aktuellen Streckenabschnitt erfassen zu können, was durch das Attribut *currentSection* realisiert wird. Für jeden Streckenabschnitt wird weiterhin eine bestimmte Menge an Treibstoff benötigt, was durch das Attribut *fuelReservations* realisiert wird. Zudem wird die benötigte Zeit pro Streckenabschnitt berücksichtigt (*drivingDuration*).

In der Methode *action* ist die rudimentäre Funktionsweise des *DrivingBehaviour*s aufgeführt. Sofern der Zustand *DRIVING* vorherrscht, wird der aktuelle Streckenabschnitt durch Aufruf

der Methode *finishSection* abgeschlossen. In der Methode *finishSection* wird die benötigte Leistung des Motors (Klasse: *Engine*) für den jeweiligen Abschnitt zurückgesetzt und der benötigte Treibstoff über den *FuelManager* aus den Tanks entfernt. Sollte der Zustand nicht *DRIVING* sein, so wird der nächste Streckenabschnitt über die Methode *startSection* gestartet. Hierbei wird für den Abschnitt die benötigte Menge an Treibstoff reserviert und das *DrivingBehaviour* zur erwarteten Ankunftszeit neu eingeplant. Für den Fall, dass nicht genug Treibstoff vorhanden ist, wird die abstrakte Funktion *handleNotEnoughFuel* aufgerufen. Dort kann ein domänenspezifisches Verhalten für den Agenten implementiert werden. Zudem wird hier der *LocationSystemAgent* über die neue Position informiert, indem die Methode *scheduleInformNewLocation* der Klasse *VehicleAgent* aufgerufen wird.

Im Fall, dass die Route zu Ende ist, wird die Methode *finishRoute* aufgerufen. Zunächst wird der Zustand auf *DESTINATION* gesetzt und die Position des *VehicleAgent*s aktualisiert. Abschließend wird die abstrakte Methode *onFinishRoute* aufgerufen, die domänenspezifisch implementiert werden muss, um festzulegen, was nach dem Abschluss einer Route unternommen werden soll.

1.3.1.2 TimeTableBehaviour

Die abstrakte Klasse *TimeTableBehaviour* wird genutzt, um Aktionen eines Fahrzeugagenten (Klasse:*VehicleAgent*) in chronologischer Reihenfolge durchzuführen. Voraussetzung dabei ist, dass der Fahrzeugagent über einen *TimeTable* verfügt, was durch die Implementierung des *HasTimeTable*-Interfaces gesichert wird.

Hierbei kann für jeden Fahrplaneintrag (Klasse:*TimeTableEntry*) das Fahrzeug über die domänenspezifisch zu implementierende Methode *driveTo* zum jeweiligen Standort befördert, sofern es sich nicht bereits dort befindet. Daraufhin werden alle Aktionen, die in dem Fahrplaneintrag hinterlegt sind, ausgeführt. Sofern der früheste Abfahrtstermin nach Beendigung der Aktionen des Eintrags bereits überschritten wurde, wird der nächste Fahrplaneintrag abgearbeitet. Sollten alle Aktionen des Fahrplaneintrags vor dem frühesten Abfahrtstermin fertiggestellt worden sein, wird eine entsprechende Wartezeit eingeplant. Das *TimeTableBehaviour* besitzt wie das *DrivingBehaviour* definierte Zustände, die in der folgenden Tabelle beschrieben werden.

Tabelle 7: Zustände des *TimeTableBehaviours*

Zustand	**Beschreibung**
BUSY	Derzeit wird gefahren oder Aktionen durchgeführt.
HANDLE_ENTRY	Der nächste Eintrag des Fahrplans wird bearbeitet.

HANDLE_ACTIONS	Die Aktionen des aktuellen Eintrags werden gestartet.
ACTIONS_FINISHED	Alle Aktionen des aktuellen Eintrags wurden beendet.
FINISHED_ENTRY	Der aktuelle Eintrag ist abgearbeitet oder es wurde noch kein Eintrag behandelt.

6.3.6 InfrastructurepointAgent

Die abstrakte Klasse *InfrastructurePointAgent* ist wie der *VehicleAgent* eine Spezialisierung des *LoadOwnerAgenten* und die Basisklasse für Agenten, die in der Verkehrssimulation einen Infrastrukturstandort (*InfrastructurePoint*) verwalten. Ein *InfrastructurepointAgent* besitzt die Funktionalitäten des LoadOwnerAgenten und eine feste Position im Streckennetz.

6.3.7 LocationSystemAgent

Der Lokalisierungssystemagent (Klasse:*LocationSystemAgent*) ist eine Spezialisierung des *DirectorySystemAgenten* und dient der Verwaltung des *Networks* innerhalb der Verkehrssimulation. Er hat einen Überblick darüber, wo sich welche Agenten mit welchem Services befinden. Die Standorte und Agentenbeschreibungen werden hierzu durch die *LocatedAgentDescription* zusammengeführt. Dadurch kann der *RequestLocationService* Agenten durch den Vergleich mit einer *AgentDescription*-Vorlage an einem bestimmten Ort finden. Für die Umsetzung der Services verfügt der *LocationSystemAgent* über ein Verzeichnis (Klasse:*LocationKnowledgeBase*), das für die Lokalisierung von Agenten durchsucht werden kann.

Auf die vom *LocationSystemAgent* angebotenen Services wird im Folgenden eingegangen.

1.3.1.3 RequestNetworkService

Die Klasse *RequestNetworkService* bietet den Verkehrsagenten (Klasse: *TrafficAgent*) einen Service an, um das Netzwerk einer Verkehrssimulation anzufordern. Dieser Service ist an das FIPA-Protokoll angelehnt und regelt das Anfordern eines Netzwerks durch die Vorgabe eines Ablaufs:

Das Anfordern erfolgt dabei über den *LocationSystemAgent*, der das Netzwerk verwaltet. Für die Service-Anfrage stellt das *RequestNetworkInitiatorBehaviour* den *TrafficAgents* entsprechende Funktionalitäten bereit. Der *LocationSystemAgent* bearbeitet anschließend die Anfrage. Hierfür steht das *RequestNetworkHandlerBehaviour* zur Verfügung.

Der *TrafficAgent* nutzt das *RequestNetworkInitiatorBehaviour,* um eine Anfrage zu initiieren. Dabei erstellt er über die Methode *createInitiationMessage* eine Nachricht und sendet diese an den *LocationSystemAgent*. Bei einer erfolgreichen Anfrage wird eine Nachricht, die das angeforderte Netzwerk beinhaltet, vom LocationSystemAgent zurückgesendet. Der Agent bearbeitet diese Nachricht in der Methode *handleRequestInform*.

Zur Bearbeitung der Anfrage auf der anderen Seite der Kommunikation verwendet der *LocationSystemAgent* das *RequestNetworkHandlerBehaviour.* Dieses Behaviour ermöglicht dem *LocationSystemAgent* die Beantwortung der Anfrage. Dabei wird über die *createRequestReply*-Methode eine Nachricht mit dem Netzwerk der Verkehrssimulation erstellt.

1.3.1.4 InformLocationService

Dieser Service ermöglicht es Verkehrsagenten, die das *HasLocation*-Interface implementieren, dem Lokalisierungssystemagenten (Klasse:*LocationSystemAgent*) ihren aktuellen Standort zu übermitteln. Für die Übermittlung des Standorts wird vom jeweiligen Agenten das *InformLocationInitiatorBehaviour* genutzt. Die neu übermittelte Position wird wiederum vom *LocationSystemAgent* durch das InformLocationHandlerBehaviour verarbeitet.

Das *InformLocationInitiatorBehaviour* erstellt eine Nachricht, wobei die aktuelle Position des *TrafficAgent*s als Inhalt gesetzt wird. Im *InformLocationHandlerBehaviour* wird die Nachricht des *TrafficAgent*s entsprechend verarbeitet. Dabei wird über die Methode *handleNewLocation* der Standort des Agenten in der *KnowledgeBase* des *LocationSystemAgents* aktualisiert.

1.3.1.5 RequestLocationService

Der *RequestLocationService* ermöglicht es Agenten, die Position von Agenten beim *LocationSystemAgenten* abzufragen, die dort registriert sind. Um eine bestimmte *LocatedAgentDescription* anzufordern, benutzt ein Agent das *RequestLocationInitiatorBehaviour*. Hierbei wird ein *LocationRequestContent* verschickt, der die *AgentDescription*, welche als Vergleichsobjekt dient, enthält. Die Anfrage wird durch das *RequestLocationHandlerBehaviour* verarbeitet. Weiterhin besteht die Möglichkeit, den *PerimeterLocationRequestContent* zu nutzen, falls nach Agenten gesucht wird, die sich in der Nähe eines bestimmten Standortes befinden.

6.4 Implementierung des Editors

In diesem Abschnitt wird auf die Implementierung des im Entwurf beschriebenen Editors eingegangen. Zur Eingabe und weiteren Verarbeitung der Daten nutzt der Editor das im Entwurf genannte EditorModel. Um die Vorgänge im Editor in diesem Kontext besser erläutern zu

können, wird zunächst das EditorModel beschrieben und im Anschluss auf den Editor selbst eingegangen. Mit einem Sternchen (*) gekennzeichnete Klassen stammen aus einem Eclipse-Package.

6.4.1 EditorModel

Abbildung 27 zeigt die Klassenstruktur des EditorModels, welche im Folgenden näher beschrieben wird.

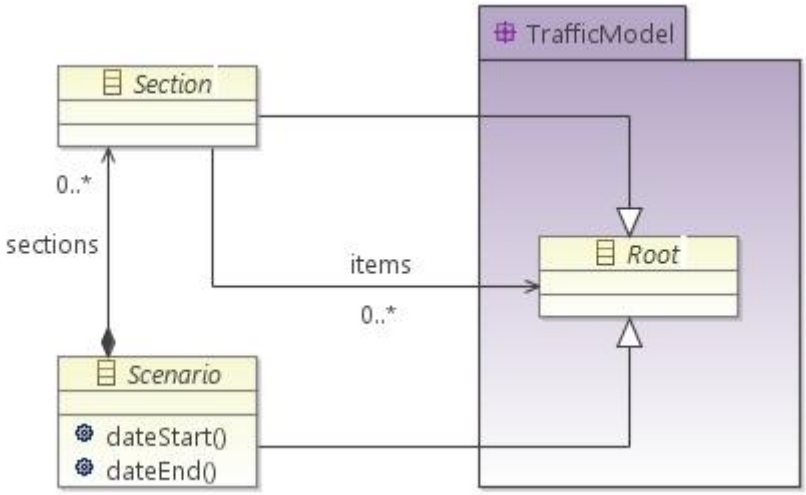

Abbildung 27: EditorModel

Die Klasse *Scenario* bildet die Grundkomponente der Simulations-Konfiguration. *Scenario* dient außerdem bei der Serialisierung in XMI als Root-Element. In ihr werden die Basisdaten, wie bspw. Name der Simulation, Startzeit (Attribut: *dateStart*) und Endzeit (Attribut: *dateEnd*) abgebildet. Zudem beinhaltet sie eine Liste von *Sections*.

Bei der Klasse *Section* handelt es sich um ein Objekt, das die Domänendaten kapselt. Spezifiziert wird die in der *Section* enthaltene Liste (*items*) durch einen generischen Typparameter. Dieser Typparameter muss *Root* implementieren. Somit können in einer *Section* die jeweiligen Domänenobjekte angelegt werden. Durch die Kapselung der Objekte in *Sections* wird das im Entwurf genannte Ziel der Gruppierung der Simulations-Bestandteile erfüllt, indem jede *Section* als ein eigenständiger Tab im Editor angezeigt wird.

Zur weiteren Verwendung dieses Models muss durch das Eclipse-EMF Plug-In der Edit-Code des Models erzeugt werden. Dadurch stehen bei der Nutzung des Models der daraus generierte Java-Code, inklusive *AdapterFactory** und *ItemProviderFactory** für das abstrakte Model, zur Verfügung. Beide Factories werden später zur Anbindung, der in dem Model vorhandenen Objekte, im Editor genutzt und bieten Funktionalitäten für die EMF-Operationen auf Objekten.

6.4.2 Editor

Da die Implementierung des Verkehrsmodells auf EMF basiert, wurde der Editor im Hinblick auf eine einfache Anbindung an die bestehende Architektur (siehe Abschnitt 5.2) implementiert. Hierzu wurde der Editor auf Grundlage des EMF-Editors entwickelt. Dabei bildet der Editor eine Eclipse RCP und kann dadurch verschiedene Perspektiven anzeigen. Dies ist besonders hilfreich, wenn Erweiterungskomponenten aus anderen Eclipse-RCP-Projekten in die bestehende Darstellung des Editors eingebunden werden sollen.

Die Implementierung des Editors besteht aus einer Vielzahl abstrakter Klassen, die u. a. durch das Eclipse-Plug-In vorgegeben sind.

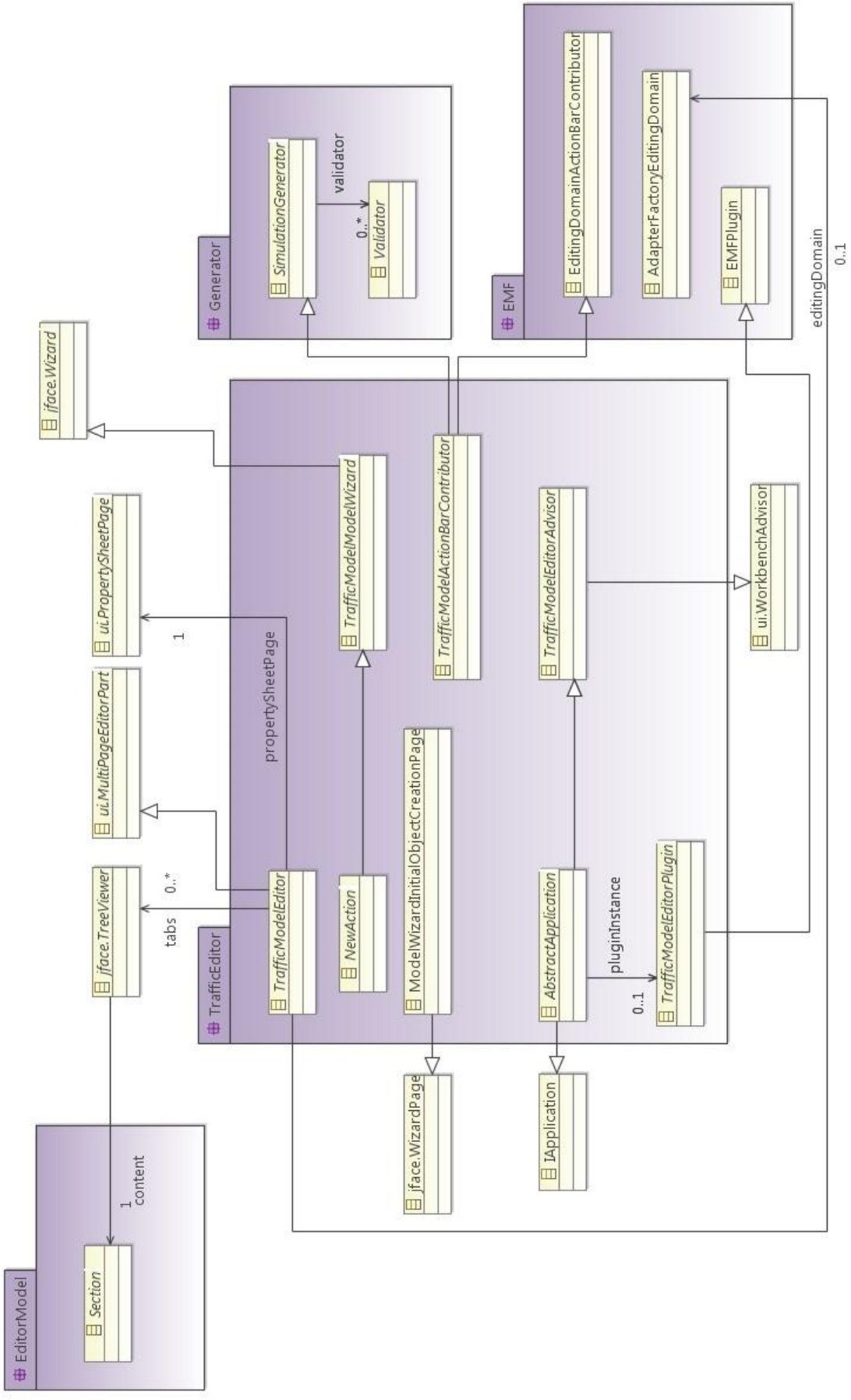

Abbildung 28: TrafficEditor

Abbildung 28 zeigt das Klassenkonstrukt des Editors, welches im Kern aus sieben Klassen gebildet wird, die untereinander Verwendung finden. Diese Klassen werden im Folgenden beschrieben.

6.4.2.1 AbstractApplication

Die *AbstractApplication* Klasse wird verwendet, um die Eclipse-RCP-Anwendung zu starten. Es erfolgt eine Ableitung von *IApplication*,* wodurch sie zu einem ausführbaren Eintrittspunkt für eine Application wird. Dabei werden der *TrafficModelEditorAdvisor* als Workbench sowie das *TrafficModelEditorPlugin* als RCP Plug-In verwendet. Beim Ausführen der Applikation werden das Plug-In und die Workbench mit dem User Interface initialisiert.

6.4.2.2 TrafficModelEditorPlugin

Das TrafficModelEditorPlugin ist das RCP-Plug-In des Editors. Diese Klasse erbt von der Klasse *EMFPlugin** des EMF-Frameworks. Durch die Ausprägung des Plug-Ins kann der Editor später als Equinox- und/oder OSGi-Bundle in andere Rich Client Applications eingebunden werden. Diese Eigenschaft wird benötigt, um den TrafficEditor zu einem domänenspezifischen Editor erweitern zu erweitern. Weiterhin ist es hierdurch auch möglich eine Erweiterung des TrafficEditors zu schaffen, die nicht domänenspezifisch ist, jedoch grundlegende Funktionalitäten erweitert. Ein Beispiel hierfür ist die Änderung der Darstellung von *Sections*. Des Weiteren verwaltet das Plug-In die internen Ressourcen der RCP, die in der plugin.xml zu spezifizierenden sind. Dabei kann es sich bspw. um abhängige Projekte oder Bilder handeln. Da es sich um das Plug-In des *TrafficEditors* handelt, muss ebenfalls in der domänenspezifischen Ausprägung eine Implementierung des *EMFPlugin** vorgenommen werden.

6.4.2.3 TrafficModelEditorAdvisor

Der *TrafficModelEditorAdvisor* erweitert den *WorkbenchAdvisor**. Er dient dazu, die Perspektiven der RCP zu initialisieren, die Menüeinträge anzulegen und deren Funktionalität auszuprägen. Dazu zählt u. a. der Öffnen-Dialog.

6.4.2.4 TrafficModelActionBarContributor

Der *TrafficModelActionBarContributor* initialisiert die Menüleiste der Editor-RCP. Die Klasse implementiert den *EditingDomainActionBarContributor** des EMF-Frameworks und verfügt damit automatisch über Undo-, Redo-, Cut-, Copy-, Paste-und Delete-Aktionen. Diese Funktionalitäten sind im TrafficEditor ebenfalls im Kontextmenü der einzelnen Sections eingebunden.

6.4.2.5 TrafficModelModelWizard

Der *TrafficModelModelWizard* leitet von *Wizard** ab und wird dazu verwendet, aus dem *EditorModel* und der domänenspezifischen Ausprägung des Verkehrsmodells eine neue Konfigurationsdatei zu erzeugen. Der *TrafficModelModelWizard* bietet dafür abstrakte Methoden an, die es ermöglichen beliebige Seiten in den Wizard einzufügen. Die entsprechenden Seiten des Wizards dienen dazu, den Benutzer bei der Erstellung einer neuen Konfigurationsdatei zu unterstützen. Eine Referenzimplementierung für eine Seite im Wizard stellt die Klasse *ModelWizardInitialObjectCreationPage* dar, die im Folgenden kurz eingeführt wird.

6.4.2.6 ModelWizardInitalObjectCreationPage

Dies ist eine Referenzimplementierung einer *WizardPage**, die im *TrafficModelModelWizard* zur Erstellung einer Konfigurationsdatei genutzt werden kann. Sie besteht aus drei visuellen Komponenten, einem Open-Dialog und zwei Combo-Boxen. Abbildung 29 zeigt die Oberfläche einer solchen *WizardPage**.

Der Open-Dialog wird dazu genutzt, einen Dateipfad anzugeben, unter dem die Konfigurationsdatei gespeichert werden soll. Eine Combo-Box dient dazu, ein Root-Element für die Konfigurationsdatei zu wählen. Das Root-Element ist typischerweise eine Ausprägung der *Section* des *EditorModels*. Die andere Combo-Box dient der Angabe des Zeichenformats, in dem die Konfigurationsdatei gespeichert werden soll.

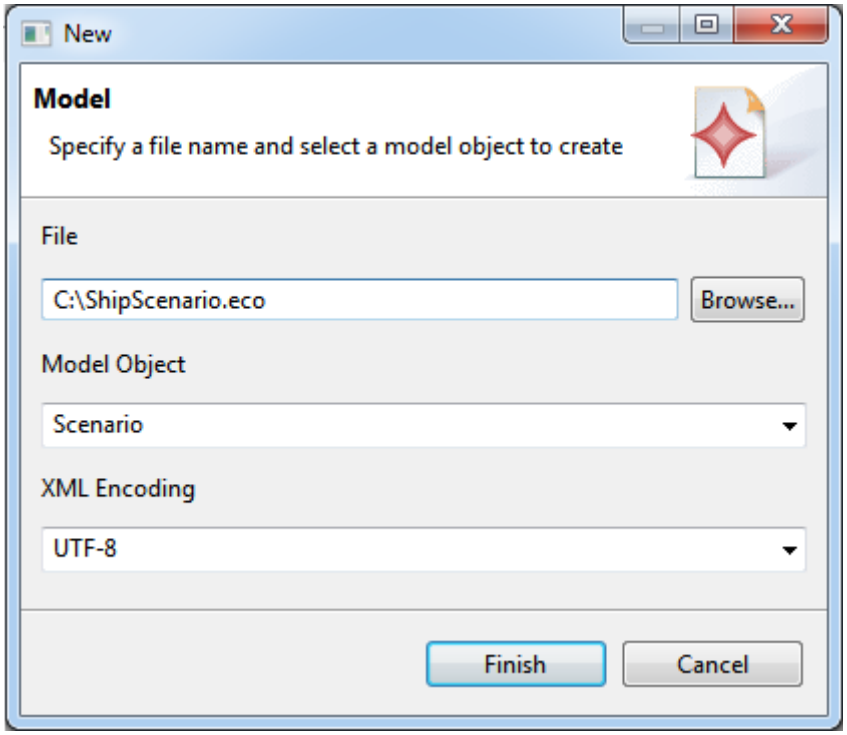

Abbildung 29: TrafficModelModelWizard mit ModelWizardInitialObjectCreationPage

6.4.2.7 TrafficModelEditor

Die Klasse *TrafficModelEditor* prägt *MultiPageEditorPart** aus. Der *TrafficModelEditor* bildet die Verknüpfung zum EditorModel. Der Edit-Code, der über EMF aus einem EditorModel erzeugt wird, kann in der späteren Ausprägung des Editors über die Implementierung der abstrakten Methode *addAdapterFactory* eingebunden werden.

Weiterhin beinhaltet der *TrafficModelEditor* Funktionalitäten, die das mit dem EditorModel implementierte Scenario- und Section-Design nutzen. Hierzu gehören Methoden, die die Initialisierung von neuen Objekten bzgl. einer Konfigurationsdatei handhaben und Komponenten bereitstellen, dessen visuelle Darstellung über den *TrafficModelEditorAdvisor* erfolgt. Solche Komponenten können bspw. *EMF-AdapterFactory*-Objekte und *ItemProvider**-Objekte sein. So kann in der später domänenspezifischen Ausprägung eine *Section* von Fahrzeugen via *setContent* an einen *TreeViewer** gebunden werden.

Eine beispielhafte Ableitung dieser Implementierung im Rahmen einer spezifischen Domänenausprägung wird im Abschnitt 7.3.4 zur Evaluation aufgezeigt.

6.4.3 Generator

Hauptbestandteil des Generators ist eine abstrakte Klasse *SimulationGenerator*. Dieser initialisiert eine Instanz eines Kopierers (Klasse: *TemplateCopier*), dessen Klasse im Konstruktor übergeben werden muss.

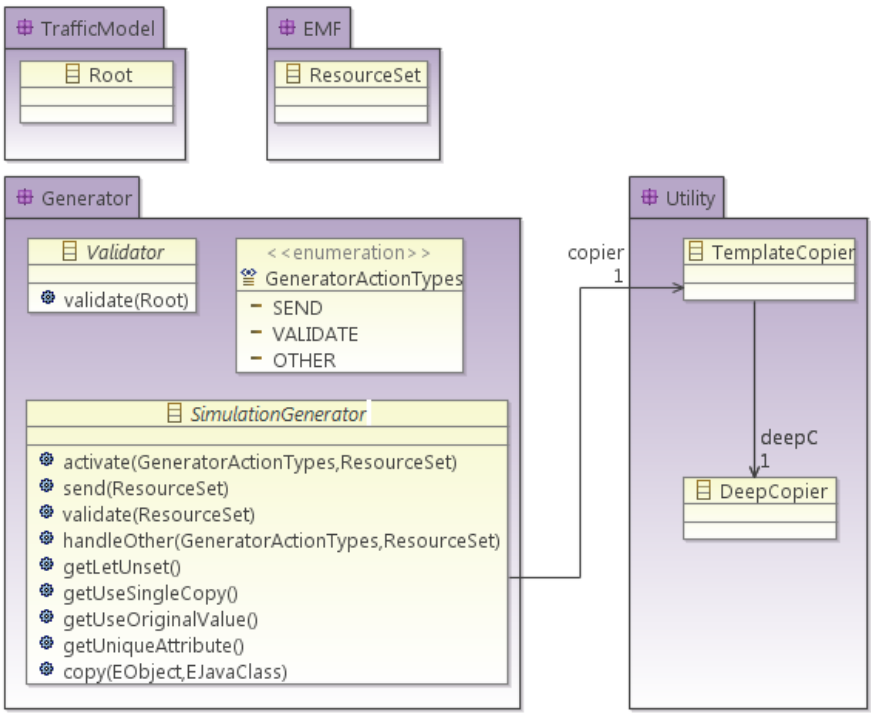

Abbildung 30: Generator

Wird der Generator extern durch die *activate*-Methode aufgerufen, wird dieser Kopierer jedes Mal neu initialisiert. Dazu wird eine neue Instanz erzeugt, der die erforderlichen Parameter übergeben werden. Diese Parameter für den *Copier* werden aus den Methoden *getLetUnset*, *getUniqueAttribute*, *getUseSingleCopy* und *getUseOriginalValue* bezogen und müssen in einer konkreten Ausprägung des Generators gegebenenfalls überschrieben und so auf das entsprechende Modell angepasst werden. Dadurch lassen sich schließlich die Funktionen zum Kopieren ohne weitere Einstellungen nutzen.

Der Generator hat standardmäßig zwei Funktionen, die über das Enum *GeneratorAction-Types* ausgewählt werden können. Das sind zum einen die Aktion *VALIDATE*, mit der das Modell auf semantische Fehler überprüft werden kann (siehe Abschnitt 6.4.4) und zum anderen die Aktion *SEND*, mit der das Modell entsprechend umgesetzt und an die Simulation übergeben wird. Zur möglichen Erweiterbarkeit wird vom Enum noch eine dritte undefinierte Aktion *OTHER* definiert. Damit über diese Konstante verschiedene Aktionen abgebildet werden können, enthält die Konstante ein String-Attribut, welches bei der Implementierung zur Unterscheidung verschiedener Funktionen genutzt werden kann. Alle drei Varianten werden im *SimulationGenerator* als abstrakte Methoden abgebildet und müssen in einem konkreten Modell entsprechend ausgeprägt werden. Der *validate*- und *send*-Methode werden dabei nur das *RessourceSet** aus dem Editor übergeben, da keine Auswertung der gewünschten Funktion mehr erforderlich ist. Der *handleOther*-Methode wird zusätzlich die aufgerufene Variable *OTHER* mit der Attribute übergeben, die in der Implementierung entsprechend ausgewertet werden muss.

6.4.4 Validator

Über die Methode *validate* des Generators ist es möglich, die Einträge in den jeweiligen Objekten zu prüfen. Dafür wurde für jede Entität des Verkehrsmodells eine korrespondierende *Validator*-Klasse entwickelt, die die jeweiligen Attribute validiert. Dabei wird sowohl das Vorhandensein eines Attributes als auch, sofern möglich, dessen Inhalt validiert. Bspw. wird bei einer *Connection* überprüft, ob alle Felder entsprechend gesetzt wurden. So muss die *Connection* sowohl einen Start- als auch einen Ziel-*TrackPoint* besitzen. Auch darf es nicht möglich sein, ungültige Werte in die Felder einzutragen. Erneut am Beispiel der *Connection* darf ein Starten der Simulation nicht möglich sein, sofern eine *Connection* keine positive Distanz zwischen dem Start und dem Ende hat.

Eine Validierung der Einträge erfordert jedoch einen Aufruf des Validators im konkreten Modell, da eine Ausprägung der *Sections* (siehe Abschnitt 6.4.1) erst im domänenspezifischen Modell erfolgen kann und damit auch erst die Inhalte der *Sections* festgelegt werden können. Einzig das Szenario wird im *SimulationGenerator* nicht vorvalidiert, da dieses bereits eine

Vielzahl erforderlicher Attribute enthält und das Root-Element der XML-Darstellung bildet. Zudem wird durch die Validierung eine Nachricht generiert, die die erfolgreiche Validierung bestätigt. Diese kann dem Benutzer über die Editoroberfläche dargestellt werden.

Die Klassen zur Validierung haben den Vorteil, dass, wenn das Verkehrsmodell (siehe Abschnitt 6.2) erweitert wird, eine Ableitung dieser Klassen nur noch die Validierung der Attribute der überschreibenden Klasse erfordert. Durch Generics wird zudem die Typsicherheit garantiert, sodass ein Fehler bereits vor der Ausführung verhindert werden kann.

7 Evaluation des Frameworks

Zur Evaluation des gesamten Frameworks wird ein Fallbeispiel herangezogen. Im Folgenden wird zuerst das Beispiel und die damit einhergehende Erhebung der notwendigen Daten vorgestellt, bevor im Anschluss daran die Umsetzung mit dem entwickelten Verkehrssimulationsframework erläutert wird. Abschließend erfolgt dann eine Evaluation der Ergebnisse des Fallbeispiels sowie eine Evaluation des gesamten Frameworks in Anbetracht der gestellten Anforderungen aus dem Kapitel Anforderungsanalyse.

7.1 Beschreibung des Fallbeispiels

In diesem Abschnitt erfolgt eine Beschreibung des Fallbeispiels, auf dem die Anwendung des Frameworks im Rahmen der Evaluation basiert. Thematisch wird hierbei die Einführung von Liquid Natural Gas (LNG) als alternativer Treibstoff in der Schifffahrt am Beispiel der Reederei Frisia behandelt. Zunächst wird hierbei im ersten Abschnitt die Thematik motiviert, bevor eine Darstellung der Reederei Frisia folgt. Im Anschluss daran werden die Emission Controlled Areas (ECA) erläutert.

7.1.1 Motivation

Der derzeit primär verwendete Treibstoff für Schiffe ist Schweröl. Schweröl ist ein Abfallprodukt der Erdölverarbeitung und erzeugt hohe Emissionen an Schwefel, Stickoxiden und Rußpartikeln. Aktuell verwenden ca. 90 % der größeren Schiffe diesen schadstoffreichen Treibstoff. Um den hohen Schadstoffausstoß zu reduzieren, wurden Emission Controlled Areas (ECA) eingeführt. In diesen Zonen ist der Schadstoffausstoß stark reglementiert. Zudem nehmen die Beschränkungen innerhalb dieser Zonen etappenweise weiter zu. In dem Nord-Ostseebereich ist die Einführung einer ECA bereits erfolgt.

Die ECA erlauben keinen Einsatz von Schweröl, wodurch in naher Zukunft alternative Kraftstoffe für Schiffsantriebe die Rolle von Schweröl als primären Treibstoff einnehmen müssen. Marinediesel erfüllt derzeit die Auflagen der ECA, ist jedoch verhältnismäßig teuer. Eine Alternative zum Marinediesel ist das kostengünstigere LNG. Dieses erfüllt durch geringe Emissionen auch zukünftig die verschärften Auflagen der ECA.

Um dieses Problem zu analysieren, ist eine Simulation erstellt worden, die die Anforderungen an die Schifffahrt betrachtet. Damit das Resultat der Simulation begutachtet werden kann, ist bei dem Fallbeispiel eine existierende Umwelt in die Simulation eingepflegt, die von einer Reederei (Frisia) für Personenbeförderung im Schiffslinienverkehr der Nordsee-Region

stammen. Die Forschung an alternativen Treibstoffen demonstriert den Nachhaltigkeitsgedanken und fördert die Entwicklung in der Region. Die hierdurch gewonnen Ergebnisse können den Aufbau einer Infrastruktur zur Nutzung von LNG fördern.

7.1.2 Reederei Frisia

Die in der Region ansässige Reederei Frisia (Aktiengesellschaft Norden-Frisia AG 2012) befährt das Schiffsliniennetz, welches als Grundlage zur Darstellung der Simulation genutzt wird. Mit dem Fallbeispiel soll exemplarisch dargestellt werden, wie sich der Betrieb der Fährschiffe mit LNG-betriebenen Maschinen im Gegensatz zu MDO-betriebenen Maschinen verhält.

Die Aktiengesellschaft Reederei Norden Frisia ist seit 1871 im Schiffsliniendienst von Norddeich nach Norderney und Juist tätig. Die Entfernung zwischen Norddeich und Norderney beträgt 10,2 km, die mit den entsprechenden Fähren in einer Stunde zurückgelegt werden kann. Die Strecke von Norddeich nach Juist ist mit 18,52 km etwas länger und benötigt daher etwa eineinhalb Stunden.

Die Reederei betreibt insgesamt 17 Schiffe, darunter unter anderem Autofähren, Fahrgastschiffe, Frachtfähren, Tonnenlegen und Shuttleschiffe. Fünf Fähren werden für den Linienverkehr (Personen- und Güterverkehr) zu den Nordseeinseln Norderney und Juist eingesetzt. Abhängig vom jeweiligen Schiffstyp und den damit verbundenen Schiffsrouten variieren sowohl Häufigkeit als auch Dauer der Fahrten. So haben bspw. Fahrgastschiffe, die zwischen Norddeich und den ostfriesischen Inseln verkehren einen festen Fahrplan. Die gesamte Frisia-Flotte nutzt Marinediesel als Treibstoff. Zur Betankung der Flotte wird eine Tankstation, die ausschließlich Marinediesel anbietet, auf der Insel Norderney genutzt.

7.1.3 Emission Controlled Areas

Dieser Abschnitt erörtert, was Emission Controlled Areas (ECA) sind und welchem Zweck sie dienen. Auflagen, die die ECAs betreffen, können hier nachgelesen werden (IMO 2012).

ECAs sind Zonen in der Schifffahrt, die besonderen Umweltauflagen unterliegen und von der International Maritime Organization (IMO) bestimmt werden. Insbesondere für den Ausstoß von Schwefeloxid (SO_X) (siehe Abbildung 33), Stickoxid (NO_x) (siehe Abbildung 32) und Rußpartikeln wurden und werden immer strengere Auflagen für die Schifffahrt festgelegt. In Abbildung 31 sind die aktuellen und geplanten globalen Umweltzonen aufgezeigt.

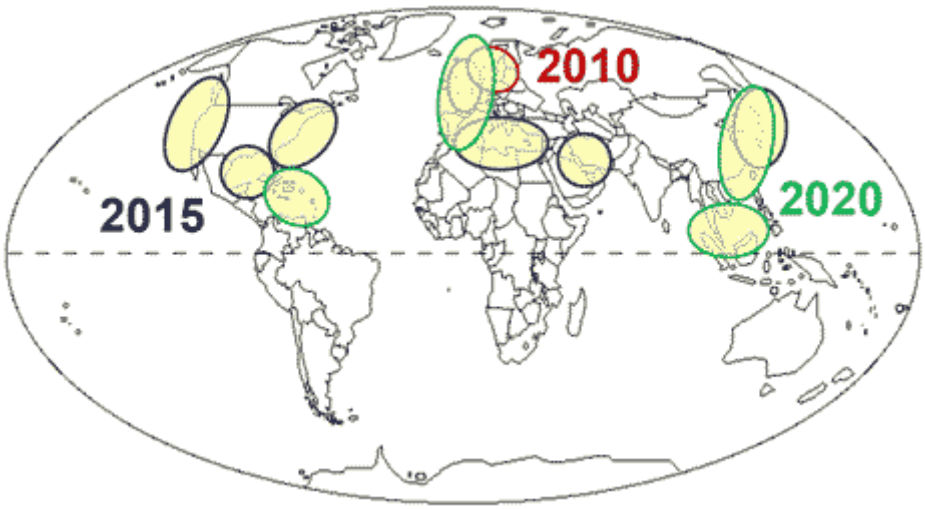

Abbildung 31: Aktuelle und geplante ECAs weltweit (Chart Ferox 2012)

Derzeit bestehen ECAs für den Bereich der Nord- und Ostsee sowie für West- und Ostküste der USA (2011). Weitere ECAs sind für den gesamten Mittelmeerraum und für die Südküste Japans geplant. Des Weiteren wird über weitere Zonen in Alaska, Hawaii, im südlichen Bereichs Nordamerikas, Australien, Südkorea sowie im Schwarzen Meer diskutiert.

Die genauen Regelungen zu den Emissionswerten zur Luftverschmutzung in der Schifffahrt hat die IMO im Rahmen des MARPOL-Abkommen in der Anlage 6 (Annex VI) festgehalten (IMO 2005). Dabei ist in Form eines langjährigen Stufenplans geregelt, wie viele Emissionen Schiffe sowohl global als auch in den separat betrachteten ECAs verursachen dürfen. Dieser Plan enthält im Groben drei Ebenen: Tier I, Tier II und Tier III**.** Jede dieser Ebenen spezifiziert einen Grenzwert bzgl. der Emissionen, der mit jeder Ebene zunimmt. Eine ECA umfasst nicht zwingend Auflagen zu allen drei Emissionstypen. So handelt es sich bei der ECA im Nord- und Ostseebereich nur um Richtwerte bzgl. des Schwefelgehalts. Die ECA im Küstenbereich Nordamerikas hingegen enthält Regelungen für Schwefel- und Stickstoffemissionen.

Die verschiedenen Emissionen bzw. deren Grenzwerte werden unterschiedlich betrachtet und berechnet, was im folgenden Abschnitt näher betrachtet wird.

7.1.3.1 Stickstoffemissionen

Richtlinien für die Stickstoffemission (NO_X) sind primär abhängig von der maximalen Drehzahl des Motors eines Schiffes. Die Drehzahl wird in Umdrehungen pro Minute (rounds per minute, rpm) gemessen. Die Grenzen für den Stickstoffausstoß werden in Tier I und II global betrachtet. Die Richtwerte für Tier III hingegen spielen nur in ECAs eine Rolle, in denen eine NO_X-Grenze vorherrscht (ECOpoint Inc. 2012).

Die Standards, die im Rahmen des MARPOL-Abkommens bzgl. der NO_x-Emission ausgearbeitet wurden, sind in der folgenden Tabelle und Grafik dargestellt.

Tabelle 8: Grenzwerte für NO_x-Emission

Tier	Datum	NO_x Limit, g/kWh		
		n < 130	130 <= n < 2000	n >= 2000
Tier I	2000	17.0	45 * n^-0.2	9.8
Tier II	2011	14.4	44 * n^-0.23	7.7
Tier III	2016	3.4	9 * n^-0.2	1.96

n := maximale Drehzahl des Motors

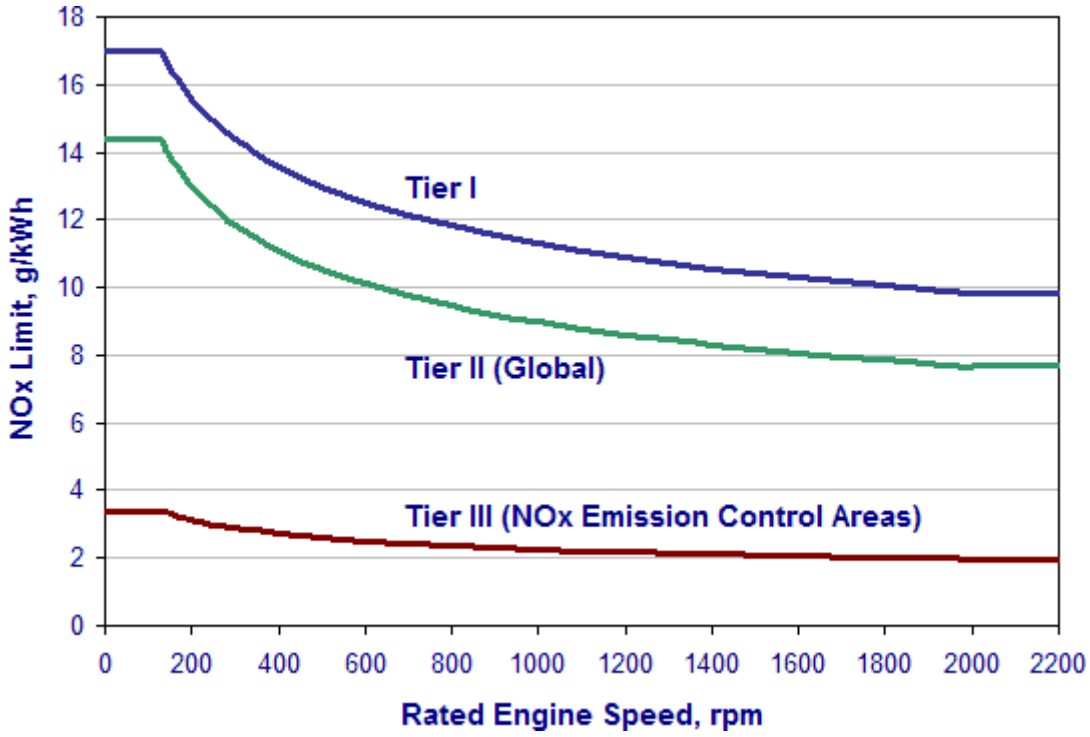

Abbildung 32: NO_x Grenzwerte

Es wird darauf abgezielt, den Tier II-Anforderungen mit Hilfe von Optimierung im Verbrennungsprozess gerecht zu werden. Hierbei kann z. B. das Zusammenspiel zwischen Einspritzzeitpunkt, Druck, Einspritzrate, Auslassventilsteuerung und Zylinderkompression im Motor verbessert werden. Diese Kompetenz liegt vor allem bei den Motorenherstellern vor, da es sich um eine grundlegende, technische Änderung handelt, die während des Motorenherstellungsprozesses berücksichtigt werden muss. Im Tier III hingegen wird weiterführende Technologie, wie z. B. die Wasserinduktion in den Verbrennungsprozess, die Abgasrückfüh-

rung oder der Einsatz von Katalysatoren notwendig, um die vorgegebenen Richtlinien einhalten zu können.

7.1.3.2 Schwefelemissionen

Im Gegenteil zur Stickstoffemission wird die Schwefelemission am Treibstoff selbst gemessen. Entscheidend dabei ist der Schwefelanteil im jeweiligen Treibstoff, um die Schwefelemission zu messen und kontrollieren zu können. Indirekt lässt sich dadurch auch der Ausstoß von Rußpartikeln berücksichtigen, wobei es diesbezüglich keine Grenzwerte gibt, der Rußpartikelausstoß jedoch stark vom Schwefelgehalt des Treibstoffs abhängt. Flüssige Treibstoffe für Schiffe, insbesondere Schweröl (HFO) und Marinedieselöl (MDO), liegen in unterschiedlichen Qualitätsformen vor, wobei der Schwefelgehalt der verschiedenen Sorten schwankt. Die Grenzwerte für den Schwefelgehalt für die jeweilige Ebene sind im Folgenden dargestellt. Anhand der Tabelle lässt sich erkennen, wie der Schwefelgehalt von Treibstoffen in den ECAs auf der einen Seite und auf globaler Ebene auf der anderen Seite im Zeitraum 2000-2020 gesenkt werden soll.

Tabelle 9: Schwefellimit im Treibstoff

Datum	**Schwefellimit im Treibstoff (% m/m)**	
	SO_x-ECA	**Global**
2000	1.5 %	4.5 %
2010	1.0 %	
2012		3.5 %
2015	0.1 %	
2020		0.5 %

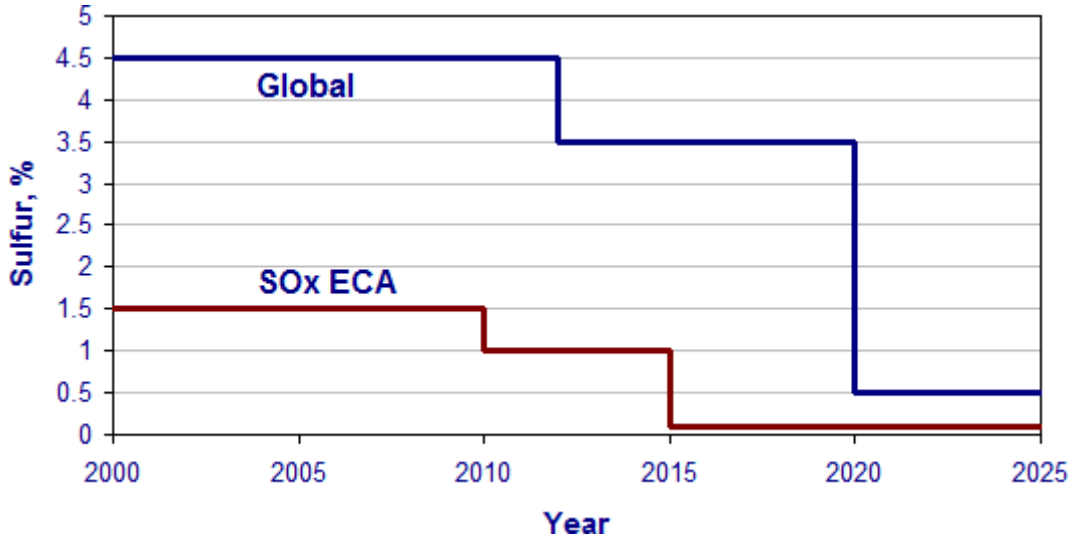

Abbildung 33: SO_x-Grenzwerte

Aus der Abbildung und der Tabelle kann entnommen werden, dass die Schwefelemission in den kommenden Jahren enorm gesenkt werden soll, sowohl global als auch in den ECAs. Global wird der Grenzwert von 4,5 % in den Jahren 2000-2010 auf bis zu 0,5% im Jahre 2020 gesenkt. Ein höheres Verhältnis weisen die Grenzwerte in den ECAs auf: Bei derzeitig 1,5 % Schwefelgehalt als Obergrenze soll dieser Wert ab 2015 auf 0,1 % gesenkt werden.

Um diese Richtlinien einhalten zu können, bedarf es einer signifikanten Umstellung hinsichtlich der Schiffsantriebstechnik. Eine Möglichkeit diesem Problem zu begegnen, ist der Einsatz von Abgas- und Filteranlagen, um den Emissionsausstoß zu senken. Allerdings ist dieser Weg für die Verwendung von Schweröl in ECAs in Zukunft unrealistisch. Der andere Weg wäre die Verwendung von alternativen und umweltfreundlicheren Treibstoffen wie MDO oder LNG.

7.1.3.3 Nationalpark Wattenmeer

In dem für das Fallbeispiel relevanten Teil der ECA gibt es weitere Anforderungen, die in die Simulation einfließen. Das Gebiet der Nordsee ist seit 1986 als Nationalpark (N. W. Wattenmeer 2012) geschützt und wurde 2009 durch die UNESCO (Deutsche UNESCO-Kommision 2012) zum Weltnaturerbe erklärt. Seit März 2010 umfasst das Gebiet 345.000 Hektar. Es ist der zweitgrößte Nationalpark Deutschlands.

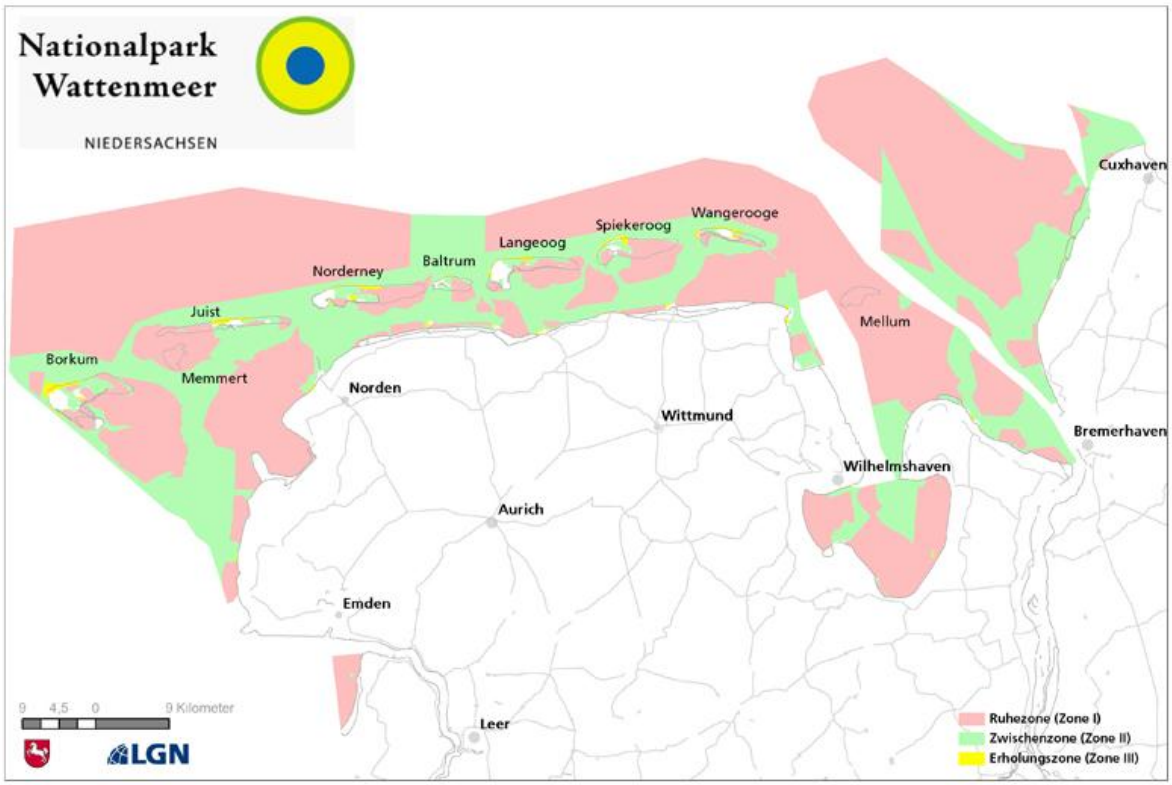

Abbildung 34: Nationalpark Niedersächsisches Wattenmeer (N. W. Wattenmeer 2012)

Im Nationalpark existiert ein Zonenkonzept, welches in drei Zonen eingeteilt ist. Durch diese Zonen führen Bundeswasserstraßen, die durch die „Verordnung über das Befahren der Bundeswasserstraßen in Nationalparken im Bereich der Nordsee (NPNordSBefV)“ vom 03.09.1997 reglementiert werden. Dieses Gesetz legt u. a. fest, dass Wasserfahrzeuge im Bereich der Nordsee eine Geschwindigkeit von 12 Knoten nicht überschreiten dürfen.

7.2 Datenerhebung

Als vorbereitende Maßnahme für die Anwendung der Simulation auf das Fallbeispiel mussten spezifische Daten beschafft werden. Im ersten Abschnitt werden hierbei die unterschiedlichen Treibstoffe, die in der Schifffahrt verwendet werden, dargestellt und verglichen. Weiterhin wird im Anschluss daran auf den Tankprozess in der Schifffahrt eingegangen, da dieser in der Simulation berücksichtigt wird.

Darauf folgend wird der Zusammenhang zwischen der Geschwindigkeit eines Schiffes und der dafür benötigten Leistung (in Kilowatt) ermittelt. Anhand dieser Informationen wird eine Geschwindigkeit-Leistung-Relation für die Fähren der Frisia-Flotte bestimmt. Ein weiterer wichtiger Punkt sind die spezifischen Schiffsdaten der Frisia-Flotte. Da insbesondere die Einführung eines LNG-Antriebs untersucht wird, um Verbrauchsdaten und verursachte Emis-

sionen der Antriebskonzepte vergleichen zu können, sind die spezifischen Motorendaten der Schiffe für die Ergebnisse ausschlaggebend. Des Weiteren sind der Schiffslinienverkehr und die Fahrpläne, in denen die Fahrtzeiten und Strecken hinterlegt sind, wichtig für das Fallbeispiel. Die Fahrpläne sind die Grundlage für das abgeleitete Streckennetz in der Simulation.

Auf Grundlage der Datenerhebung und der daraus resultierenden Erfahrungen und Informationen werden das Verkehrsmodell und die Verkehrssimulation um schiffsspezifische Aspekte erweitert.

7.2.1 Treibstoffe in der Schifffahrt.

7.2.1.1 Schweröl

Schweröl (HFO) ist im engeren Sinne ein Abfallprodukt der Rohölverarbeitung und damit ein fossiler Brennstoff. Je nach technischem Stand der Raffinerie bleiben bei dem Verarbeitungsprozess von Rohöl ca. 40 % als Schweröl über. Andere Bezeichnungen für Schweröl sind Bunkeröl, HFO, RFO, IFO oder Bunker C, wobei die internationale Handelsbezeichnung „Marine (Residual) Fuel Oil“ (MFO) ist. HFO ist ein sehr zähflüssiger Treibstoff und liegt in unterschiedlichen Qualitäten vor, die sich insbesondere im Schwefelgehalt unterscheiden (Douvier 2008).

Ein Großteil der heutigen Schiffe wird durch Schweröl angetrieben. Ein wichtiger Vorteil hierbei liegt im Preis für den Treibstoff: Schweröl ist im Vergleich zum höherwertigen und umweltfreundlichen Marinediesel sehr günstig. Demgegenüber steht der hohe Schadstoffgehalt von Schweröl, wodurch beim Verbrennungsprozess ein hohes Maß an Schwefel- und Stickoxiden sowie Rußpartikeln ausgestoßen wird. Somit ist die Zukunftsfähigkeit von HFO fraglich, zumal die Umweltrichtlinien für Schwefel- und Stickstoffemissionen stark verschärft werden und mit der Verwendung von Schweröl trotz Reduzierung der Emissionen kaum einzuhalten sind.

7.2.1.2 Marinedieselöl

Marinedieselöl (engl.: Marine Diesel Oil, MDO) ist ein Kraftstoff für Schiffsdieselmotoren und ein Gemisch aus verschiedenen Destillaten, die während der Erdölverarbeitung entstehen. Die internationale Bezeichnung ist Marine (Destillate) Fuel Oil. Wie auch bei Schweröl wird beim Marinedieselöl zwischen verschiedenen Sorten und Qualitätsgraden differenziert, die sich in Viskosität, Dichte und Flammpunkt unterschieden (Douvier 2008).

MDO ist im Gegensatz zu HFO umweltfreundlicher und für zukünftige Umweltauflagen, wie sie in den ECAs eingeführt werden, geeignet. Allerdings handelt es sich auch um einen höherwertigen und teureren Treibstoff. Bei schwankenden Preisen für Schweröl und Marinedie-

sel ist MDO fast doppelt so teuer wie HFO und ist daher aus ökonomischer Sicht nur eine Alternative.

7.2.1.3 Liquid Natural Gas

Flüssigerdgas (engl.: Liquid Natural Gas, LNG) wird als vielversprechende Alternative als Treibstoff für Schiffe gehandelt. LNG erlangt durch eine Abkühlung auf -160 °C den flüssigen Aggregatszustand und hat in dieser Form nur 1/600stel des Volumens der Gasform. Die Energiedichte von flüssigem Erdgas beträgt nur ca. 60 % im Vergleich zum Marinediesel. Dies führt zwangsläufig dazu, dass größere Tanks installiert werden müssen, um auf das gleiche Maß an Energieerzeugung zu kommen und gleiche Reichweiten zu erzielen. Logische Schlussfolgerung ist hierbei die Einbuße von Ladefläche (ca. 3 %), wobei jedoch die meisten Schiffe ohnehin nicht voll beladen ihre Routen fahren (Olesen, et al. 2009).

In flüssiger Form ist LNG weder giftig, explosiv oder ätzend. Einzig die tiefe Temperatur ist für Stahl ein Problem, dem mit dem Einsatz von geeigneten Materialien entgegengewirkt werden kann. Weiterhin müssen entsprechende Sicherheitssysteme vorhanden sein, um einen möglichen Gasaustritt signalisieren zu können. Im Falle eines Gasaustrittes liegen die Gefahren in einer möglichen Explosion oder Ersticken beteiligter Personen, sobald LNG seine Gasform annimmt.

Im Vergleich zu den fossilen Brennstoffen HFO und MDO ist LNG ein sehr umweltfreundlicher Treibstoff, wodurch Emissionen stark gesenkt werden können. Da LNG keine Schwefelanteile hat, kann dieser Ausstoß um bis zu 100 % gesenkt werden. Die Emission von Stickoxiden ist um 85 % reduziert und der CO^2-Ausstoß um 30 %. Weiterhin weist LNG eine sehr geringe Partikelemission auf. Aufgrund dieser Fakten ist LNG als Treibstoff gut geeignet, um den zukünftigen Umweltrichtlinien in den ECAs gerecht zu werden (Olesen, et al. 2009).

7.2.1.4 Vergleich der Treibstoffe

In diesem Abschnitt werden die Treibstoffe MDO und LNG miteinander verglichen. Aufgrund der Tatsache, dass die Reederei Frisia kein Schweröl für den Fährverkehr nutzt und dieser Treibstoff somit nicht in der Simulation berücksichtigt wird, erfolgt an dieser Stelle überwiegend der Vergleich von MDO und LNG. Ausschließlich bezüglich der verursachten Emissionen wird Schweröl mit den anderen Treibstoffen in Bezug gestellt.

Für die Eingabe in der Simulation sind vor allem die Energiewerte und die Emissionswerte der beiden Treibstoffe relevant.

Tabelle 10: Energiewerte der Treibstoffe (Rolls-Royce 2011)

Treibstoff	Unterer Heizwert(MJ/Kg)	Dichte (Kg/m³)	Energiedichte (MJ/m³)
MDO	42,7	900	38,430
LNG	54,7	442	24,177
LNG / MDO Energiedichteverhältnis (gleiches Volumen): 1,6			

In Tabelle 10 sind die Energiewerte von LNG und MDO dargestellt. MDO weist dabei eine deutlich höhere Dichte auf und besitzt eine höhere Energiedichte. Das typisch leichte Gewicht von LNG wird an der geringeren Dichte erkennbar – demgegenüber steht jedoch die geringere Energiedichte dieses Treibstoffs. Somit wird mehr LNG als MDO benötigt, um die gleiche Menge an Leistung erzeugen zu können.

Zur Grundlage der Emissionsermittlung ist folgende Tabelle genutzt worden, in der die spezifischen Werte pro Treibstoff aufgelistet sind. In dieser Tabelle wird ersichtlich, welche Emissionsvorteile LNG gegenüber den anderen Treibstoffen besitzt und somit als umweltfreundliche Alternative in der Schifffahrt gilt. Gasoil ist hierbei eine spezielle Sorte von Marinedieselöl, welches im Schwefelgehalt stark reduziert ist.

Tabelle 11: Emissionswerte nach Treibstoffen

Fuel Type	SO_x (g/kWh)	NO_x (g/kWh)	PM (g/kWh)	CO2 (g/kWh)
Residual oil 3,5 % sulphur	13	9 - 12	1,5	580 - 630
Marine diesel Coil, 0,5 % sulphur	2	8 - 11	0,25 - 0,5	580 - 630
Gasolin, 0,1 % sulphur	0,4	8 - 11	0,15 - 0,25	580 - 630
Natural gas (LNG)	0	2	~0	430 - 480

Resultierend aus Informationen stellt Tabelle 12 dar, welche Angaben bezüglich der Treibstoffe für die Simulation relevant sind und übernommen wurden.

Tabelle 12: Treibstoffdaten für die Simulation

Bezeichnung	Maßeinheit	LNG	MDO
Energiegehalt	kWh	6.715	9.8
Gewicht	kg	0.44	0.88
NO_x	g	2.0	9.5
SO_x	g	0.0	2.0
CO^2	g	455.0	605.0
Die angegebenen Werte gelten für jeweils einen Liter LNG/MDO			

Wie bei der Energie zu sehen, besitzt LNG nicht den gleichen Energiegehalt wie MDO. Gegenüber einem Liter Diesel werden demnach 1,67 Liter LNG benötigt, um das gleiche Maß an Energie zu erzeugen. LNG hat im Vergleich zu Diesel eine geringere Dichte und wiegt somit weniger.

7.2.2 Tankprozess

Aufgrund der unterschiedlichen Eigenschaften der Treibstoffe (insbesondere Dichte und Energiegehalt) müssen Schiffe entweder unterschiedlich oft betankt werden oder Treibstofftanks mit größerem Volumen verbaut werden, um die entsprechenden Reichweiten zu erzielen. Durch den Einbau von Tanks mit größerem Volumen reduziert sich die Ladefläche bzw. der Nutzraum des Schiffes.

Bei der Betankung eines Schiffes werden die Möglichkeiten einer fest installierten Station und eines Tanklasters betrachtet. Diese beiden Varianten unterscheiden sich insbesondere bezüglich der Durchsatzrate beim Tanken, was direkte Auswirkungen auf die Dauer des Tankvorgangs hat. Da es sich bei beiden Treibstoffen um Flüssigkeiten handelt, wird an dieser Stelle angenommen, das LNG und MDO die gleichen Durchsatzraten beim Tankvorgang aufweisen. In Tabelle 13 sind Daten bezüglich des Volumens und der Durchsatzrate der unterschiedlichen Varianten dargestellt.

Tabelle 13: Tankgeschwindigkeit per LKW oder Station

Tankdaten	**Tanklaster**	**Tankstation**
Volumen	55 m³ 55.000 Liter	1000 m³ 1.000.000 Liter
Tankgeschwindigkeit pro Stunde	30 m³ / Stunde 30.000 Liter / Stunde	100 m³ / Stunde 100.000 Liter / Stunde
Tankgeschwindigkeit pro Minute	0,5 m³ / Min 500 Liter / Min	1,66 m³ / Min 1666 Liter / Min

In Rotterdam gibt es Bunkermöglichkeiten für LNG Tankschiffe, die evtl. als Bezugsquelle (Gate terminal 2012) für Deutsche Reedereien nutzbar wäre. Kosten für LNG können aber nur hypothetisch angenommen werden. Zusätzlich steuerliche Belastungen können hinzukommen.

Aufgrund der geringen/ nicht vorhandenen Nutzung von LNG als Treibstoff in der Bundesrepublik Deutschland gibt es noch keine Regularien für die Nutzung. Aktuell müssen Schiffe mit LNG-Antrieb eine Sondergenehmigung für jeden Hafen haben, der angefahren wird.

7.2.3 Relation zwischen Geschwindigkeit und Leistung

Ein Ansatz zur Ermittlung der Leistung, die für eine bestimmte Geschwindigkeit eines Schiffes aufgebracht werden muss, ist in dem Paper „Ship emissions and air pollution in Denmark“ (Olesen, et al. 2009) dargestellt. Dieser Ansatz besagt, dass der Zusammenhang zwischen der Geschwindigkeit und Leistung durch die folgende Formel herstellen lässt:

Formel 1: Berechnung der benötigten Leistung (Olesen, et al. 2009)

$$P_{cal} = P_s \times \left(\frac{V_{obs}}{V_s}\right)^3, \quad mit\ P_{cal} := benötigte\ Maschinenleistung,¶$$
$$P_s := Service\ Power\ (Ø\ Motorleistung\ in\ KW)¶$$
$$V_{obs} := \ Festgestellte/gemessene\ Geschwindigkeit¶$$
$$V_s := Service\ Speed\ (Ø\ Geschwindigkeit\ bei\ Normalbetrieb\ )$$

Die Formel stellt eine generalisierte bzw. allgemeine Geschwindigkeit-Leistung-Relation eines Schiffes dar. Für Schiffe mit großer Motorenleistung (z. B. Containerschiffe, Tanker etc.) liefert diese Formel insgesamt annährend gute Ergebnisse. So ist aus der Abbildung 35 zu

erkennen, dass der Verlauf der Geschwindigkeit-Leistung-Kurven der Containerschiffe und der Tanker, die auf Basis von einer Regressionsanalyse geschätzt werden, der oben aufgeführten Formel (bzw. ihrer Umkehrfunktion) entspricht.

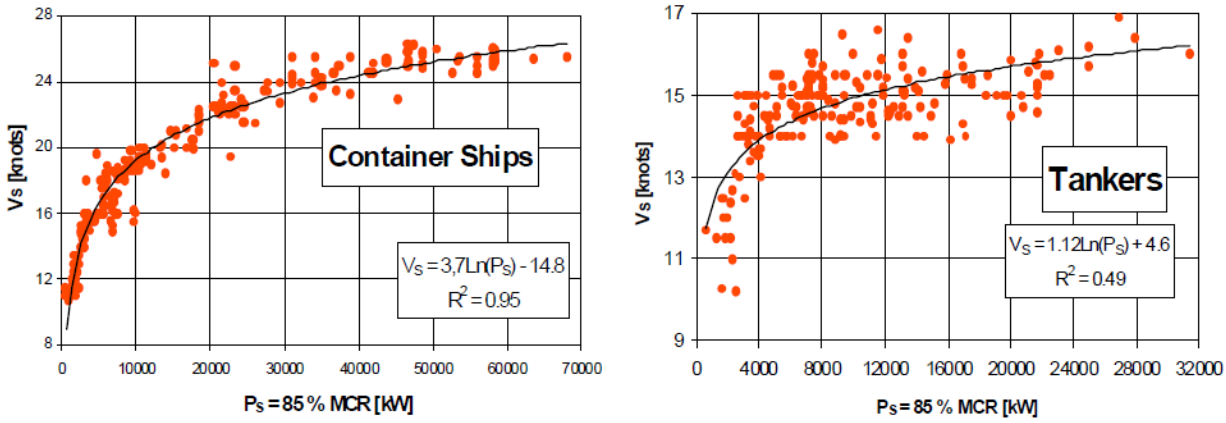

Abbildung 35: Geschwindigkeit-Leistung-Kurven bei Containerschiffen und Tanker (Olesen, et al. 2009)

Die Formel kann jedoch nicht für alle Schiffstypen direkt angewendet werden, da in manchen Fällen die Geschwindigkeit-Leistung-Relation von der oben genannten Funktion abweicht. Dies gilt vor allem bei Schiffstypen kleinerer Größenklassen, wie z. B. die RoRo-Passagier-Schiffe (*Roll-on-Roll-off-Passagier-Schiffe)* (Olesen, et al. 2009). So zeigt die folgende Abbildung die Geschwindigkeit-Leistung-Kurve für diese Schiffsklasse, bei der ein lineares Verhältnis zwischen der Geschwindigkeit und der dafür benötigten Leistung besteht.

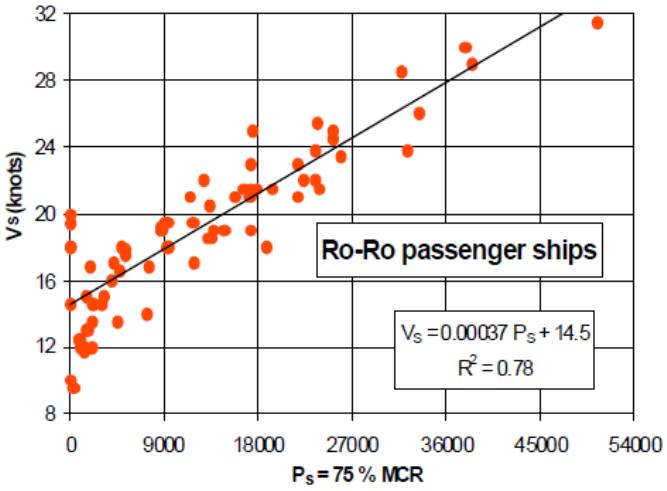

Abbildung 36: Geschwindigkeit-Leistung-Kurve bei RoRo-Passagier-Schiffen (Olesen, et al. 2009)

Unter den im Dokument genannten Schiffsklassen sind die Fähren der Frisia-Flotte den RoRo-Passagier-Schiffe zuzuordnen. Somit kann angenommen werden, dass auch eine

lineare Geschwindigkeit-Leistung-Relation bei den Frisia-Fähren besteht. Basierend auf dieser Annahme, wurde im Rahmen der Datenerhebung eine Formel gesucht, mit der sich diese Relation abbilden lässt. Dabei soll diese Formel eine lineare Funktion sein, die für eine bestimmte Geschwindigkeit (in km/h) eines Schiffes die annähernd benötigte Leistung (in kW) liefert. Außerdem soll sie allgemeingültig sein, d. h. die Formel soll für alle Fähren der Frisia-Flotte anwendbar sein.

Formel 2: Berechnung der benötigten Leistung bei Frisia-Fähren

$$P_{benötigt} = \frac{P_{max} - \frac{1}{6}P_{max}}{V_{max}} * V_{aktuell} + \frac{1}{6}P_{max}$$

$$mit: P_{benötigt} := benötigte\ Leistung\ (in\ KW)$$
$$P_{max} := maximale\ Leistung\ des\ Schiffes\ (in\ KW)$$
$$V_{max} := maximale\ Geschwindigkeit\ des\ Schiffes\ (in\ km/h)$$
$$V_{aktuell} := aktuell\ festgestellte\ Geschwindigkeit\ des\ Schiffes\ (in\ km/h)$$

Die Formel entstand aus der Überlegung bzw. Annahme heraus, dass für eine maximale Geschwindigkeit eine maximale Leistung der Fähre benötigt werden muss. Des Weiteren wird angenommen, dass eine bestimmte Anfangsleistung erzeugt werden muss, um die Fähre zu beschleunigen. Mathematisch ausgedrückt bedeutet das, dass die Funktion die y-Achse an einem positiven Punkt (positiver y-Achsenabschnitt) schneidet. Der Wert für die Anfangsleistung wurde auf ein Sechstel der maximalen Leistung der Fähre festgesetzt.

Abbildung 37 stellt die Geschwindigkeit-Leistung-Relation bei Frisia I beispielhaft dar.

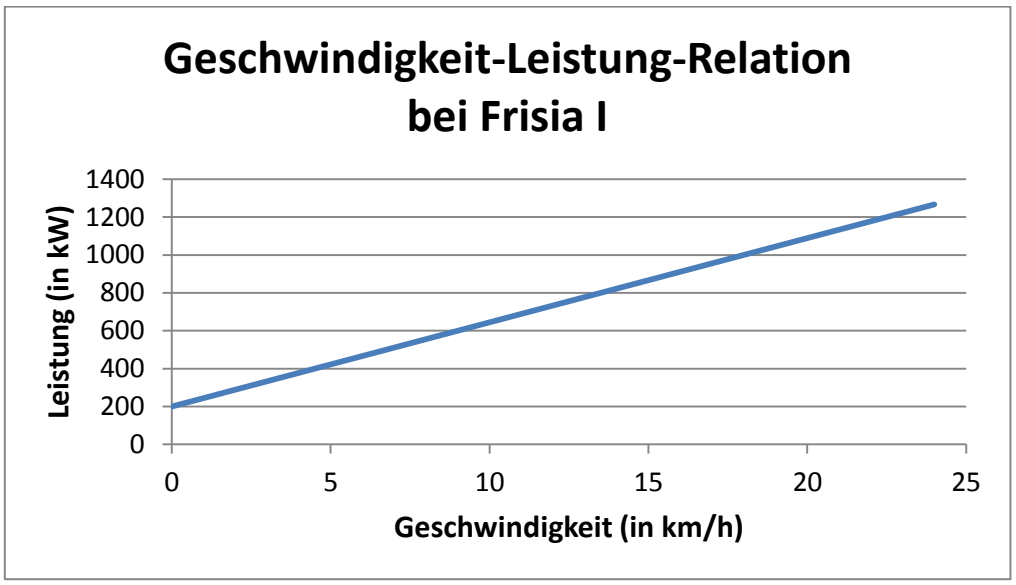

Abbildung 37: Geschwindigkeit-Leistung-Relation bei Frisia I

7.2.4 Technische Daten

7.2.4.1 Vorgehensweise

In der folgenden Abbildung ist der Prozess von der Ermittlung der Daten bis hin zu der Ableitung von Leistungs- und Verbrauchskurven, die für die Implementierung der domänenspezifischen Simulation verwendet werden, aufgezeigt.

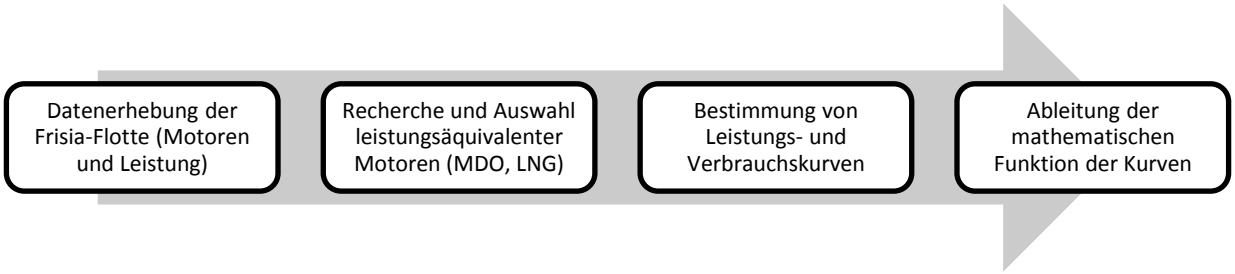

Abbildung 38: Prozess der Bestimmung von Leistungs- und Verbrauchskurven

Ausgehend von den technischen Informationen der Fähren der Reederei Frisia wurden Recherchen zu den MDO-Motoren der Flotte getätigt und darauf aufbauend ein äquivalenter LNG-Motor bestimmt. Ausgehend von den Spezifikationen der Hersteller wurden entsprechende Leistungs- und Verbrauchskurven der Motoren ermittelt, aus denen mathematische Funktionen abgeleitet wurden, die für die Implementierung des Szenarios verwendet wurden.

An dieser Stelle ist zu erwähnen, dass aufgrund der Informationsverfügbarkeit bezüglich der Motoren an manchen Stellen Annahmen getroffen wurden, die im Folgenden entsprechend gekennzeichnet sind.

7.2.4.2 Technische Daten der Frisia-Flotte

In diesem Abschnitt werden alle relevanten Daten bzgl. der Schiffe, insbesondere der Motorendaten, erhoben. Als ersten Schritt wurden die spezifischen Daten der Reederei Frisia beschafft. Ausgehend von den erhaltenen Informationen sind im Folgenden die Eigenschaften der Schiffe der Frisia exemplarisch an vier Schiffen der Flotte aufgezeigt.

Tabelle 14: Schiffseigenschaften der Frisia-Fähren

Merkmal	**Frisia I**	**Frisia II**	**Frisia IV**	**Frisia V**
Länge (m)	63,7	63,5	70,7	63,75
Breite (m)	12	12	13,7	12
Tiefgang (m)	1,8	1,25	1,4-1,75	1,75
Tragfähigkeit (t)	258,54	203,53	326	250,26
Geschwindigkeit	12,5	12	12	11,8
Hauptmaschinen	3x Volvo	2x MTU	4x Mitsubishi	1x Volvo, 2x Deutz
Typ	TAMD 163/D25	16V 2000 M60	D30 M	TAMD 163/12V816
Antrieb	2x Schottel / Schraube	Schraube	4x Voith-Schneider	2x Schottel / Schraube
Leistung (kW)	1127	1600	1880	1114
Verbräuche (Norddeich - Norddeich) (Liter)	360	340	500 (3 Antriebe) 580 (4 Antriebe)	260
Personenkapazität	794/1500	702/1350	990/1338	700/1442
Kfz-Kapazität	50	50	60	50

Die in der Tabelle 14 aufgezeigten Schiffe besitzen mehrere Motoren und werden mit MDO betrieben. Aufbauend auf diesen Informationen galt es nun, einen leistungsäquivalenten LNG-Motor, mit dem die Schiffe potenziell ausgerüstet werden können, zu ermitteln. Da es sich bei den Fährschiffen der Reederei Frisia um Schiffe der gleichen Größenklasse handelt (vgl. Länge, Breite und Tragfähigkeit), wurde ein einheitlicher Motor, der dem benötigten Leistungsvolumen aller Schiffe gerecht wird, ermittelt.

Die wichtigste Kennzahl in diesem Zusammenhang ist die Leistung der Motoren, die in Kilowatt (kW) angegeben ist. Damit einhergehend ist die Umdrehungszahl des Motors, gemessen in Umdrehungen pro Minute (Rounds per minute - rpm), für die Bestimmung der (maxi-

malen) Geschwindigkeit und des Treibstoffverbrauchs signifikant. Dabei ist die Geschwindigkeit umso höher, desto höher die Umdrehungszahl der Motoren ist (zunehmende Auslastung des Motors).

Wie aus der Tabelle 14 ersichtlich wird, befinden sich die Leistungen der Motoren zwischen 1114 kW und 1880 kW. Demnach kommt ein Motor infrage, der in diesem Leistungsspektrum liegt.

Abgeleitet von der Ermittlung der Motorendaten, konnten anhand der Motorenspezifikationen der Hersteller entsprechende Leistungs- und Verbrauchskurven für die jeweiligen Motoren ermittelt werden. In dieser Hinsicht war es erforderlich zwischen dem Leistungs- und Verbrauchsverhalten von Marinedieselmotoren und LNG-Motoren zu unterscheiden, welche ein unterschiedliches Verbrauchsverhalten in Relation zur Umdrehungszahl aufweisen.

7.2.4.3 Leistungs- und Verbrauchskurven MDO-Motor

Die Daten des Marinedieselmotors sind an die Spezifikation des Modells TAMD D25 des Herstellers Volvo angelehnt (Volvo 2004). Da nur für dieses Modell hinreichende Informationen aus der Spezifikation des Herstellers für die Bestimmung der Leistungs- und Verbrauchskurven vorlagen, wurde dieser Motor als Referenzmotor für alle MDO-betriebenen verwendet. Laut der Spezifikation besitzt einer dieser Motoren eine Leistung von 400 kW. Demnach müssen drei dieser Motoren für ein Schiff verwendet werden, um insgesamt auf 1320 kW Gesamtleistung zu kommen, was im oben beschriebenen Leistungsspektrum der Frisia-Fähren liegt.

Anhand der Spezifikation wurde zunächst die Abhängigkeit der erzeugten Leistung (kW) von der Umdrehungszahl des Motors (rpm) untersucht. Die in der folgenden Tabelle aufgezeigten Werte wurden den Diagrammen der Spezifikation entnommen und in der Abbildung 39 in Form einer Kurve dargestellt. Sinngemäß steigt dabei die Leistung bei zunehmender Umdrehungszahl des Motors an.

Tabelle 15: Leistungsdaten MDO-Motor

Bezeichnung	**Maßeinheit**								
Umdrehungen /min	rpm	1000	1100	1200	1300	1400	1500	1600	1700
Leistung	kW	110	140	180	230	290	360	440	550
Verbrauch	g/kWh	237	225	215	209	209	212	215	216

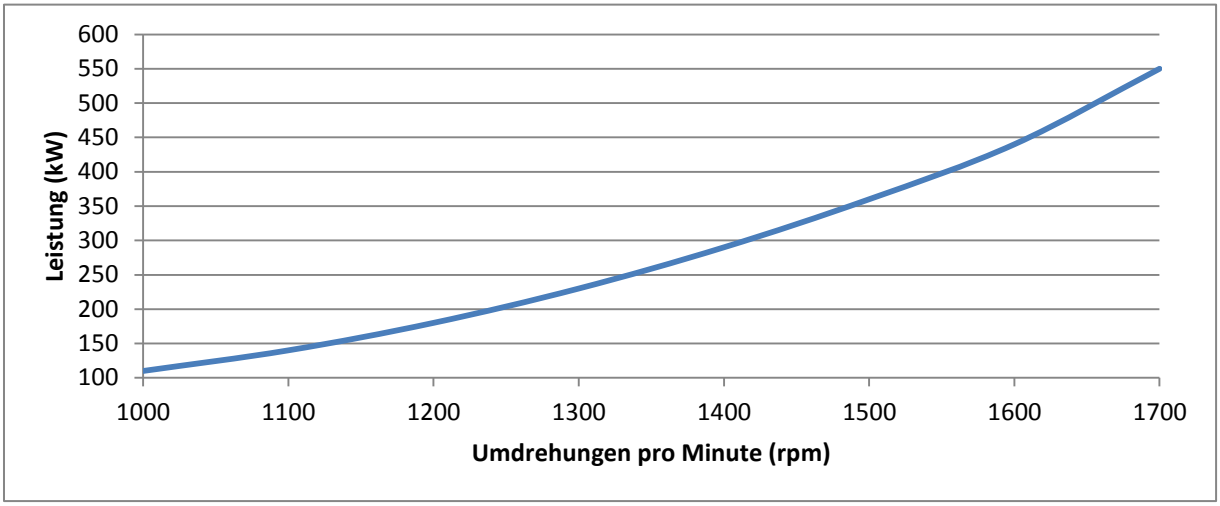

Abbildung 39: Leistungskurve MDO-Motor

Aus der in Abbildung 39 aufgezeigten Leistungskurve wurde durch das Verfahren der Regression folgende Formel hergeleitet:

Formel 3: Leistungskurve MDO-Motor

$$\text{Leistungskurve}_{\text{MDO}} = 1{,}131 \times 10 e^{2{,}298 \times 10^{-3} x}$$

Ausgehend von diesen Informationen kann ermitteln, bei welcher Umdrehungszahl des Motors wie viel Leistung erzielt wird. Der Treibstoffverbrauch, in diesem Fall gemessen in Gramm pro Kilowattstunde (g/kWh), hängt wiederum von der Umdrehungszahl des Motors ab. In Abbildung 40 ist die Verbrauchskurve für den MDO-Motor aufgezeigt, die diese beiden Messgrößen in Relation zueinander stellt. Analog zur Leistungskurve wurde die entsprechende Formel per Regression ermitteln:

Formel 4: Verbrauchskurve MDO-Motor

$$\text{Verbrauchskurve}_{\text{MDO}} =$$

$$-6{,}439 \times 10^{-10} x^4 + 3{,}285 \times 10^{-6} x^3 - 6{,}039 \times 10^{-3} x^2 - 4{,}673\, x - 1{,}039 \times 10^3$$

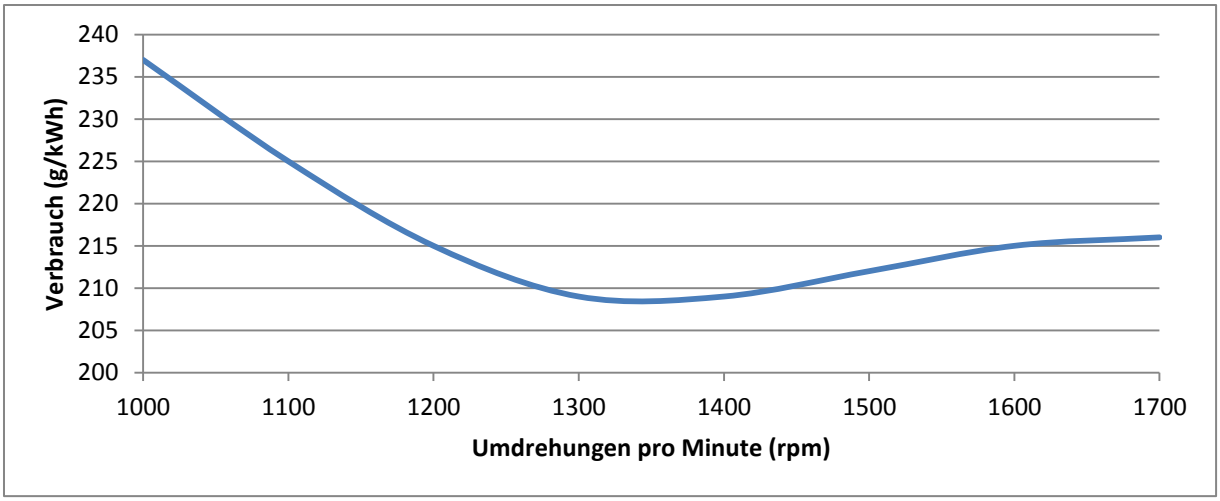

Abbildung 40: Verbrauchskurve MDO-Motor

7.2.4.4 Leistungs- und Verbrauchskurven LNG-Motor

Wie im einleitenden Teil dieses Abschnitts bereits erwähnt, musste ein LNG-fähiger Motor bestimmt werden, der dem Leistungsspektrum der Frisia-Fähren entspricht.

Angelehnt an der Einführung der ersten mit LNG betriebenen Fährschiffe in Norwegen, wurde die Fähre M/F Glutra als Referenz genutzt (Statoil Research Center 2002). Im Folgenden sind Eckdaten dieser Fähre aufgeführt:

Tabelle 16: Schiffsdaten zur Fähre Glutra

Länge	94,8 m
Breite	85,2 m
Personenkapazität	300
Kfz-Kapazität	96
Motoren	4x Mitsubishi GS 12R-PTK
Treibstofftanks	2x 27.2 m³ LNG-Tanks

Diese ist mit vier Mitsubishi-Motoren vom Typ GS12R-PTK in der V12-Konfiguration ausgerüstet. Neben Mitsubishi sind auch Rolls Royce, MAN und Wärtsilä Anlaufstellen der Recherchen nach einem vergleichbaren Motor gewesen. Jedoch konnten entweder aus den Spezifikationen der Motoren nicht die notwendigen Informationen hergeleitet werden, oder die Motoren waren für die Fähren in diesem Anwendungsbeispiel zu groß.

Ein Mitsubishi-Motor (Typ GS12R-PTK) weist eine Leistung von 675 kW auf. Für das Fallbeispiel kommen somit zwei dieser Motoren für die Ausstattung einer Fähre infrage, um eine

Gesamtleistung von 1350 kW zu erzielen und somit dem Leistungsspektrum der Frisia-Schiffe zu entsprechen.

In Tabelle 15 sind die Leistungs- und Verbrauchsdaten des Motors hinterlegt. Der Motor erzielt bei einer Umdrehungszahl von 1500 rpm seine maximale Leistung von 675 kW und verbraucht dabei 0,34 m³/kWh (Statoil Research Center 2002). Da für den Motor ausschließlich eine Kurve angegeben war, die den Verbrauch zur Leistung darstellt, wurde an dieser Stelle die Annahme getroffen, dass sich die Umdrehungszahl pro angegebenen Punkt um 100 verringert, wodurch sich der Umdrehungsbereich von 1000-1500 rpm erstreckt.

Tabelle 17: Leistungsdaten LNG-Motor

Bezeichnung	**Maßeinheit**						
Umdrehungen/min	rpm	1000	1100	1200	1300	1400	1500
Leistung	kW	264	300	350	410	500	675
Verbrauch	m³/kWh	0,34	0,31	0,285	0,27	0,26	0,258

Aus dieser Tabelle wurden analog zum MDO-Motor sowohl eine Leistungs- als auch eine Verbrauchskurve abgeleitet, die in den folgenden Abbildung dargestellt sind.

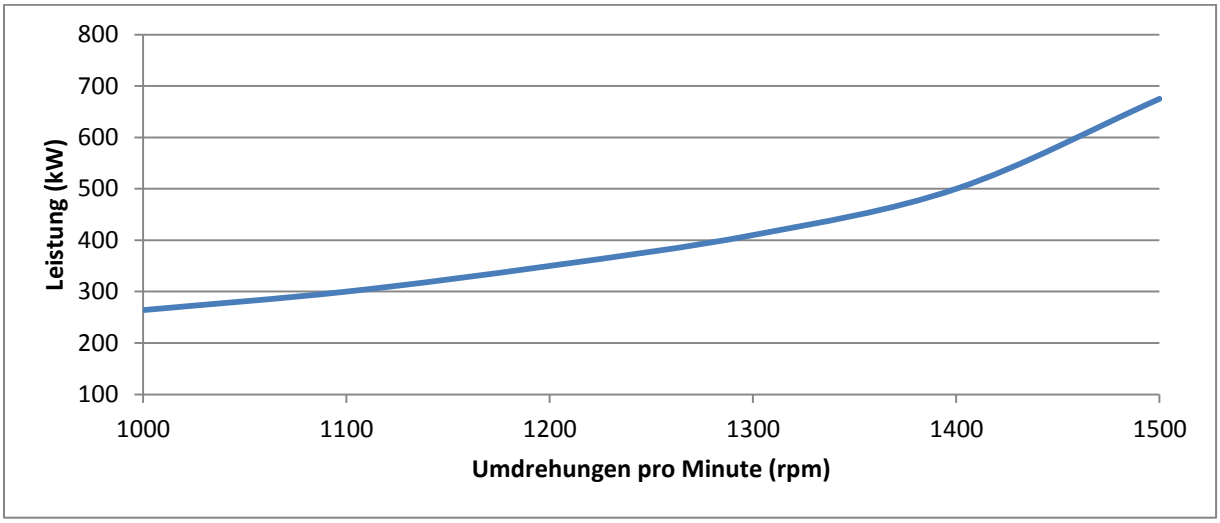

Abbildung 41: Leistungskurve LNG-Motor

Die Leistungskurve beider Motoren weist hierbei das gleiche Steigungsverhalten auf. Die hieraus abgeleitete Formel für die Leistungskurve lautet:

Formel 5: Leistungskurve LNG-Motor

$$\text{Leistungskurve}_{\text{LNG}} = 1{,}229 \times 10^{-8}\text{x}^4 - 5{,}762 \times 10^{-5}\text{x}^3 + 1{,}015 \times 10^{-1}\text{x}^2 - 7{,}917 \times 10\text{x} + 2{,}324 \times 10^4$$

Im Gegensatz dazu weist die Verbrauchskurve ein anderes Verhalten auf als die MDO-Verbrauchskurve. Während beide Motoren im niedrigen Umdrehungsbereich den höchsten Verbrauch aufweisen, hat der MDO-Motor im Bereich 1300-1400 rpm den niedrigsten Verbrauch. Danach steigt der Verbrauch bei zunehmender Umdrehungszahl wieder an. Im Gegensatz dazu ist der LNG-Motor umso effizienter, desto höher die Umdrehungszahl ist. Bei maximaler Auslastung arbeitet dieser Motor am wirtschaftlichsten.

Die aus der Verbrauchskurve abgeleitete Formel wurde wiederum durch Regression bestimmt:

Formel 6: Verbrauchskurve LNG-Motor

$$\text{Verbrauchskurve}_{\text{LNG}} = 1{,}042 \times 10^{-12}\text{x}^4 - 4{,}884 \times 10^{-9}\text{x}^3 + 8{,}899 \times 10^{-6}\text{x}^2 - 7{,}662 \times 10^{-3} + 2{,}945$$

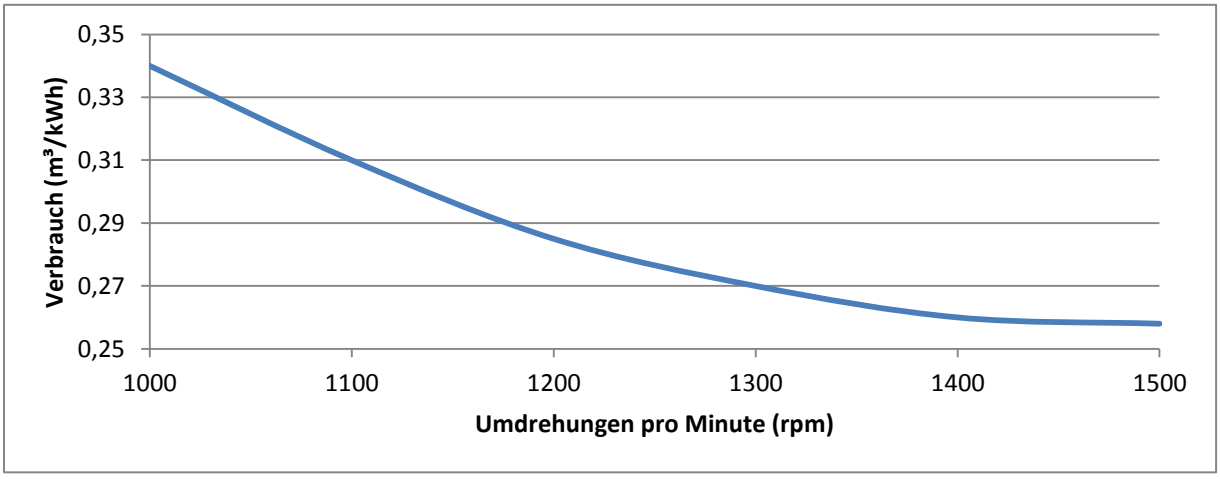

Abbildung 42: Verbrauchskurve LNG-Motor

7.2.4.5 Leistungseffizienz der Motoren

Durch die im oberen Abschnitt ermittelten Leistungs- und Verbrauchskurven kann bestimmt werden, wie viel Treibstoff benötigt wird, um in einem Motor eine bestimmte Leistung zu erzeugen. Jedoch verfügt jeder Motor über einen Wirkungsgrad, der als die Leistungseffizienz betrachtet werden kann. Der Wirkungsgrad eines Motors beschreibt das Verhältnis zwischen zugeführter und abgegebener Leistung (Input-Output) und berechnet sich durch den Quotienten aus abgegebener Leistung und zugeführter Leistung:

Formel 7: Berechnung des Wirkungsgrads allgemein

$$\text{Wirkungsgrad}_{\text{Motor}} = \frac{Abgegebene\ Leistung}{Zugefuehrte\ Leistung}$$

Angegeben wird der Wirkungsgrad entweder als Dezimalzahl oder als Prozentsatz.

Ausgehend von einer vorgegebenen Geschwindigkeit der Schiffe, muss die abgegebene Leistung des Motors bestimmt werden. Über die hier abgeleiteten Wirkungsgradkurven wird das Verhältnis bestimmt, wie viel Leistung dem Motor hinzugefügt werden muss, um die abgegebene Leistung zu ermitteln.

Die Wirkungsgradkurve sowohl für LNG als auch für MDO basiert dabei auf den Leistungs- und Verbrauchsdaten sowie auf dem Energiegehalt der Treibstoffe. Die Ermittlung der Wirkungsgradkurven erfolgt nach dem folgenden Vorgehen:

1. Der Verbrauch wird in g/kWh (Gramm pro Kilowattstunde) angegeben. Es wird anhand des Energiegehalts des Treibstoffs bestimmt, wie viel Energie (Kilojoule, KJ) der Treibstoff besitzt.
2. Der ermittelte Energiewert in Kilojoule wird in kWh umgerechnet (Faktor 0,278 * 10^{-3}). Dadurch wurde ausgehend vom Treibstoffverbrauch ermittelt, wie viel Energie Input erzeugt werden muss, um eine kWh im Motor zu erzeugen.
3. Stellt man diese beiden Input- und Output-Größen gemäß des Wirkungsgrads (Formel 7) in Relation zueinander, so erhält man den Wirkungsgrad in Form einer Dezimal- bzw. Prozentzahl.

Die folgende Tabelle stellt diese Vorgehensweise für den MDO-Motor dar, welche in der Abbildung 43 als Kurve dargestellt ist.

Tabelle 18: Berechnung des Wirkungsgrads MDO-Motor

Bezeichnung	**Maßeinheit**								
Umdrehungen /min	rpm	1000	1100	1200	1300	1400	1500	1600	1700
Leistung	kW	110	140	180	230	290	360	440	550
Verbrauch	g/kWh	237	225	215	209	209	212	215	216
Berechnung der Input-Energie anhand der Menge Treibstoff (g/kWh) und des Energiegehalts des Treibstoffs (MDO := 42,7 KJ/g) → Angabe in kWh									
Input-Energie für 1 kWh Output	kWh	2,813	2,671	2,552	2,481	2,481	2,517	2,552	2,564
Berechneter Wirkungsgrad	%	35,55	37,44	39,18	40,31	40,31	39,74	39,18	39,00

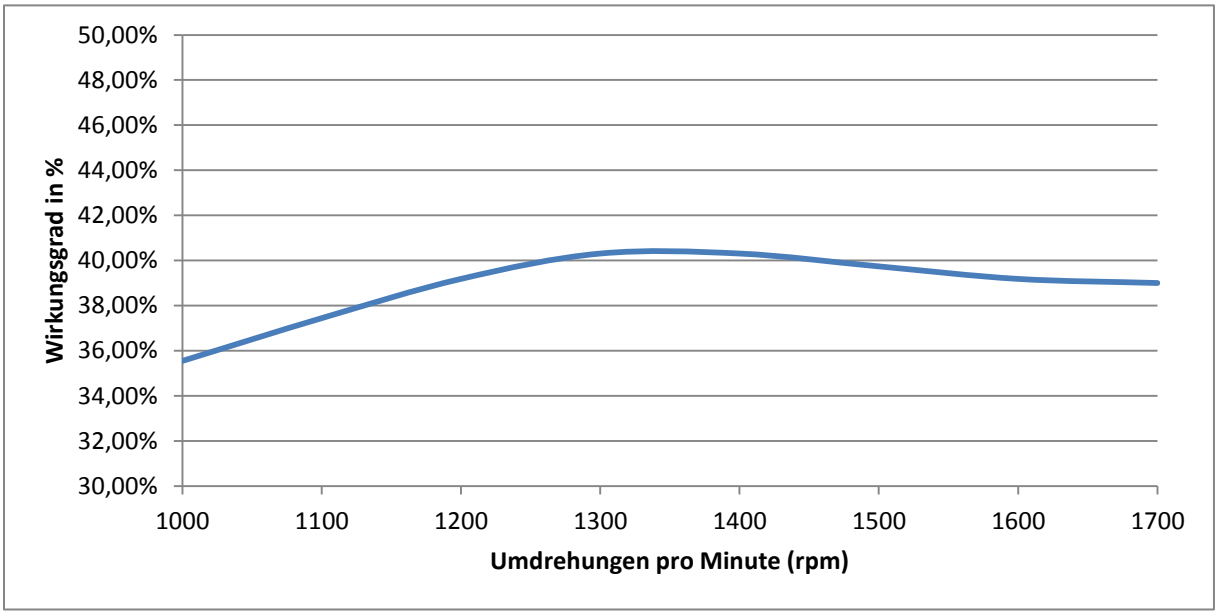

Abbildung 43: Wirkungsgradkurve MDO-Motor

Die daraus abgeleitete Formel lautet:

Formel 8: Wirkungsgradkurve MDO-Motor

$$\text{Wirkungsgrad}_{\text{MDO}} = 1{,}9967 \times 10^{-12}\text{x}^4 + 1{,}0323 \times 10^{-8}\text{x}^3 + 1{,}951 \times 10^{-5}\text{x}^2 + 1{,}5883 \times 10^{-2} - 5{,}0542$$

Analog zum MDO-Motor musste ebenso für den LNG-Motor eine Wirkungsgradkurve ermittelt werden. Dabei wird die gleiche Vorgehensweise verwendet, wobei jedoch der Energiegehalt des Treibstoffs und der rpm-Bereich entsprechend unterschiedlich sind (siehe Tabelle 10). In der folgenden Tabelle sind die Wirkungsgradwerte für den LNG-Motor aufgezeigt, welche in der darauf folgenden Abbildung als Kurve dargestellt sind.

Tabelle 19: Berechnung des Wirkungsgrads LNG-Motor

Bezeichnung	**Maßeinheit**						
Umdrehungen/min	rpm	1000	1100	1200	1300	1400	1500
Leistung	kW	264	300	350	410	500	675
Verbrauch	m³/kWh	0,265	0,26	0,27	0,285	0,31	0,34
Input-Energie für 1 kWh Output	kWh	2,285	2,084	1,916	1,815	1,748	1,781
Wirkungsgrad	%	43,76	47,99	52,20	55,10	57,22	56,14

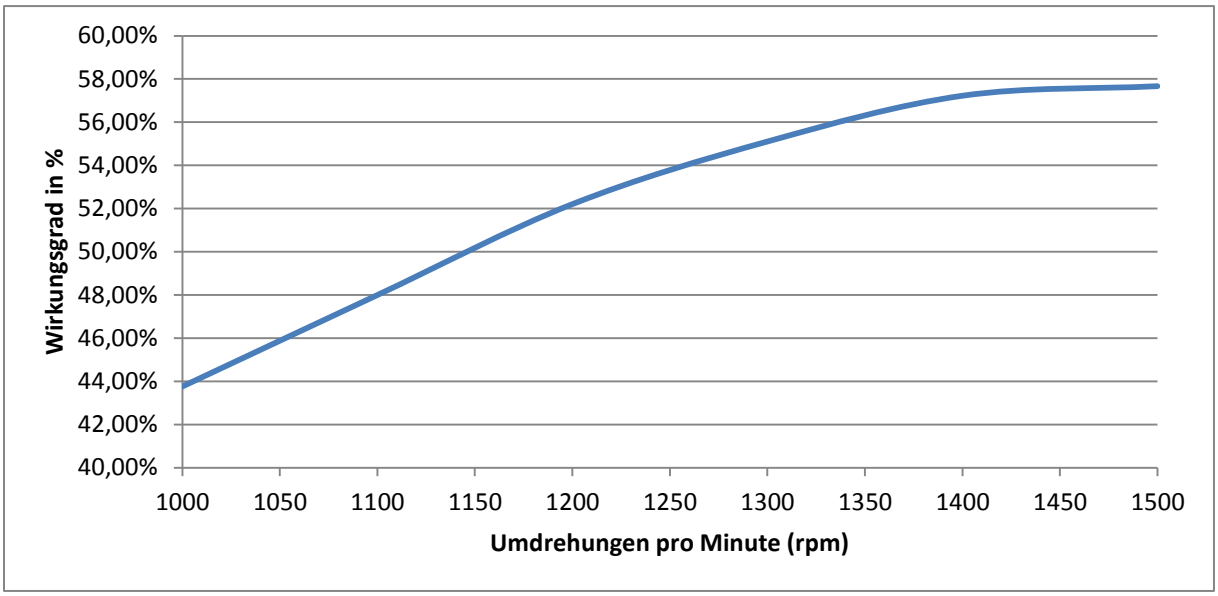

Abbildung 44: Wirkungsgradkurve LNG-Motor

Dementsprechend wurde daraus die Wirkungsgradkurve des LNG-Motors abgeleitet:

Formel 9: Wirkungsgradkurve LNG-Motor

$$\text{Wirkungsgrad}_{\text{LNG}} = -2{,}361 \times 10^{-12}\text{x}^4 + 1{,}048 \times 10^{-8}\text{x}^3 + 1{,}764 \times 10^{-5}\text{x}^2 + 1{,}374 \times 10^{-2} - 3{,}781$$

7.2.5 Streckenplanung des Fährverkehrs

Die Darstellung der Infrastrukturstandorte ist mit der Unterstützung der Reederei Frisia erstellt worden. Relevante Häfen zur Abbildung des Schiffslinienverkehrs sind der Festlandhafen Norddeich sowie die Inselhäfen Norderney und Juist. Auf dieser Strecke existiert ein Linienverkehr mit den Strecken Norddeich – Norderney, Norddeich – Juist. Für die Strecke Norddeich–Norderney müssen 10,2 km gefahren werden. Die in der Schifffahrt gültige Entfernungsangabe ist die Seemeile (sm) und wird mit dem Verhältnis 1 sm = 1852m berechnet. Somit müssen von Norddeich nach Norderney 5,5 sm zurückgelegt werden. Äquivalent dazu beträgt die Strecke von Norddeich nach Juist 10 sm bzw. 18,52 km (Reederei Norden Frisia AG 2011). In der Simulation ist den Häfen ein geschwindigkeitsreduzierter Bereich von 500 Metern vorgelagert. Das Streckennetz der Simulation ist in Abbildung 45 dargestellt.

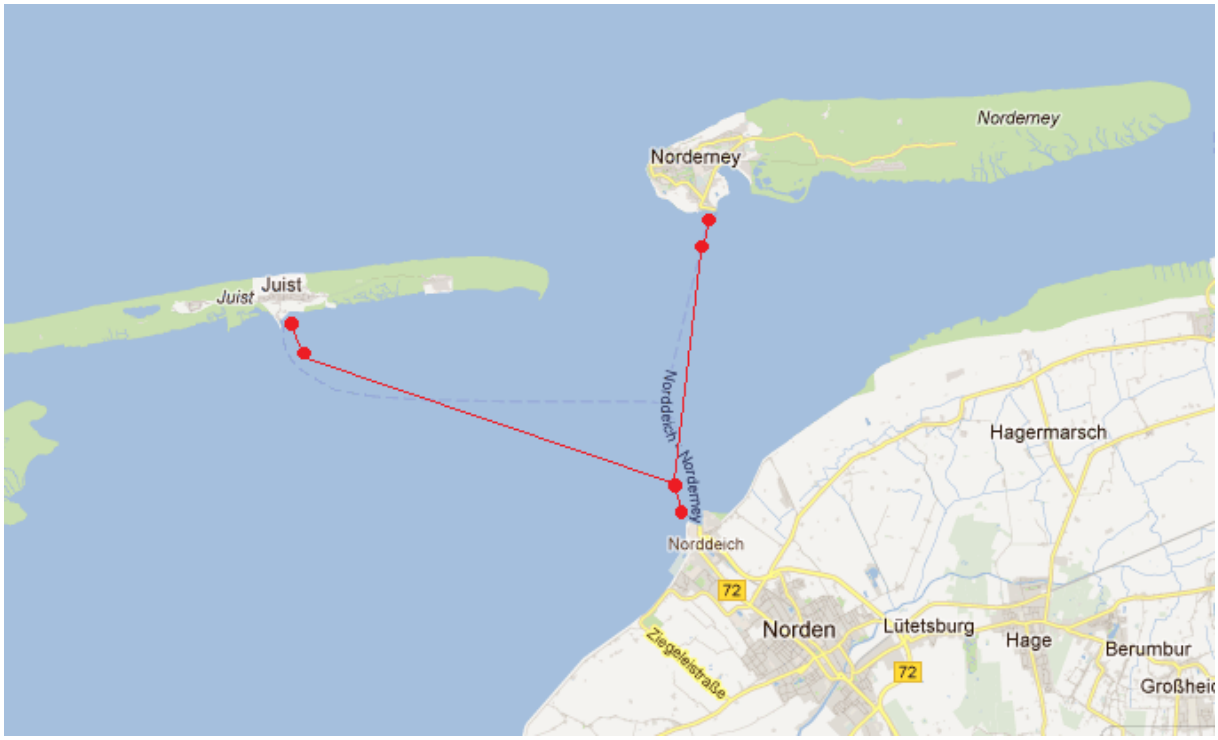

Abbildung 45: Schiffslinienverkehr der Reederei Frisia (Eigene Darstellung nach (Google 2012))

Der für das Streckennetz gültige Fahrplan ist simulationsgerecht aufgearbeitet und für ein Jahr, unterteilt in Sommer und Winterfahrplan, erstellt worden. Für die Anfertigung des Fahrplans sind die regelmäßigen Fahrten identifiziert und für die Simulation aufbereitet worden. Tabelle 20 stellt den Winterfahrplan exemplarisch dar. In der Simulation sind den Fahrplänen die Schiffe zugewiesen.

Tabelle 20: Winterfahrplan Norddeich Norderney (Reederei Norden Frisia AG 2011)

	Samstag u. Sonntag		**Montag bis Donnerstag**		**Freitag**	
Abfahrtsort	*Norderney*	*Norddeich*	*Norderney*	*Norddeich*	*Norderney*	*Norddeich*
	07:30	06:30	06:15	06:15	06:15	06:15
	09:45	08:45	07:30	07:30	07:30	07:30
	12:00	11:00	08:45	08:45	08:45	08:45
	14:15	13:15	10:30	10:15	10:30	10:15
	16:45	15:30	11:45	11:45	11:45	11:45
	19:30	18:00	13:30	13:15	13:30	13:15
			15:30	15:15	15:30	15:15
			16:45	16:45	16:45	16:45
			18:15	18:00	18:15	18:00
					20:30	19:15

7.3 Umsetzung des Fallbeispiels

7.3.1 Szenariobeschreibung

Die in der Datenerhebung ermittelten Informationen fließen in die Erstellung zweier Fallbeispiele ein. Bei den Fallbeispielen werden die Treibstoffe LNG und MDO gegenübergestellt. Die Schiffe und die Infrastruktur sind bei beiden Fallbeispielen identisch, um vergleichbare Werte zu erhalten. Aufgrund der fehlenden Infrastruktur für den Treibstoff LNG in der Bundesrepublik Deutschland sind die aus der Datenerhebung hypothetisch angenommenen Werte übernommen worden. Resultierend aus dieser Annahme wird davon ausgegangen, dass Distributionswege existieren, um die Tankstationen zu beliefern.

Dieser Abschnitt beschreibt den Entwurf des Szenarios und welche Spezialisierungen für die Strecke und Infrastruktur eingepflegt wurden. Anschließend wird die Spezialisierung der Fahrzeuge auf die Schifffahrt erörtert. In der folgenden Implementierung des Szenarios finden die erstellten Klassen für die Spezialisierung Betrachtung, während der letzte Abschnitt der Auswertung des Szenarios dient.

7.3.2 Entwurf des Szenarios

Der Entwurf des Szenarios erfordert die Erweiterung des Verkehrsmodells und der Verkehrssimulation. Abbildung 46 beschreibt das Mapping der Verkehrssimulation auf die in der Szenariobeschreibung erläuterten Anforderungen.

Im Folgenden wird die Erweiterung der einzelnen Komponenten erläutert. Dazu werden zuerst die Spezialisierung des Streckennetzes und der Infrastrukturstandorte beschrieben (Abschnitt: Spezialisierung des Streckennetzes und des Infrastrukturstandortes). Anschließend wird die Erweiterung des Fahrzeugs inklusive der Motoren beschrieben. Weiterhin sind die Erweiterung der Verkehrsagenten und deren Logik erklärt.

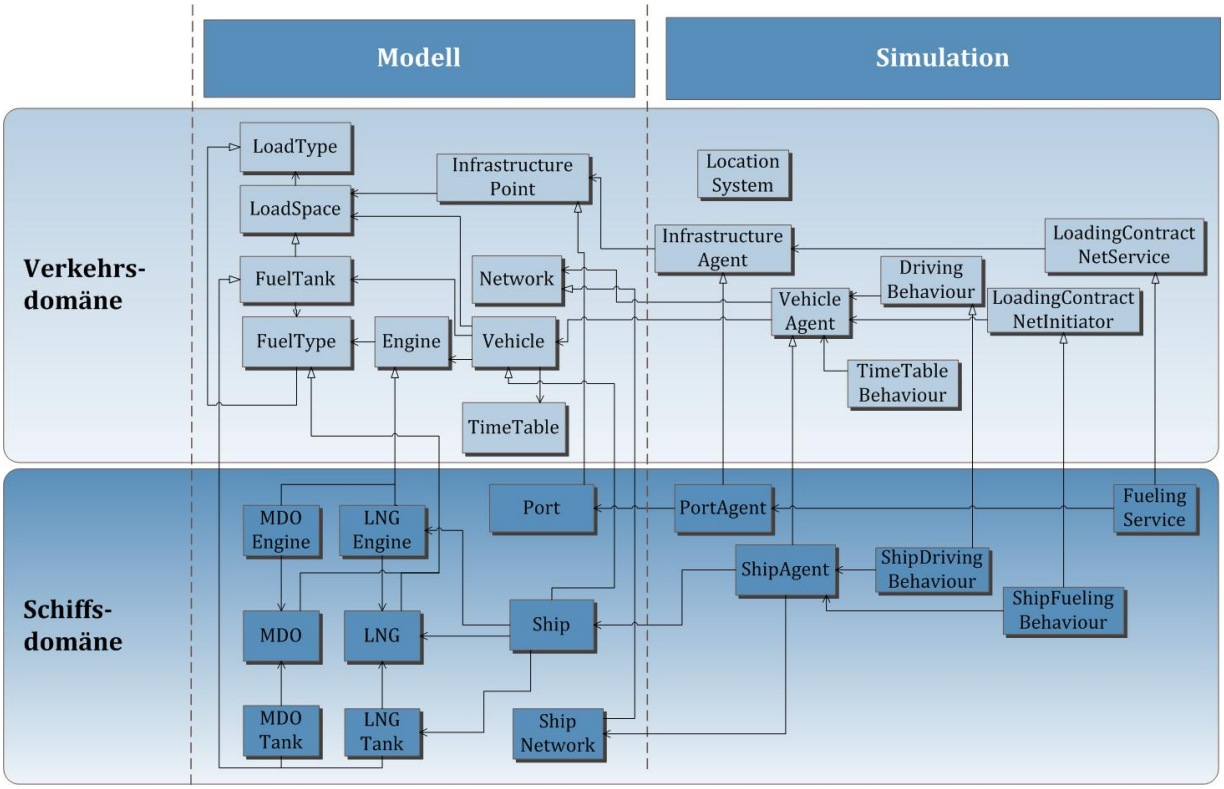

Abbildung 46: Mapping des Verkehrsmodells auf die Schifffahrt

7.3.2.1 Spezialisierung des Streckennetzes und des Infrastrukturstandortes

Der in 6.2.2 beschriebene Entwurf des Infrastrukturstandortes ermöglicht die Darstellung von Häfen, in denen ein Güteraustausch stattfindet. Für das vorliegende Beispiel spielen Güter keine große Rolle, da aufgrund der Datenbasis keine sinnvollen oder nützlichen Annahmen getroffen werden können. Wichtig für die gegebene Problemstellung ist vor allem der verwendete Treibstoff. Daher müssen die spezialisierten Infrastrukturstandorte (Klasse: *Port*) in der Lage sein, Treibstoff zu lagern und auszutauschen. Der Austausch findet mithilfe von Verladestellen (Klasse: *Terminals*) statt, die bereits in der Verkehrssimulation ausreichend

ausgeprägt sind und nicht weiter spezialisiert werden müssen. Zur Realisierung des Austausches müssen sie von Agenten gesteuert werden, die entsprechende Dienste anbieten.

Das Schiffsnetzwerk muss das Verkehrsnetzwerk um die Eigenschaften erweitern, die für das Beispielszenario relevant sind. Da der Vergleich von LNG und MDO anhand der Flotte von Frisia vorgenommen wird, benötigen die Strecken keine maximalen Emissionsgrenzen, da die Emissionen der Frisia-Schiffe in jedem Fall im zulässigen Bereich sind (siehe Abschnitt 7.1.1). Da aber klare Vorgaben bzgl. der maximalen Geschwindigkeit in diesem Bereich der Nordsee existieren, müssen die Verbindungen eine maximale Geschwindigkeit aufweisen.

Umwelteinflüsse wie Tidenverläufe werden aufgrund der geringen Auswirkung auf einen Vergleich von Treibstoffen nicht modelliert. Für andere Vergleiche könnten die Schiffsverbindungen um weitere Eigenschaften erweitert werden.

7.3.2.2 Spezialisierung des Fahrzeugs

Schiffe (Klasse: *Ship*) bilden die Fahrzeuge innerhalb des Schiffsmodells ab. In Analogie zu den Häfen verfügen sie ebenfalls über Treibstofftanks und Verladestellen, da sie in der Lage sind Treibstoffe zu verbrauchen, um sich zwischen Streckenpunkten auf dem Streckennetz zu bewegen.

Darüber hinaus sind Schiffe in der Lage einen Fahrplan abzufahren. Dies wird in der Simulation dazu genutzt, um zu bestimmten Zeiten an bestimmten Orten zu sein. Dadurch wird ermöglicht, den Fahrplan der Frisia-Flotte – wie in Abschnitt 7.2.5 beschrieben – von einem Schiff originalgetreu abfahren zu lassen.

Motoren

Zum Zwecke der Fortbewegung besitzen Schiffe spezialisierte Motoren, welche die notwendige Leistung für das Fahren des Schiffs auf einer Verbindung zur Verfügung stellen und dabei Treibstoff verbrauchen. Treibstoffe weisen dabei Eigenschaften wie NO_x- und SO_x-Schadstoffausstoß pro Treibstoffeinheit auf, um die durch den Treibstoffverbrauch erzeugten Emissionen abzubilden. Es müssen für einen Vergleich der Treibstoffe LNG und MDO-Motoren modelliert werden, um die Zusammenhänge aus Abschnitt 7.2.4 darstellen zu können.

Tanken

Das Tanken stellt das Beladen eines Treibstofftanks eines Schiffes durch einen Hafen an einem Terminal da. Für die Modellierung ist eine Tankstrategie vonnöten. Die Schiffe müssen immer so viel Treibstoff haben, dass sie Ihren Fahrplan einhalten können, ohne eine Verzögerung zu verursachen. Da die Entscheidung, wann ein Schiff betankt wird, dem

Kapitän obliegt und bedarfsorientiert ist, wurde dies auch in der Schiffsimulation modelliert. Schiffe überprüfen am Tagesende, ob sie noch über genügend Treibstoff verfügen. Wenn sie unter einem Viertel der Tankkapazität liegen, suchen sie den nächsten Hafen mit Treibstoffbunker und bewegen sich zur Betankung dorthin.

7.3.2.3 Spezialisierung des Verkehrssimulationsagenten

Um die verschiedenen Anforderungen an die Schiffe und Häfen zu erfüllen, werden sie von entsprechenden Agenten gesteuert (Klasse: *PortAgent* und *ShipAgent*). Die Hafenagenten müssen einen Service zum Betanken anbieten, der von Schiffsagenten genutzt werden kann. Dabei muss es für die Schiffe möglich sein, Tankmenge und Geschwindigkeit auszuhandeln. Die Modellierung von Kosten wird an dieser Stelle vernachlässigt, da ohne fundierte Informationen über die Preise von LNG keine dynamische Preisfindung stattfinden kann. Stattdessen können nach der Simulation Preise und Kosten anhand von Schätzungen aggregiert dargestellt werden.

Schiffsagenten müssen weiterhin die Logik zur Fortbewegung der Schiffe besitzen. Dazu muss der Agent um Funktionalitäten zum Abfahren der Schiffsverbindungen erweitert werden. Schiffe fahren das Minimum aus erlaubter Maximalgeschwindigkeit der Verbindung und der Maximalgeschwindigkeit des Schiffes. Für die Simulation bedeutet dies, dass die Schiffe zu jedem Zeitpunkt die Maximalgeschwindigkeit der Strecken fährt, da die Höchstgeschwindigkeit im Wattenmeer 8 kn beträgt und alle Schiffe der Flotte höhere Maximalgeschwindigkeiten haben.

7.3.3 Implementierung des Szenarios

Nachdem die Erweiterung des Verkehrsmodells und der Verkehrssimulation entworfen wurde, findet eine Implementierung der dafür notwendigen Klassen statt.

Zuerst werden auf die Schiffs- und Hafenagenten und deren Services und Behaviours eingegangen. Anschließend werden die Ausprägungen der Schiffs- und Motorklassen und des Netzwerks vorgestellt und erläutert. Zuletzt wird erklärt, wie die Klassen mithilfe der *ShipSimulation* konsolidiert werden.

7.3.3.1 Ship

Ein Schiff ist die Ausprägung eines Fahrzeugs des Verkehrsmodells. Da das Fahrzeug bereits einen Großteil der notwendigen Logik zur Verfügung stellt, sind nur geringe domänenspezifische Anpassungen notwendig. Zum einen wird der Klasse *Ship* eine Maximalgeschwindigkeit (Attribut: *maxSpeed*) zugeordnet. Weiterhin beinhaltet ein Schiff die notwendige Logik, um zu ermitteln, wie schnell sich ein Schiff bei einer gegeben Motorleistung bewegt bzw. wie viel Leistung ein Schiff für eine gegebene Geschwindigkeit benötigt. Dies wird in

den Methoden *getPower* und der Umkehrfunktion *getSpeed* realisiert. Die Ermittlung dieser domänenspezifischen Funktionen ist im Abschnitt 7.2.3 beschrieben. Diese Methoden werden im *ShipDrivingBehaviour* benutzt, um die entsprechenden Werte zur Treibstoffverbrauchkalkulation zu erhalten.

7.3.3.2 ShipEngine

Die Erweiterung des Motors (Klasse: *Engine*) ist der *ShipEngine*. Auch hier brauchen kaum Erweiterungen zum *Engine* des Verkehrsmodells (siehe Abschnitt 6.2) vorgenommen werden. Auf dieser Ebene wird dem Motor nur eine maximale Leistung (Attribut: *maxPower*) zugewiesen. Eine genauere Spezialisierung findet dann in den Klassen *ShipEngineLNG* und *ShipEngineMDO* statt. In diesen Klassen werden die Effizienzkurven der Schiffsmotoren (Methode:*getEnergyConversionEfficiency*) über die Umrechnung auf Umdrehungen pro Minute (Methode: *getEngineRpm*) implementiert. Eine Herleitung dieser Funktionen kann in Abschnitt 7.2.4 gefunden werden.

7.3.3.3 ShipAgent

Der Schiffsagent bewegt das Schiff mit dem *ShipDrivingBehaviour*, welches eine Spezialisierung des *PresetPowersDrivingBehaviours* ist. Dies stellt die Funktionalität zur Verfügung, dass ein Schiffsagent die Motorenleistung für einen Streckenabschnitt in Abhängigkeit der Maximalgeschwindigkeit festsetzt und damit von einem Punkt im Netzwerk über eine *ShipConnection* zum nächsten Punkt fahren kann. Bei Abschluss eines Streckenabschnitts wird in dieser Klasse die Treibstoffreservierung ausgeführt (siehe Abschnitt 6.2.4), die Position erneuert und ein Log-Eintrag geschrieben. Für den Fall, dass ein Schiff nicht über genügend Treibstoff verfügt, wird in der Simulation ein Fehler gemeldet und die Simulation abgebrochen.

Weiterhin verfügt der Schiffsagent über das *FuelingBehaviour*, welches eine Ausprägung des *LoadingContractNetServices* (siehe Abschnitt 6.3.4) aus der Verkehrssimulation darstellt. Dazu wird am Ende des Tages mithilfe des *CheckFuelingNecessaryBehaviours*, welches eine domänenspezifische Tankstrategie darstellt, ermittelt, ob ein Tankvorgang notwendig ist. Wenn dies der Fall ist, wird der Tankvorgang mit dem *PrepareFuelingBehaviour* vorbereitet, indem der nächste Hafen, der den entsprechenden Treibstoff anbietet, gesucht wird. Existiert solch ein Hafen, wird zum Tanken dorthin gefahren, sofern sich das Schiff nicht bereits dort befindet. Der Agent nutzt dazu das *CreateFuelingEntryBehaviour,* um die Fahrt in der Simulation einzuplanen (siehe Abschnitt 6.1.4). Vor Ort wird mit dem *FuelingBehaviour* ein Treibstoff und die Tankgeschwindigkeit ermittelt und ein Tankvorgang mit dem Hafen ausgehandelt. Wenn ein Tankvorgang erfolgreich beendet wurde, wird ein Log-Eintrag geschrieben.

Den Großteil der Logik übernimmt im Schiffsagenten das *ShipTimeTableBehaviour*. Es stellt Funktionalitäten zum Abfahren eines gegebenen Fahrplans (siehe Abschnitt 6.3.5) mithilfe des *DrivingBehaviours*. Dazu wird zuerst die Route zum Ziel berechnet (Methode: *calculateRoute*), wobei für die Streckenabschnitte bereits die notwendige Motorleistung bestimmt wird. In der Methode *driveTo*, wird ein entsprechendes *DrivingBehaviour* zum richtigen Zeitpunkt bei dem *Scheduler* der Simulation einplant.

7.3.3.4 PortAgent

Der Hafenagent (Klasse: *PortAgent*) als Erweiterung des Infrastrukturagenten (Klasse: *InfrastructurePointAgent*) bedarf kaum domänenspezifischer Anpassung. Die Erweiterung besteht in dem Anbieten eines Betankungsservices (Klasse: *FuelingService*) welches das Betanken eines Schiffes anhand des in Abschnitt 6.3.4 beschriebenen Protokolls ermöglicht.

Da am *InfrastructurePoint* keine domänenspezifischen Veränderungen vorgenommen werden müssen, steuert der *PortAgent* direkt den *InfrastructurePoint*.

7.3.3.5 ShipConnection

Die Schiffsverbindung erweitert die Klasse *Connection* um eine Möglichkeit, Geschwindigkeitsbegrenzungen auf Strecken festzulegen. Die Klasse gibt die Höchstgeschwindigkeit in einem Streckenabschnitt an, die von den Fahrzeugen gefahren werden darf.

7.3.4 Editor

Für die domänenspezifische Ausprägung, die in diesem Fallbeispiel beschrieben ist, wird der *TrafficEditor* zu der domänenspezifischen Ausprägung *ShipEditor* abgeleitet. Folgendes Vorgehen skizziert, wie diese Ableitung geschieht, um den Editor so zu erweitern, dass er für die Domäne verwendet werden kann:

1. Modellgetriebene Erstellung einer Ableitung des *EditorModel* für die Schifffahrtsdomäne und Erzeugung des dazugehörigen EMF.Edit-Codes.
2. Programmatische Ableitung des Generators für die Schifffahrtsdomäne und Erstellung der *Validatoren* für den Editor.
3. Programmatische Ableitung des *TrafficEditors* für die Schifffahrtsdomäne. Dazu gehört auch die Einbindung
    a. der domänenspezifischen *Verkehrssimulation*,
    b. des domänenspezifischen *EditorModel* inklusive dessen EMF.Edit-Codes und
    c. des *Generators*.

Im Folgenden wird das Vorgehen genauer beschrieben.

7.3.4.1 Erstellung des domänenspezifischen EditorModels

Mit Hilfe von EMF-Ecore wird das *TrafficEditorModel* erweitert. Die Erweiterung hat den Namen *ShipEditorModel*.

Abbildung 47 zeigt einen Ausschnitt des *ShipEditorModel*. In dem Model gibt es eine Ausprägung der Klasse *Scenario* und diverse Ausprägungen der Klasse *Section*. Die Ableitungen der Klasse *Section* sind modellbasiert erstellt worden. Der Inhalt einer Section ist durch einen generischen Typparameter auf die jeweilige Klasse geprägt worden, bspw. die ShipSection mit dem generischen Typ Ship. Nach der Erstellung der *ShipEditorModel* wurde mit Hilfe von EMF für alle Klassen des neuen Models ein EMF.Edit-Code erzeugt. Die dazugehörigen *ItemProvider* und *AdapterFactories* ermöglichen eine einfache Einbindung in den zu erstellenden *ShipEditor*. Die Implementierung der *getItems*-Methode in einer *Section* und der Tab-Name für die Benennung des Section-Tabs im Editor (*getTabName*-Methode) wurde manuell ausgeprägt.

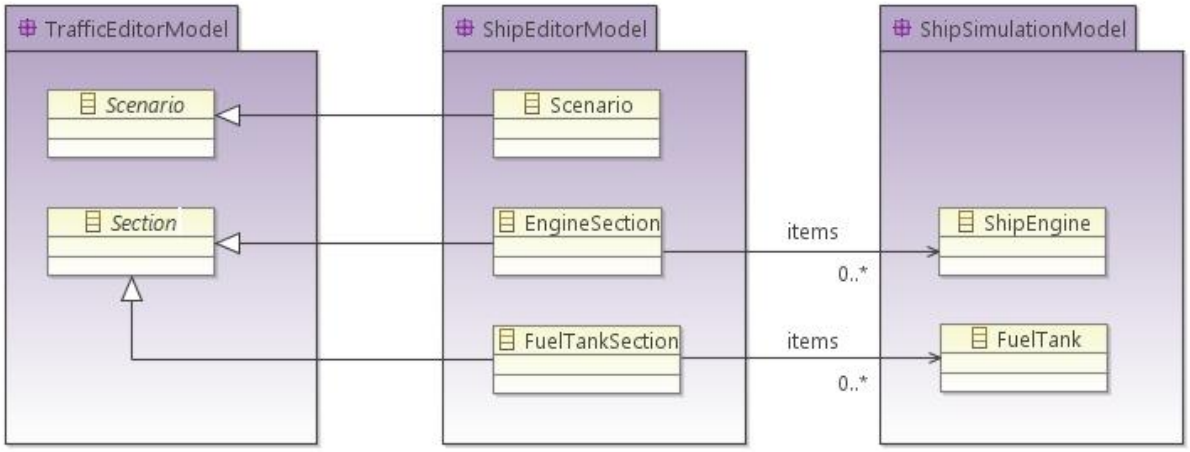

Abbildung 47: Ausschnitt des erstellten ShipEditorModel

7.3.4.2 Erstellung des domänenspezifischen Generators und der Validatoren

Generator

Abbildung 38 gibt eine Überblick über den domänenspezifischen Generator, welcher im Folgenden beschrieben wird.

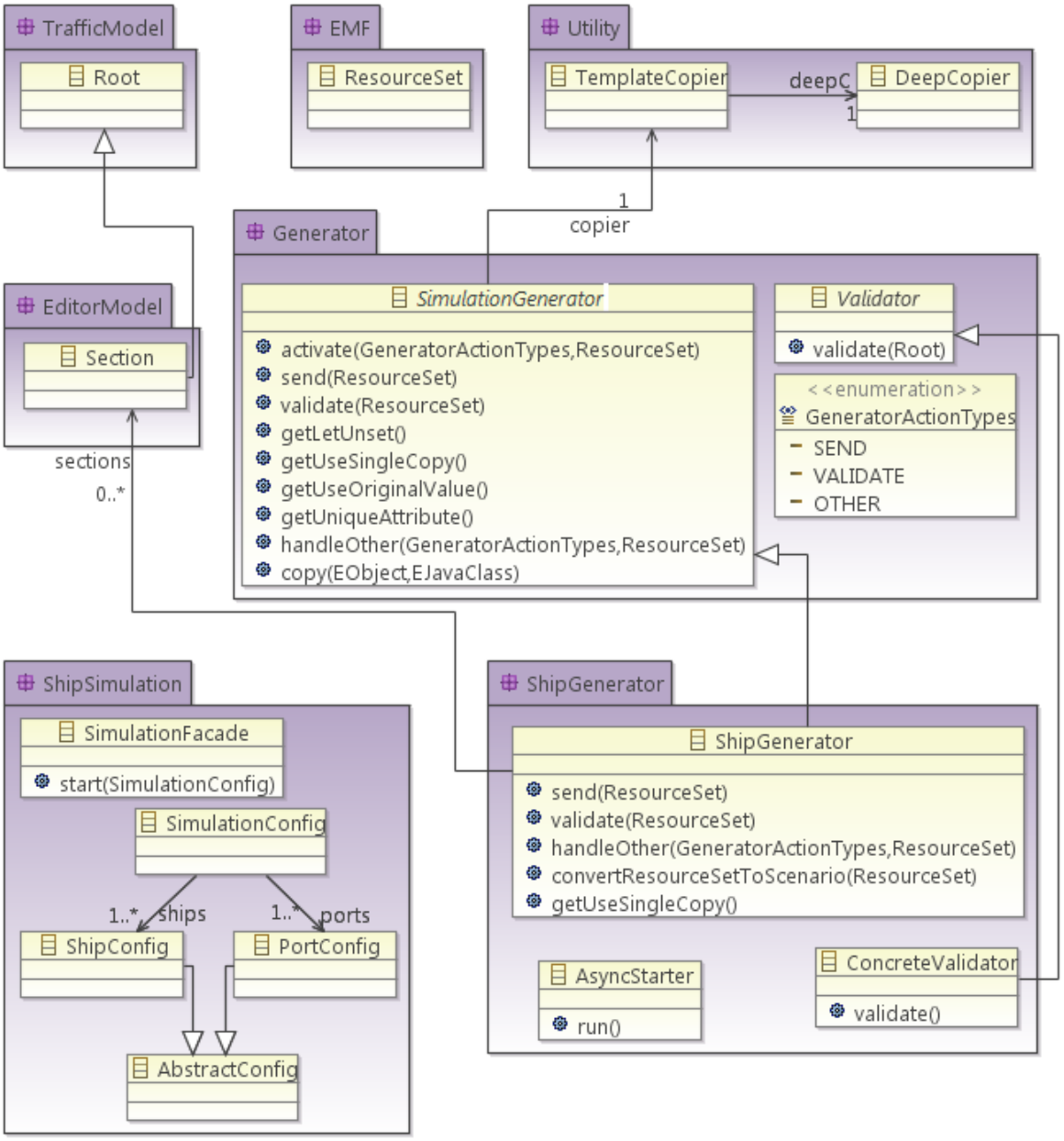

Abbildung 48: Domänenspezifischer Generator

In der domänenspezifischen Ausprägung des Generators werden die abstrakten Methoden *send*, *validate* und *handleOther* des *SimulationGenerator*s überschrieben. Da bei dieser Ausprägung neben dem Starten der Simulation und der Validierung der Attribute keine weitere Aktion vorgesehen ist, werden die Konstante *OTHER* und die korrespondierende *handleOther*-Methode nicht weiter verwendet. Die beiden anderen Methoden bekommen als Parameter von der Unterklasse vorgegebene *ResourceSet* übergeben. Da nur das darin enthaltene Rootobjekt *Scenario* von Bedeutung ist, wird über die Methode *convertResourceSetToScenario* dieses aus dem *ResourceSet* extrahiert.

Beim Initialisieren des Generators wird für jede *Section*, die vom Editor im *ResourceSet* übergeben wurde, intern eine Instanz abgelegt. Dadurch lassen sich die unmittelbaren Einträge aus dem Editor entnehmen.

Soll die Simulation über die *GeneratorActionTypes*-Konstante *SEND* gestartet werden, wird zuvor immer eine Validierung ausgeführt, weshalb hier zunächst die Validierung über den im folgenden Abschnitt beschriebenen Validator und daraufhin das Starten der Simulation erläutert wird.

Beim Validieren werden die Daten direkt aus der *Section* bezogen, ohne zuvor aggregiert oder anders manipuliert worden zu sein. Dies garantiert, dass nur die Eingaben des Benutzers beachtet werden und so Fehlermeldungen generiert werden können, die annähernd direkt auf einen konkreten Fehler hinweisen können.

Für die Validierung wird über die einzelnen *Sections* iteriert, deren Inhalt ausgelesen, und an den entsprechenden Validator übergeben. Ein invalider Eintrag in einer *Section* bricht die Validierung durch eine *InvalidConfigurationException* ab. Die *InvalidConfigurationException* ist keine Spezialisierung einer *RuntimeException*, sodass bei der Verwendung des Validators diese Fehlermeldung abgefangen und entsprechend behandelt werden kann. Dies geschieht, indem an der Benutzungsoberfläche der Inhalt der Fehlermeldung präsentiert wird und der Anwender diesen Fehler beheben muss. Soll ein Szenario simuliert werden, so muss der Generator über die *GeneratorActionTypes*-Konstante *SEND* aktiviert werden, indem unter anderem diese an die *activate*-Methode übergeben wird. In einem solchen Fall werden nach der Validierung die Daten für eine Simulation erstellt, indem die Vorgaben aus dem Editor generiert werden. Dazu wird aus der *SimulationFacade* ein entsprechendes Konfigurationsobjekt übergeben. Dies speichert alle erforderlichen Daten, welche die Simulation benötigt. Dabei werden sowohl die Startparameter gesetzt als auch die Schiffe mit den dazugehörigen Fahrpläne und Häfen kopiert. Das Kopieren wird durch die *copy*-Methode, die durch den *SimulationGenerator* bereitgestellt wird, durchgeführt, wobei schließlich noch eine optionale Namensgebung eingesetzt wird.

Die vollständig gefüllte Konfigurationsdatei wird über einen asynchronen Dienst an die Simulation übergeben. Dazu wird mit dem *AsyncStarter* ein neuer Thread erstellt, welcher die Konfigurationsdatei an die *start*-Funktion der SimulationFacade übergibt und somit Simulation und Editor entkoppelt.

Validator

Zum Validieren des gesamten Szenarios kommen mehrere Validatoren zum Einsatz. Jeder konkrete Validator validiert genau eine Entität aus dem Traffic- bzw. Shipmodel. Dies hat den Vorteil, dass der Validator durch generische Attribute auf eine bestimmte Entität beschränkt

werden kann und dadurch explizites casten entfällt. Die Validatoren auf der Framework-Ebene sind Klassen, die jedoch, da sie teilweise direkt verwendet werden, noch ein generisches Attribut beim Aufruf benötigen. Diese lassen sich auf der domänenspezifischen Ebene direkt setzen, was das folgende Beispiel zeigt.

Abhängig davon, ob ein Validator zu einer überschriebenen Klasse aus dem Model entsprechend umgesetzt wurde, erfolgt der Aufruf entweder über *new ShipValidator().validate(ship);* oder bspw. über *new VehicleValidator<Ship>().validate(ship).* Letztere Möglichkeit wurde z. B. bei *FuelType* oder *Terminal* verwendet, da für diese Klassen keine Erweiterung des Validators des Verkehrsmodells verwendet wurde.

7.3.4.3 Erstellung des ShipEditors

Zunächst wurde ein neues EMF-Plug-In-Projekt angelegt. In diesem Projekt werden Klassen erzeugt, die auf den Klassen des *TrafficEditors* basieren. Abbildung 49 verdeutlicht diesen Zusammenhang.

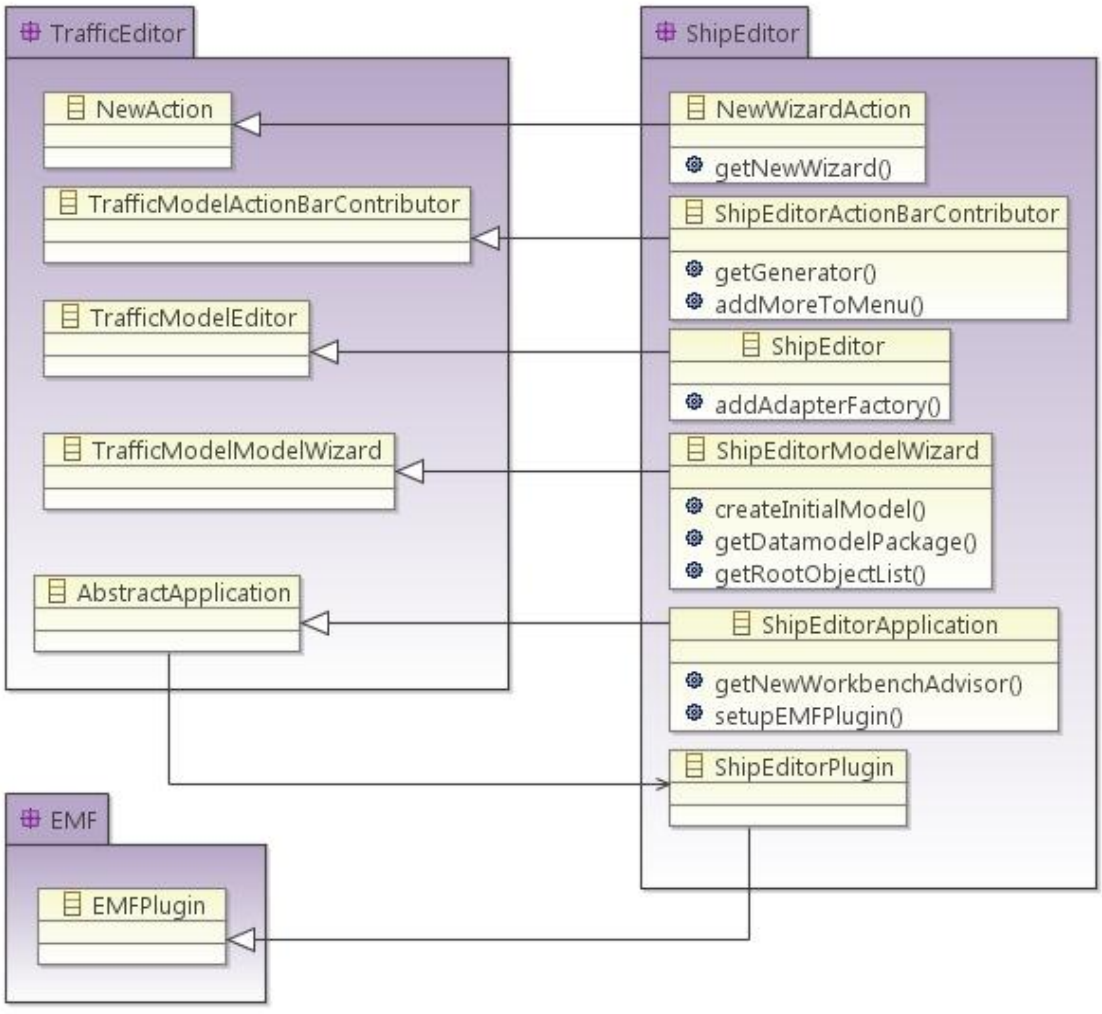

Abbildung 49: ShipEditor

Das *ShipEditorPlugin* erweitert die Klasse *EMFPlugin* und bildet damit ein „normales“ Projekt-Plug-In. Es ist als einziges nicht direkt von der Traffic-Ebene abhängig, allerdings Voraussetzung für ein Eclipse Plug-In. Wie schon in der Implementierung erwähnt, können durch diese Klasse externe Plug-In Ressourcen eingebunden werden. Ein Beispiel hierfür ist der Name des Editors, der standardmäßig „Editor Application“ ist, jedoch durch die Definition der Variablen „_UI_Editor_label“ in der Datei *plugin.properties* umbenannt werden kann.

ShipEditorApplication

Die *ShipEditorApplication* leitet die Klasse *AbstractApplication* aus dem *TrafficModelEditor* ab. Hier wurde über die Methode *getNewWorkbenchAdvisor* der notwendige *TrafficModelEditorAdvisor* initialisiert und zurückgegeben. Außerdem wird durch die Methode *setupEMFPlugin* die Instanz des *ShipEditorPlugin* zurückgegeben.

ShipEditorModelWizard

Der *ShipEditorModelWizard* erweitert den *TrafficModelModelWizard* und ist damit in der Lage, eine neue Konfigurationsdatei aus den an den Editor angeschlossenen *Factories* zu erzeugen und anzulegen. Dazu verwendet der *ShipEditorModelWizard* die *ShipEditorModelFactory* in der Methode *createInitialModel* und erzeugt daraus das *Root*-Objekt der Konfiguration (*Scenario*). Anschließend wurde die Konfiguration mit den notwendigen *Sections* preinitalisiert, sodass diese beim Erstellen einer neuen Konfiguration nicht manuell angelegt werden müssen.

NewWizardAction

In der Klasse *NewWizardAction* wurde über die Methode *getNewWizard* eine neue Instanz des *ShipEditorModelWizard* erzeugt und von der Methode zurückgegeben. Die NewWizardAction wurde über die *plugin.xml* des *ShipEditor*Plug-In Projektes eingebunden.

ShipEditorActionBarContributor

Über den *ShipEditorActionBarContributor*, der den *TrafficModelActionBarContributor* ableitet, wurde ein zusätzlicher Menüeintrag in die Menüleiste eingefügt. Hierbei handelt es sich um den Menüeintrag für die *CleanAction*, die alle gespeicherten Einstellungen der RCP aus dem User-Home-Verzeichnis löscht. Außerdem wurde an dieser Stelle der *ShipSimulationGenerator* initialisiert. Dieser ist eine Erweiterung des Generators aus dem *TrafficModelEditor*, der an die domänenspezifische Ausprägung „Ship“ angepasst wurde.

ShipEditor

Die Klasse *ShipEditor* erweitert die Klasse *TrafficModelEditor* und prägt die Methode *addAdapterFactory* aus. Über diese Methode wurde die *AdapterFactory* des *ShipEditorModels an* den Editor übergeben.

Hierdurch wurde die *Ship*-spezifische Ausprägung des *EditorModels* erstellt. Der in Abbildung 50 gezeigte Editor ist der finale Editor zum Erstellen von Schiffsmodellen.

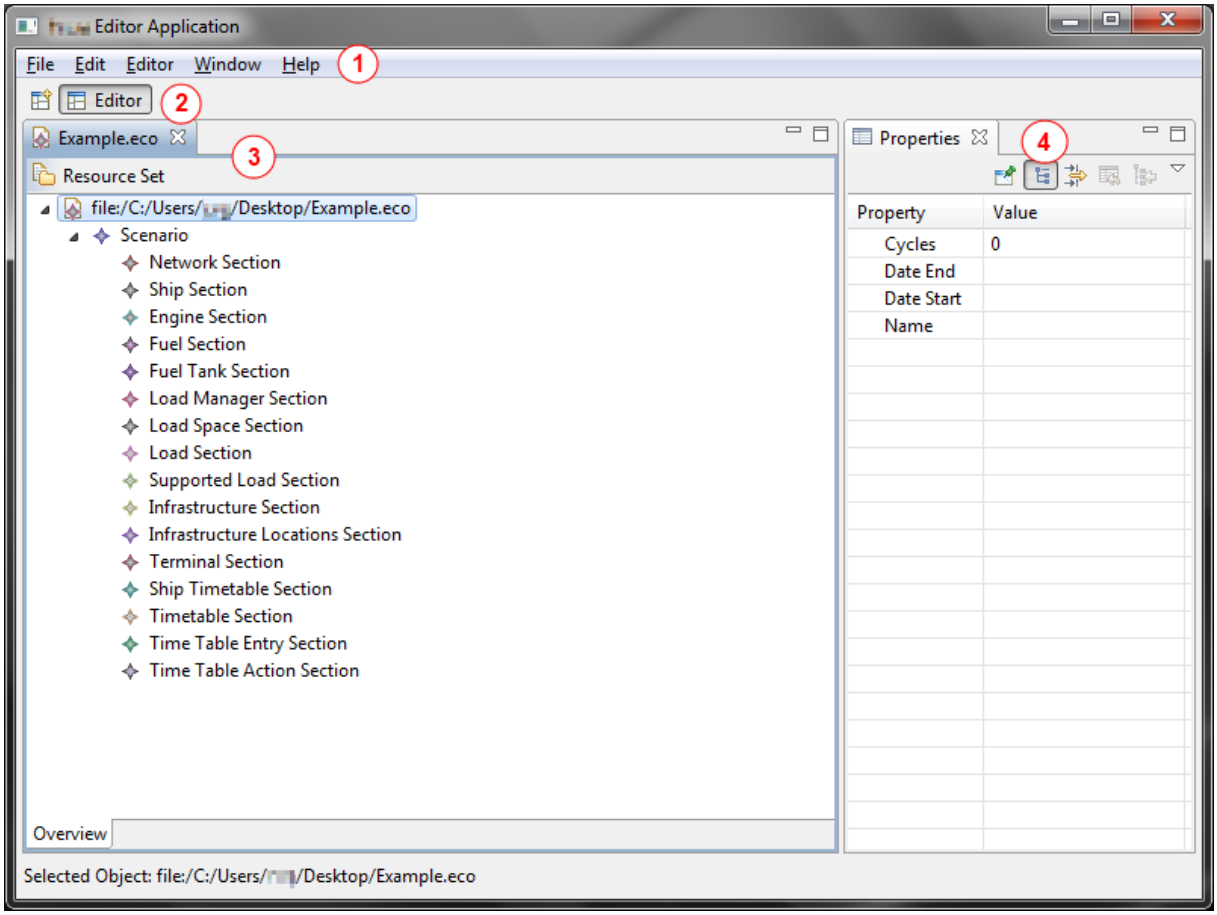

Abbildung 50: GUI des ShipEditors

In Abbildung 50 ist eine domänenspezifische Ausprägung der Editor-GUI zu sehen. Die Oberfläche des Editors besteht zum einen aus der Menüleiste (1), über die grundlegende Funktionen, wie z. B. das Öffnen und Speichern von Dateien, möglich sind. Das Hauptfenster des Editors (3) wird beim Laden oder Anlegen einer Konfiguration für die Simulation angezeigt. Auf der rechten Seite ist das Property-Fenster (4) hinterlegt, in dem die Eigenschaften der ausgewählten Objekte angezeigt und verändert werden können. Zudem können bei einer Ableitung des TrafficEditors weitere Perspektiven als Plug-In-Extension integriert werden. Diese Perspektiven sind dann über das Menüelement für Perspektiven (2) erreichbar. Für den ShipEditor wurde die Reporting-Perspektive eingebunden, die Funktionen für die Konfiguration des Reporting und Logging besitzt.

Über die Reiter am unteren Ende des Fensters können einzelne Bereiche zur Dateneingabe ausgewählt werden. Im rechten Fenster erfolgt die Eingabe der Werte.

Um das Fallbeispiel in die Simulation einzugeben, wird eine Umwelt benötigt. Das sog. Netzwerk (*Network*) enthält die benötigten Strecken. Diese Strecken bestehen aus verschie-

denen Punkten (*Trackpoints*). Für die Simulation ist das Schiffsliniennetz der Reederei Frisia implementiert. Die Streckenpunkte sind die Inseln Norderney und Juist sowie der Festlandhafen Norddeich. Jeder Hafen besitzt einen vorgelagerten Punkt, der die Hafenein-/ausfahrt kennzeichnet. Über Strecken bestehen Regulierungen der Geschwindigkeit.

Die Agenten erhalten die Daten zur Darstellung von Schiffen durch Eigenschaften, die in den Bereichen „Ships“, „Fuel“, „Fueltank“ und „Engines“ eingegeben sind.

Im Fallbeispiel sind vier Schiffe Implementiert. Jedes Schiff hat einen Namen, der existenten Schiffen der Flotte nachempfunden ist. Das sind die Schiffe Frisia 1, 2, 4 und 5. Jedes Schiff hat einen Starthafen, von dem aus es die Simulation startet. Um Bewegung aufnehmen zu können, hat jedes Schiff mindestens einen Motor. Durch die Informationen von der Reederei Frisia wurden Informationen zu echten Motoren eingeholt und genutzt. Den Motoren wird dann die Leistung (*maxPower*) und einen Treibstoff, der Verwendung findet, zugewiesen. Leistungskurven definieren den Verbrauch beim Befahren der Strecken.

Das Fallbeispiel nutzt MDO oder LNG. Um diese Treibstoffe verwenden zu können, bedarf es der Zuweisung eines Tanks im Schiff. Dieser beschreibt die maximale Ladungsmenge eines Treibstoffes, die ein Schiff laden kann. Die Eigenschaften des Treibstoffes müssen über den Reiter „Fuel“ zusätzlich eingegeben werden. Hierbei existieren neben der Energie, dem Volumen und Gewicht noch Informationen zu den Emissionswerten, die ein Treibstoff erzeugt. Dies dient zur Emissionsermittlung, die über später im Reporting aufgelistet wird.

Um eine realistische Umwelt zu erzeugen, wurden die regelmäßigen Fahrten der Reederei Frisia zu einem Fahrplan zusammengestellt. *TimeTableEntries* beschreiben jede einzelne Fahrt mit Zielhafen und Ankunftszeitpunkt und werden in *TimeTables* zusammengeführt. Anschließend werden den *TimeTables* Schiffe zugeordnet, die diese dann befahren. Sog. *TimetableActions* regeln die geplanten Tankvorgänge.

Jeder Agent, der ein Fahrzeug darstellt, hat zusätzlich die Möglichkeit, Güter und Personen darzustellen, die er geladen hat. In der aktuellen Simulation ist diese Funktion aber aufgrund fehlender Grundlagen nicht implementiert.

7.3.5 Auswertung des Szenarios

Nachfolgend werden kurz die für das Evaluationsszenario eingesetzten Technologien vorgestellt. Hierzu zählen neben Teneo und EclipseLink auch das zur Informationsdarstellung eingesetzte Business Intelligence Reporting Tools.

7.3.5.1 Teneo

EMF bietet im Standard keine direkte Möglichkeit an, um EMF-basierte Objekte in einer Datenbank speichern zu können. Teneo als datenbankorientierte Persistenzlösung für EMF

erweitert dahingehend die Möglichkeiten von EMF (Rummler und Voigt 2009). Teneo bietet eine automatische Generierung eines Mappings von EMF auf einen objektrelationalen Mapper (O/R-Mapper) wie EclipseLink (EclipseFoundation 2010). Es stellt hierbei weitestgehend eine Schicht zwischen EMF und dem O/R-Mapper dar. Die EMF-Objekte werden dabei nicht direkt annotiert, sondern Teneo ist auf Grundlage von in EMF-Modellen enthaltenen Metadaten direkt in der Lage, ein entsprechendes Mapping zu erzeugen. Das erstellte Mapping wird in eine XML-Datei geschrieben. Diese orm.xml-Datei kann von EclipseLink gelesen werden, sodass die aufgeführten Objekt-Referenzen in eine relationale Datenbank übertragen werden können.

Zusammenfassend kann gesagt werden, dass durch den Einsatz von Teneo die Speicherung der EMF-Objekte vereinfacht wird, da keine manuellen Annotationen gesetzt werden müssen. Änderungen am zugrunde liegenden EMF-Modell ziehen keine manuellen Anpassungen am entstandenen Model-Code nach sich. Stattdessen kann die orm.xml automatisch von Teneo erzeugt werden.

7.3.5.2 EclipseLink

EclipseLink ist ein Persistenzframework, das es ermöglicht Java-Objekte in einer relationalen Datenbank zu speichern (EclipseFoundation 2011). EclipseLink ist die Referenzimplementierung der Java Persistence API (JPA) 2.0 (Oracle 2011).Mittels objektrelationalem Mapping können Java-Objekte in verschiedene Datenbanksysteme überführt werden. Bei diesem Vorgang werden bspw. Objekte auf entsprechende Tabellenstrukturen einer relationalen Datenbank abgebildet. In diesem Projekt wird hierfür eine MySQL-Datenbank verwendet. Ein einmal persistiertes Objekt kann mit EclipseLink aus der relationalen Datenbank abgefragt werden. Aus dieser Abfrage kann EclipseLink ein entsprechendes Java-Objekt erstellen.

EclipseLink vereinfacht die Speicherung der erstellten Objekte. Abfragen auf der Datenbank werden nicht über EclipseLink vorgenommen, da persistierte Daten in BIRT weiter verwendet werden sollen und BIRT intern JDBC nutzt.

7.3.5.3 Business Intelligence and Reporting Tools

Das Business Intelligence Reporting Tool (BIRT) ist ein Plug-In für die Eclipse-Plattform. BIRT bietet dem Benutzer die Möglichkeit Anwendungen mit Reporting-Funktionalität auszustatten, um bspw. gesammelte Daten in Form von benutzerdefinierten Berichten auszugeben (Dencker 2007). Die Berichte bestehen aus verschiedenen Elementen (EclipseFoundation, BIRT 2012).

Die wichtigsten Elemente sind die darzustellenden Daten und die Quellen, aus denen diese entnommen werden sollen. Daten können u. a. aus Datenbanken stammen oder auch per Webservice angebunden werden. BIRT ist aber auch in der Lage, Daten aus XML-Quellen

einzulesen und JOIN-Operationen auf diesen durchzuführen. Es existieren noch weitere Datenquellen, auf die aber nicht näher eingegangen wird (Dencker 2007).

Datentransformation und Geschäftslogik bilden zwei weitere Elemente eines Berichts, mit denen die Daten aufbereitet werden können. Die Datentransformation dient hierbei dazu, simple Daten wie Textdateien entsprechend der Benutzerbedürfnisse zu sortieren, zusammenzufassen, zu gruppieren oder zu filtern. Mithilfe der Geschäftslogik können Daten in nützliche Informationen überführt werden. Dies wird bei BIRT zum einem durch das Einbinden von Skripten (JavaScript) und zum anderen durch das Aufrufen von Java-Code erreicht (Dencker 2007).

Die Präsentation, also die Darstellung des Berichts in Form von Tabellen und Diagrammen, stellt den letzten Schritt bei der Erstellung eines Berichtes dar. Hierzu nutzt BIRT den Report-Designer, der die Berichtsgestaltung ermöglicht. Der Report-Designer umfasst dabei bspw. Werkzeuge, die es ermöglichen Layout-Komponenten wie Tabellen oder Labels aus einer Palette per Drag&Drop in den Bericht zu ziehen (Dencker 2007).

Eine weitere Komponente ist die Design-Engine, mit der Report-Designs im XML-Format erstellt werden können. Des Weiteren nutzt der Report-Designer die Design-Engine-API, um aus den erstellten Designs eine XML-Datei zu erzeugen. Sie ist somit das Bindeglied zwischen dem Report-Designer und der Report-Engine (Dencker 2007).

Die Report-Engine wiederum ist für die Erstellung des Berichts auf Basis der zuvor erstellten XML-Datei zuständig. Sie öffnet u. a. Verbindungen zu Datenquellen, führt hinterlegte Geschäftslogik aus und bindet bspw. mithilfe der Chart-Engine Diagramme in Berichte ein (Dencker 2007).

Die letzte in diesem Abschnitt erläuterte Komponente in BIRT ist der sog. Report-Viewer, der eine Vorschau auf den Bericht innerhalb von Eclipse bietet. Er nutzt dafür die Funktionalitäten der Report- und Chart-Engine. Der Bericht kann in der Vorschau vom Benutzer sowohl in HTML- als auch in PDF-Form betrachtet werden (Dencker 2007).

7.3.5.4 Logging

Im ersten Abschnitt dieses Unterkapitels wird das Logging-Modell inkl. der damit zu beantwortenden Fragen beschrieben. Des Weiteren wird der Vorgang des Speicherns kurz vorgestellt und anschließend das zugrunde liegende Logging-Konzept eingeführt. Der letzte Teilabschnitt stellt exemplarisch einen Bericht und die darin enthaltenen Daten vor. Zum Abschluss wird in diesem Abschnitt kurz auf die Umsetzung der Berichte eingegangen.

Logging-Modell

Das in diesem Abschnitt zu beschreibende Logging-Modell wurde auf Basis von angestrebten Ergebnissen erstellt. Das Modell dient dabei als Hilfsmittel, um im gegebenen Kontext bspw. folgende Fragestellungen zu beantworten:

a. Wie hoch ist der Gesamtverbrauch eines Schiffes/Hafens?
b. Wie hoch sind Emissionsausstöße eines Schiffes auf einer Strecke?
c. Wie hoch sind die anfallenden Kosten eines Schiffes für Treibstoff, laufenden Betrieb etc.?
d. Wie viel Treibstoff hat ein Schiff innerhalb eines Szenarios getankt?
e. Welche Eingangsdaten wurden beim Erstellen des Szenarios gesetzt?

Ausgehend von diesen Fragen wurde das in Abbildung 51 dargestellte EMF-Modell erstellt.

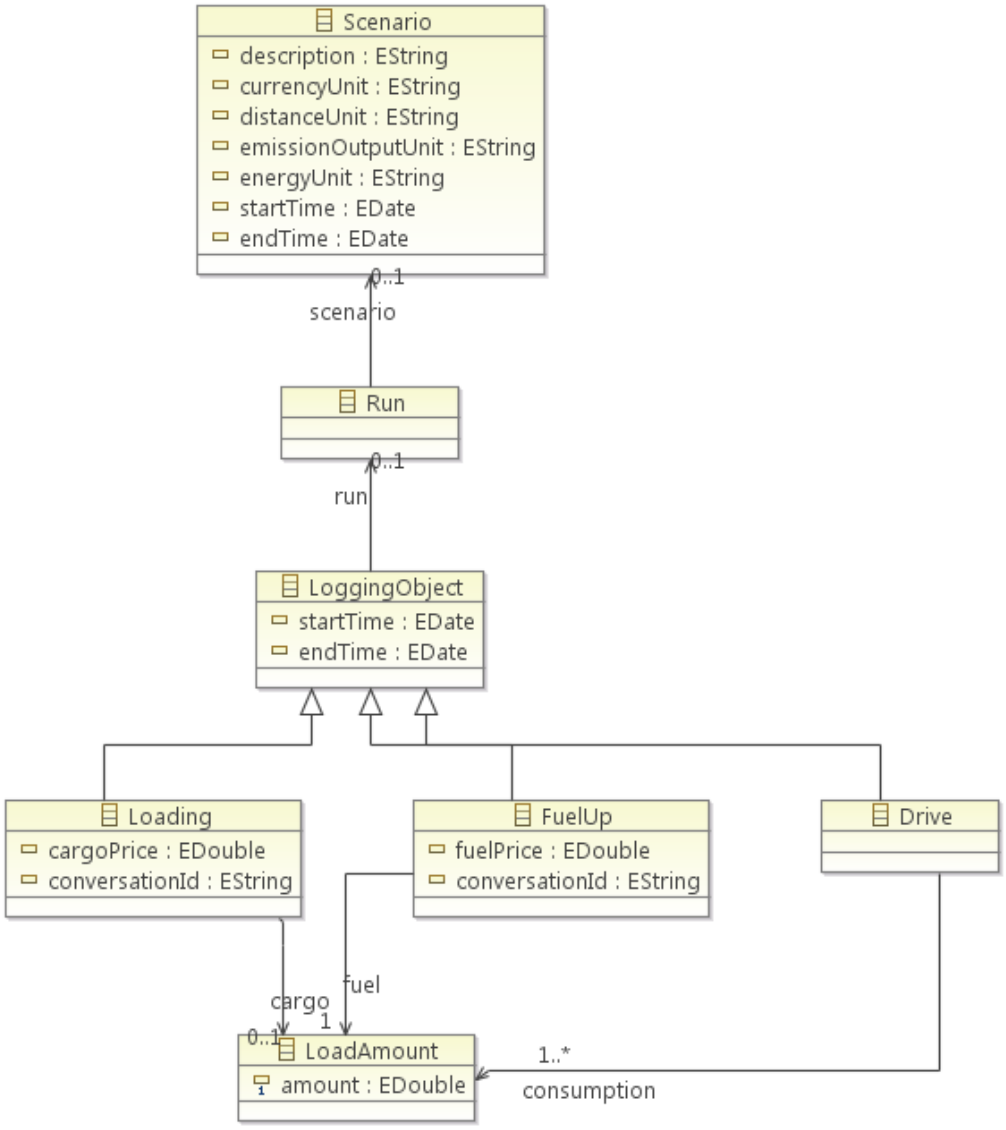

Abbildung 51: Logging-Modell (EMF-Modell)

Das dargestellte Modell hält keinerlei Referenzen auf Daten aus dem Verkehrssimulation-Modell und dem ShipSimualtion-Modell, da diese die Übersichtlichkeit beeinträchtigen würden. Das Modell ist in den Teilbereich der statischen und den der dynamischen Daten aufgeteilt. Die statischen Daten umfassen diejenigen Daten, die szenario-weit gelten und als Eingangsdaten beim Erstellen des Szenarios gesetzt werden. Dies entspricht der Szenario und Run-Klasse, sowie den Daten aus dem Verkehrssimulation-Modell. Die dynamischen Daten hingegen entsprechen den Daten, die spezifisch für einen Simulationsdurchlauf anfallen. Diese Daten müssen gespeichert werden, um die oben genannten Fragestellungen beantworten zu können. Hierzu zählen die Klassen Loading, FuelUp und Drive, sowie deren Hilfsklassen LoggingObject und LoadAmount. Die Klasse Drive bildet die Fahrten innerhalb eines Simulationsdurchlaufs ab. Sie hält den Verbrauch eines Schiffes auf der entsprechenden Strecke fest. FuelUp wiederum entspricht einem Tankvorgang eines Schiffs, bzw. einer Infrastruktur. In der Klasse Loading werden evtl. durchgeführte Ladevorgänge eines Schiffes oder einer Infrastruktur festgehalten. Die Hilfsklasse LoadAmount dient dabei dem Zweck der Mengenbestimmung von Treibstoffen bei Lade- und Tankvorgängen sowie der Verbrauchsbestimmung bei Fahrten. LoggingObject erweitert die drei dynamischen Klassen um einen Zeitaspekt. Dieser ermöglicht Zeitvergleiche, wie z. B. die Tank- oder Fahrtendauer. Alle Klassen des Logging-Modells erweitern direkt oder indirekt die Root-Klasse des Verkehrssimulation-Modells, damit alle Klassen ein ID-Feld erhalten, welches in der Datenbank als Primärschlüssel dient. Dieses Feld wird mittels einer „Auto-Strategie“ (numerische Werte werden automatisch hochgezählt) mit einem eindeutigen Wert versehen, um jedes Objekt identifizierbar zu machen.

Die Abbildung der Vererbungsbeziehungen der Klassen, wurde über die Einstellung „JOINED“ ermöglicht. Dadurch werden alle Klassen einer Hierarchie auf eine Tabelle abgebildet, was bedeutet, dass eine Tabelle für die Basisklasse und eine Tabelle für jede Subklasse erstellt wird (Lahres und Rayman 2009).

Dem Speichervorgang der Logging-Objekte und der dazugehörigen Daten der anderen Modelle in einer Datenbank liegt der Ablauf aus Abbildung 52 zugrunde. Als Basis des Speichervorgangs dient das zuvor beschriebene EMF-Modell. Dieses beinhaltet Metadaten, die von Teneo genutzt werden, um eine sog. orm.xml zu erzeugen. Diese enthält die Struktur der Tabellen und Beziehungen zwischen ihnen, jedoch nicht die konkreten Daten. Sie dient EclipseLink dementsprechend als Grundlage für die initiale Erzeugung des Datenbank-Schemas. EclipseLink dient darüber hinaus zur Persistenz der zu speichernden Daten in der MySQL-Datenbank.

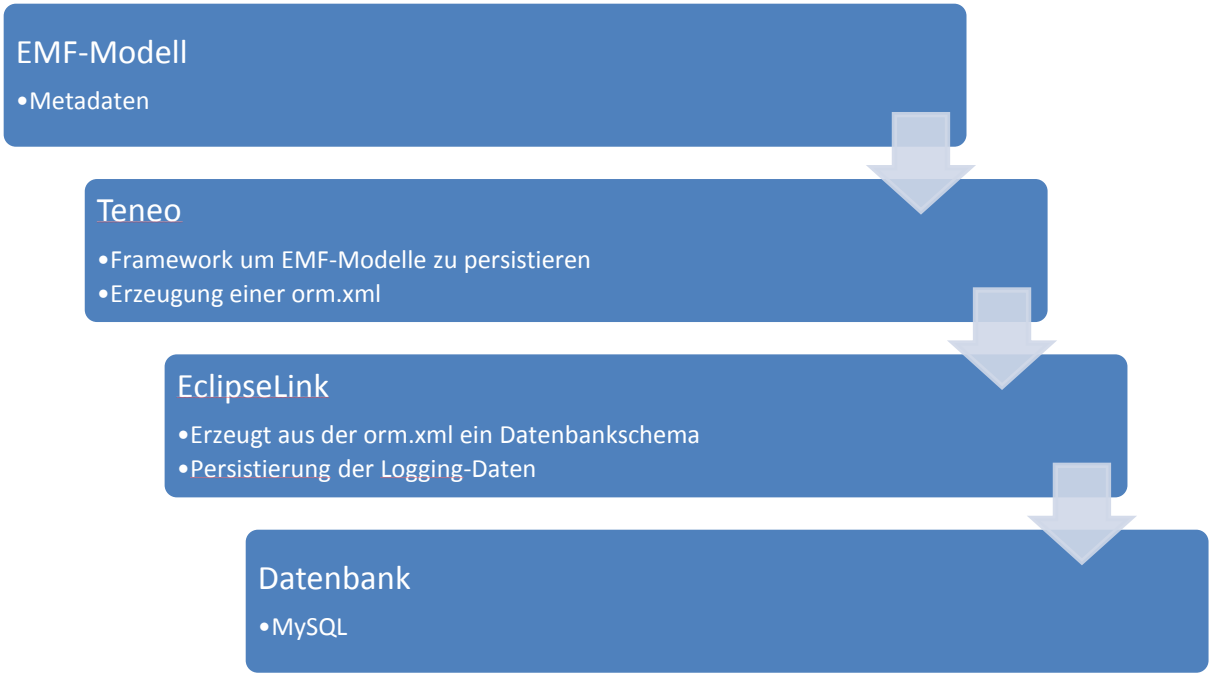

Abbildung 52: Ablauf der Persistierung

Logging-Konzept

Das Logging-Konzept beschreibt die Vorgehensweise, die der Speicherung der zu persistierenden Daten zugrunde liegt. Unter Vorgehensweise werden in diesem Zusammenhang die Logging-Zeitpunkte und die entsprechende Umsetzung des Logging-Vorgangs verstanden.

Die Logging-Zeitpunkte sind entsprechend der Aufteilung in statische und dynamische Daten zu differenzieren. Die statischen Daten werden zu Beginn der Simulation gespeichert, da sie, wie im Abschnitt Logging-Modell erwähnt, die Eingangsdaten für das jeweilige Szenario darstellen. Die entsprechenden Eingangsdaten werden vom Editor und Generator bereitgestellt. Zu diesen Daten zählen unter anderen die Schiffe, Häfen und das Schiffsnetzwerk. Darüber hinaus wird durch diesen Schritt eine ID-Vergabe durchgeführt, die es bspw. ermöglicht, Schiffe einzelnen Tankvorgängen zuordnen zu können.

Die dynamischen Daten hingegen werden erst am Ende eines jeden Simulationsdurchlaufs vollständig persistiert. Diese umfassen die bereits im Abschnitt Logging-Modell eingeführten Fahrten, Lade– und Tankvorgänge. Für die Persistenz dieser Daten sind einzelne Behaviours um Logging-Funkionalitäten erweitert worden. So wurde bspw. das *ShipDriving-Behaviour* erweitert und bietet somit die Funktionalität, am Ende einer Route entsprechende Einträge für eine Fahrt zu speichern. Die Implementierung kann dem folgenden Quellcode-Ausschnitt entnommen werden.

```
1.  @Override
2.      protected void finishSection(RouteSection<ShipConnection>section) {
3.        List<LoadAmount<FuelType>>consumptions = new ArrayList<LoadAmount<FuelType>>();
4.        for (LoadReservation<FuelType>reservation : fuelReservations) {
5.            LoadAmount<FuelType>consumption = BaseFactory.eINSTANCE
6.                    .createLoadAmount();
7.            consumption.setAmount(reservation.getAmount());
8.            consumption.setType(reservation.getLoadType());
9.            consumptions.add(consumption);
10.       }
11.       super.finishSection(section);
12.       Drive driveLog = LoggingFactory.eINSTANCE.createDrive();
13.       driveLog.setConnection(section.getConnection());
14.       driveLog.setShip(getAgent().getLoadOwner());
15.       driveLog.getConsumption().addAll(consumptions);
16..      ...
          getAgent().addLogEntry(driveLog);
        }
```

Abbildung 53: Erweiterung Behaviour

Das zuvor beschriebene Logging-Konzept wird durch Abbildung 54 dargestellt.

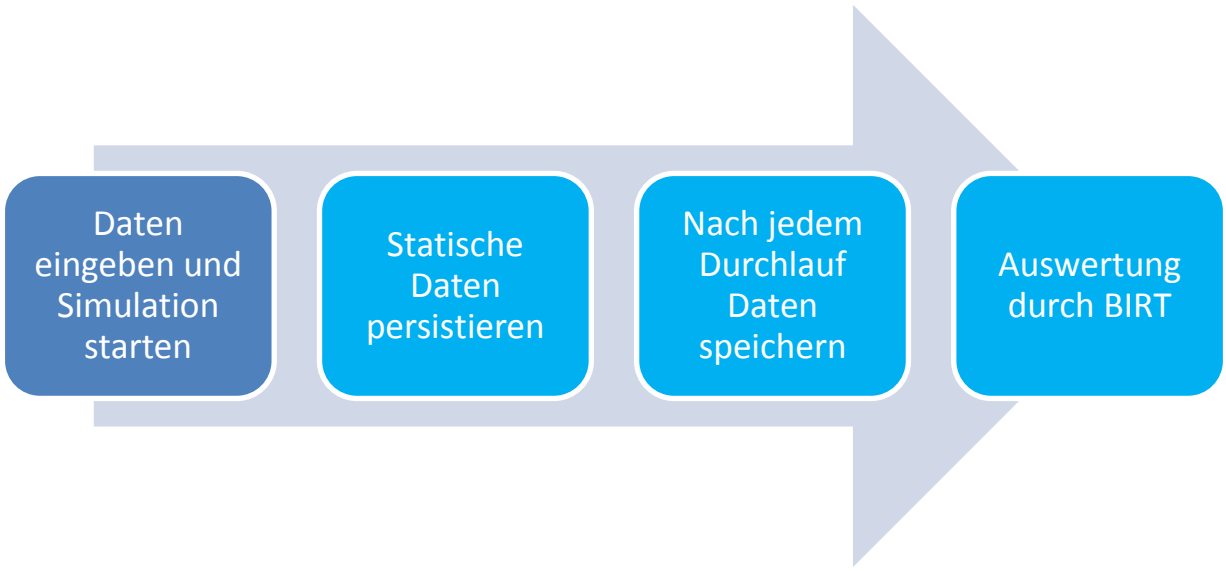

Abbildung 54: Logging-Konzept

7.3.5.5 Reporting

Im nachfolgenden Text wird auf den Aufbau eines Reports bzw. auf die enthaltenen Elemente und deren Daten eingegangen. Dabei wird der Aufbau exemplarisch anhand des allgemeinen Übersichts-Berichts ausführlich erläutert. Die Beschreibungen der weiteren Berichte können dem Anhang A entnommen werden.

Berichte

Der erste Abschnitt des Übersichtsberichts enthält die Szenario-Eingangsdaten, die der Abbildung 56 zu entnehmen sind. Es handelt sich um statische Daten. Diese Daten werden in den Rubriken „Szenarioinformationen“ und „Verwendete Treibstoffe“ in Tabellenform dargestellt. In der Rubrik „Angelegte Objekte“ wird der übersichthalber bewusst auf eine detaillierte Auflistung der einzeln angelegten Daten, wie bspw. Schiffe oder Häfen, verzichtet. Damit die Eingangsdaten dennoch nachvollziehbar bleiben, werden sie in aggregierter Form als Summe aufgelistet.

Abbildung 55: Report Header

Eingangsdaten

Szenarioinformationen

Id	Szenario	Beschreibung	Startzeit	Endzeit
1	MDOEvaluation	-	17.04.2011 00:00	18.04.2012 23:59
93936	LNGEvaluation	-	17.04.2011 00:00	18.04.2012 23:59

Angelegte Objekte

Szenario	Treibstoffanz.	Fahrzeuganz.	Ladungsanz.	Hafenanz.	Streckenanz.	Simulationen
MDOEvaluation	1	4	0	3	6	1
LNGEvaluation	2	4	0	3	6	1

Verwendete Treibstoffe

Szenario	Treibstoff	Co2-Ausstoß pro Einheit	Nox-Ausstoß pro Einheit	Sox-Ausstoß pro Einheit
MDOEvaluation	MDO	605,00	9,50	2,00
LNGEvaluation	LNG	455,00	2,00	0,00
LNGEvaluation	MDO	605,00	9,50	2,00

Abbildung 56: Eingangsdaten des Übersichts-Berichts

Um das Verhältnis der Emissionsanteile des jeweiligen Treibstoffs gegenüberstellen zu können, wurden diese in einem Säulendiagramm zusammengefasst. Neben einer prozentualen Anteilsermittlung der Emissionsanteile des Treibstoffs ermöglicht dies ebenfalls eine Vergleichbarkeit der verwendeten Treibstoffe in den unterschiedlichen Szenarien, was aus Abbildung 57 ersichtlich wird. Sollten weder der Treibstoffname noch der Szenarioname vor-

handen sein, so werden die entsprechenden IDs als Bezeichner an der x-Achse verwendet. Dies wird analog dazu in allen weiteren Berichten ebenso gehandhabt.

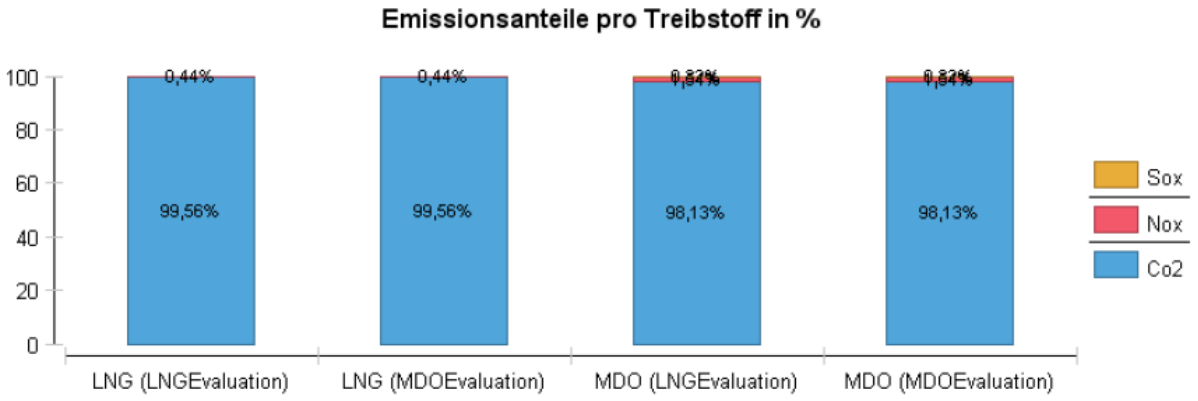

Abbildung 57: Emissionsdiagramm des Übersichts-Berichts

Im letzten Abschnitt werden die ausgewerteten Daten des Szenarios dargestellt. Dies ermöglicht dem Benutzer einen Überblick über den szenarioweiten Treibstoffverbrauch und den daraus resultierenden Emissionsausstößen zu erhalten. Dem Benutzer wird mittels der in Abbildung 58 dargestellten Tabelle und dem Diagramm die Möglichkeit gegeben, die Ausstoßarten unterschiedlicher Treibstoffe szenarioübergreifend und zusätzlich aufgeschlüsselt nach den einzelnen Emissionstypen miteinander zu vergleichen.

Ergebnisse

Szenario-Name	Treibstoff	Verbr. Treibstoff	Gesamter Co2-Ausstoß	Gesamter Nox-Ausstoß	Gesamter Sox-Ausstoß
MDOEvaluation	MDO	1.450.942,22	877.820.041,62	13.783.951,07	2.901.884,44
LNGEvaluation	LNG	1.643.859,15	747.955.913,30	3.287.718,30	0,00
LNGEvaluation	MDO	0,00	0,00	0,00	0,00

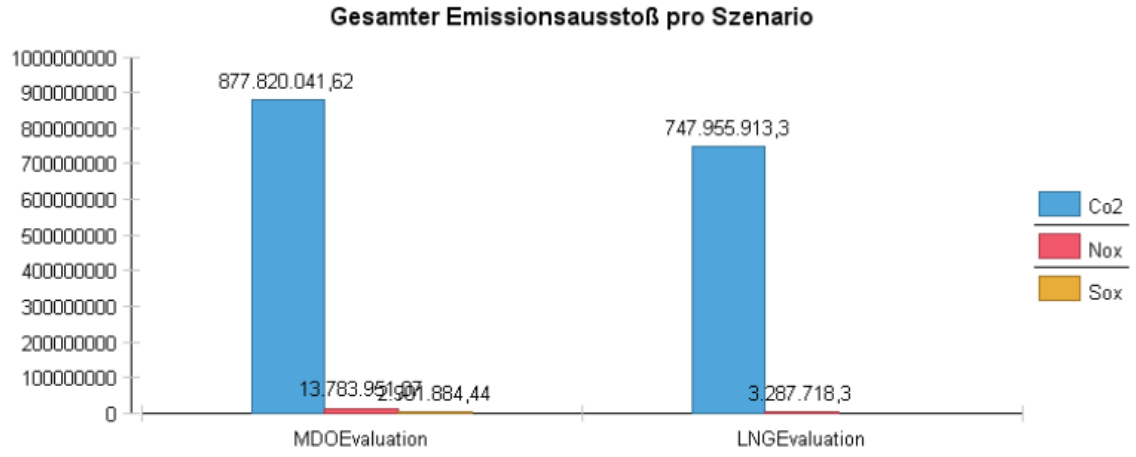

Abbildung 58: Ergebnisse des Übersichts-Berichts

Umsetzung

Zur Erstellung der Berichte wurden einige BIRT-Komponenten verwendet. Die Designs für die einzelnen Berichte wurden unter der Verwendung des in Eclipse integrierten BIRT-Designers angefertigt.

Hierfür wurden zunächst auf Basis der im Abschnitt Logging-Modell aufgestellten Fragestellungen mögliche Report-Elemente wie Tabellen, Diagramme und andere grafische Bestandteile identifiziert und auf entsprechende Report-Designs übertragen. Das Layout der Tabellen wurde nachträglich u. a. durch die Verwendung einer CSS-Datei und dem Setzen von sog. Highlights in BIRT individuell angepasst. Darüber hinaus sind die für die Datenabfrage benötigten SQL-Statements und die dazugehörigen Datasets direkt in den Designs enthalten. Parameter, die ebenfalls in den Designs angelegt wurden, ermöglichen eine Filterung der Daten, die in dem jeweiligen Report angezeigt werden. Gesetzt werden diese Parameter über eine erstellte grafische Oberfläche und den Reports bei ihrer Erzeugung mitgegeben. Somit wird dem Benutzer bspw. die Möglichkeit gegeben, im Übersichtsbericht nur die Daten zu einem Szenario anzuzeigen, sofern er das gewünschte Szenario zuvor über die grafische Oberfläche ausgewählt hat. Die Tabelle der Rubrik „Szenarioinformationen“ aus Abbildung 56 hätte in diesem Fall nur noch einen Eintrag. Eine nachträgliche Erweiterung eines Reports um weitere Daten ist hingegen nicht vorgesehen.

Die Erstellung der Berichte erfolgt direkt über die GUI. Das Standardausgabeformat, welches dem Benutzer grundsätzlich zu Verfügung gestellt wird, ist HTML. Zusätzlich besteht jedoch die Möglichkeit, die Berichte im PDF- oder Excel-Format ausgeben zu lassen.

Die zuvor erwähnte GUI wurde als Eclipse-Plug-In-Projekt realisiert, um sie problemlos in den Schiffseditor integrieren zu können. Dabei wurde die Oberfläche mittels SWT gestaltet und die für die Filterung der Daten notwendige Datenbankverbindung via JDBC bereitgestellt.

7.3.6 Ergebnis des Fallbeispiels

Für eine grafische Analyse der Simulationsergebnisse wurden mit BIRT Berichte erstellt. Diese vergleichen die LNG- und MDO-Szenarien bzgl. des Kraftstoffverbrauchs und der dadurch verursachten Emissionen und Kosten.

Im Folgenden werden zuerst die Ergebnisse der Simulation vorgestellt und anschließend kritisch beleuchtet.

7.3.6.1 Allgemeine Ergebnisse

Im allgemeinen Teil des Reports wird ein Vergleich der Szenarien anhand des Treibstoffverbrauchs, der zurückgelegten Distanz und des daraus resultierenden Verbrauchs pro Kilome-

ter vorgenommen. Der Treibstoffverbrauch wird hier in Litern gemessen. Da eine größere LNG-Tankkapazität angenommen wurde, sind die zurückgelegten Distanzen sehr ähnlich. Nur die Frisia 1 führt einige Bunkerfahrten mehr im MDO-Szenario durch.

Die Frisia 1 verbraucht ca. 60.000 Liter Treibstoff im LNG-Szenario mehr. Dies kommt daher, dass die Energiedichte von LNG nur etwa der Hälfte der von MDO entspricht. Die zurückgelegte Distanz ergibt sich aus der Summe der Strecken der Fahrpläne und den zusätzlich notwendigen Tankfahrten am Ende eines Tages (siehe Abschnitt 7.3.2). Der Verbrauch pro km ermöglicht den einfachen Vergleich von Szenarios.

Allgemein

Schiff	Szenario	Treibstoff	Treibstoffverbrauch	Zurückgelegte Distanz	Verbrauch pro KM
Frisia 1 I	MDOEvaluation	MDO	488.748,58	28.009,20	17,45
Frisia 2 I	MDOEvaluation	MDO	9.889,11	591,60	16,72
Frisia 4 I	MDOEvaluation	MDO	871.239,73	29.957,40	29,08
Frisia 5 I	MDOEvaluation	MDO	81.064,80	5.865,00	13,82
Frisia 1 I	LNGEvaluation	LNG	548.635,56	27.907,20	19,66
Frisia 2 I	LNGEvaluation	LNG	13.176,39	591,60	22,27
Frisia 4 I	LNGEvaluation	LNG	981.568,38	29.957,40	32,77
Frisia 5 I	LNGEvaluation	LNG	100.478,82	5.865,00	17,13

Abbildung 59: Allgemeine Tabelle

7.3.6.2 Kosten

Die Kostentabelle stellt die aggregierten Gesamttreibstoffkosten dar. Diese werden durch eine Multiplikation des Treibstoffverbrauchs mit dem Preis pro Liter des entsprechenden Treibstoffs errechnet. Für diese Tabelle wurde ein MDO-Preis von 0,8 € und LNG-Preis von 0,3 € pro Liter angenommen. Die Kosten für Treibstoff im LNG-Szenario betragen nur ca. 42 % gegenüber dem MDO-Szenario.

Kosten

Schiff	Szenario	Treibstoff-kosten
FRISIA 1 I	MDOEvaluation	390.998,86
FRISIA 2 I	MDOEvaluation	7.911,29
FRISIA 4 I	MDOEvaluation	696.987,83
FRISIA 5 I	MDOEvaluation	64.851,84
FRISIA 1 I	LNGEvaluation	164.590,67
FRISIA 2 I	LNGEvaluation	3.952,92
FRISIA 4 I	LNGEvaluation	294.470,51
FRISIA 5 I	LNGEvaluation	30.143,65

Abbildung 60: Kostentabelle

In dem Balkendiagramm wird die Kostentabelle zum besseren Vergleich visualisiert. Hier werden die Kosteneinsparungen im LNG-Szenario noch deutlicher.

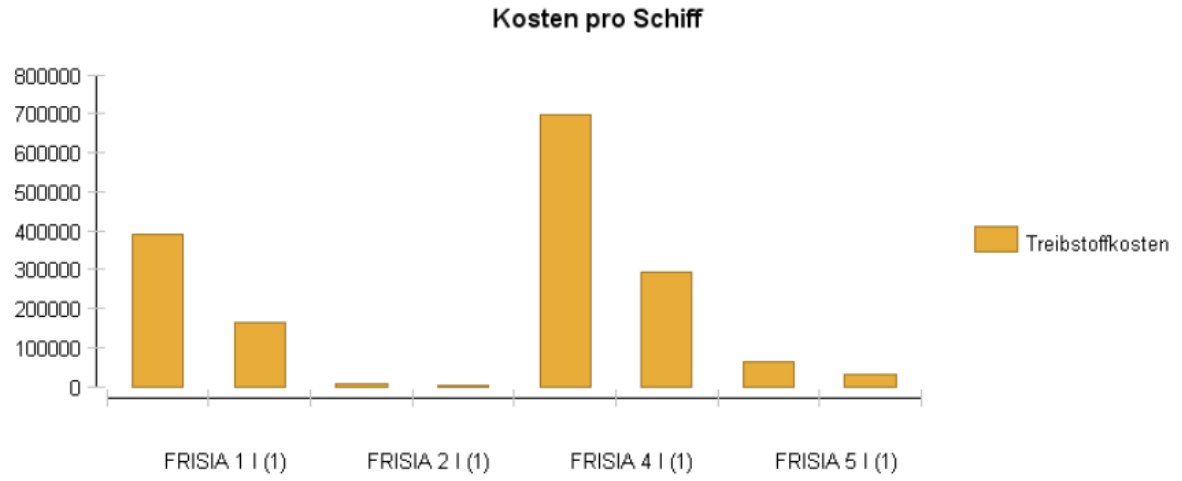

Abbildung 61: Kostenbalkendiagramm

7.3.6.3 Emissionen

Die Emissionen sind ein weiterer Untersuchungsgegenstand. Zur besseren Visualisierung ist hier eine logarithmische Skalierung der y-Achse gewählt worden. Der Unterschied an CO_2 ist sehr gering, doch im LNG-Szenario kann ein großer Teil der NO_x-Emissionen eingespart werden und SO_x-Emissionen entstehen gar nicht mehr.

Ergebnisse

Szenario-Name	Treibstoff	Verbr. Treibstoff	Gesamter Co2-Ausstoß	Gesamter Nox-Ausstoß	Gesamter Sox-Ausstoß
MDOEvaluation	MDO	1.450.942,22	877.820.041,62	13.783.951,07	2.901.884,44
LNGEvaluation	LNG	1.643.859,15	747.955.913,30	3.287.718,30	0,00
LNGEvaluation	MDO	0,00	0,00	0,00	0,00

Abbildung 62: Emissionstabelle

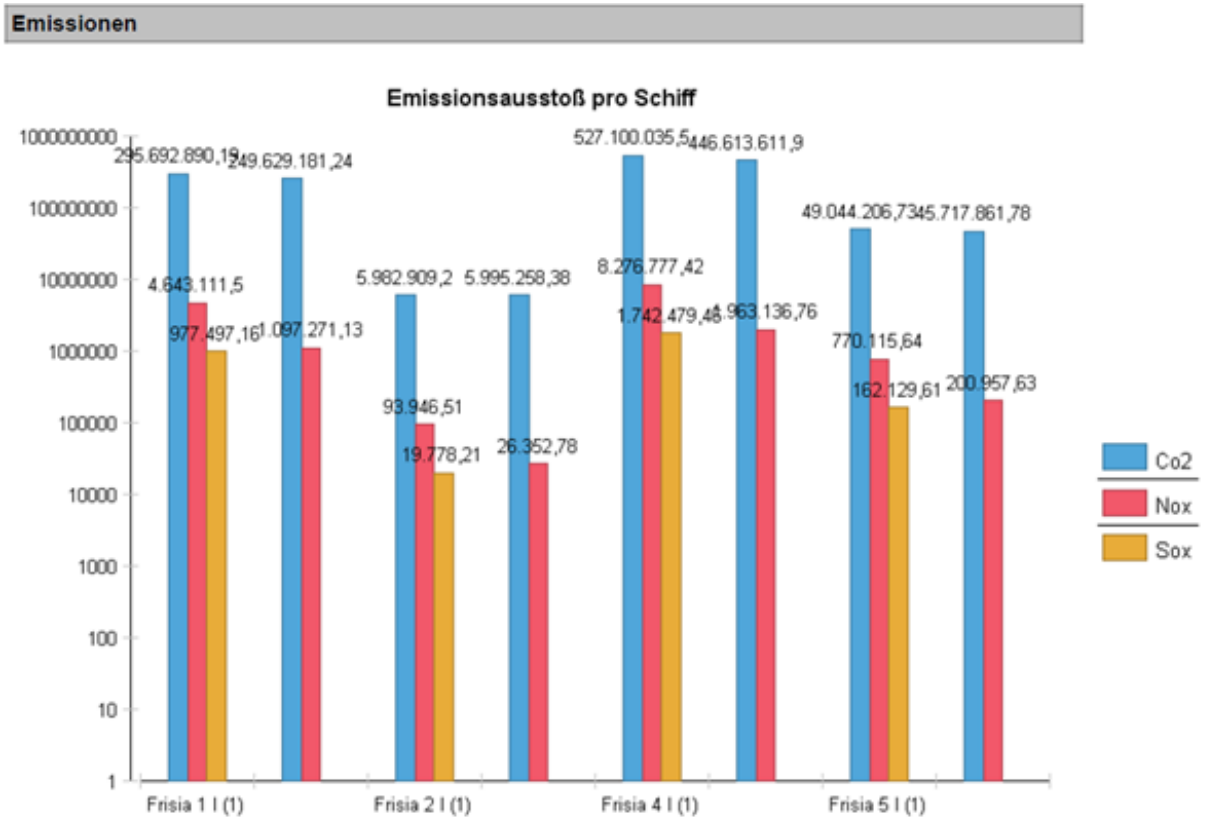

Abbildung 63: Emissionsbalkendiagramm

Ein Drill-Down auf die Emissionen erlaubt eine Auswertung der Emissionen pro Strecke. Besonders für die Auswertung von Emissionen in ECAs oder im Nationalpark Wattenmeer ist diese Ansicht sehr hilfreich. Es können sowohl die Emissionen im Hafen als auch die Emissionen auf den Strecken genau untersucht werden.

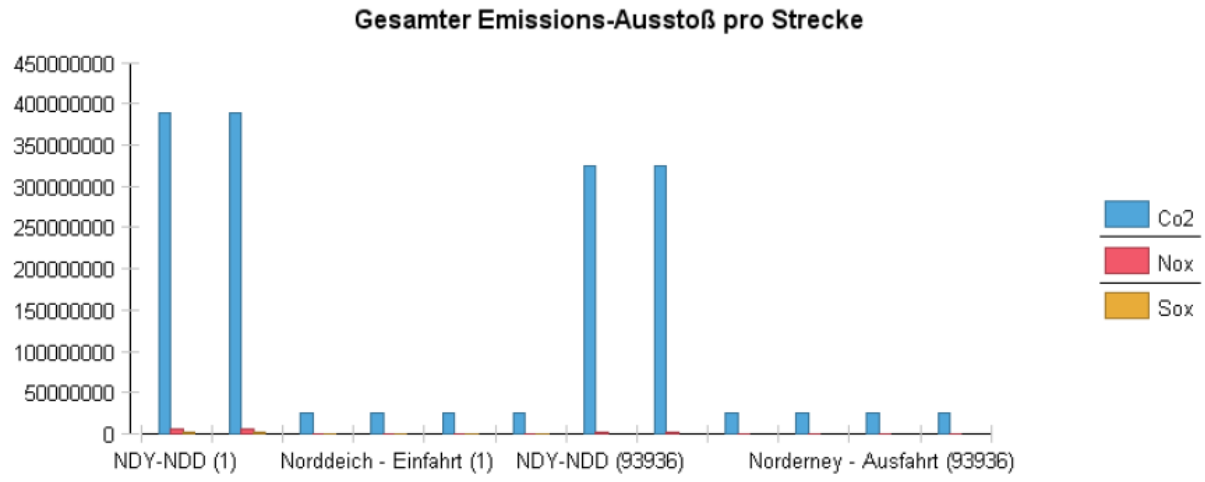

Abbildung 64: Gesamter Emissions-Ausstoß pro Strecke

7.4 Fazit des Fallbeispiels

Die Auswertung des MDO-Szenarios lässt auch einen Vergleich mit den von der Frisia-Reederei zur Verfügung gestellten Echtdaten zu. Aus Tabelle 21 geht hervor, dass die Simulation realitätsgetreue Daten errechnet. Auf Grundlage besserer Daten könnten noch genauere Werte simuliert werden.

Tabelle 21: Vergleich der Echtdaten mit den Simulationsdaten

	Echtdaten in l/km	**Simulationsdaten in l/km**
Frisia 1	17,64	17,45
Frisia 2	16,7	16,72
Frisia 4	24,08	29,08
Frisia 5	12,75	13,82

Ein Vergleich mit LNG-Echtdaten ist leider nicht möglich, da keinerlei Daten bzgl. des Verbrauchs von LNG-Fähren erhoben werden konnten. Da aber ebenfalls realistische Annahmen für die LNG-Motoren getroffen wurden (siehe Abschnitt 7.2), kann auch hier von einem realistischen Ergebnis ausgegangen werden.

Auf Grundlage einer besseren Datenbasis könnten auch weitaus genauere Simulationen vorgenommen werden. Besonders die in der Datenerhebung beschrieben Funktionen zur Berechnung der Leistungskurven der Motoren und der Umsetzung der Energie in die Bewegung des Schiffs sind elementar für die Genauigkeit der Berechnungen.

Auch für die Berechnung der Kosten müssen besonders im LNG-Szenario eine Reihe von Annahmen getroffen werden. Die Kosten für die Bereitstellung einer LNG-Infrastruktur würden wahrscheinlich an die LNG-Abnehmer weitergegeben werden und damit den Preis erhöhen. Die Kosten einer solchen Infrastruktur würden wahrscheinlich auch weit über den Kosten einer MDO-Bunkerstation liegen.

Durch das größere Volumen der LNG-Tanks müssten Platzeinbußen bei Fahrzeugen hingenommen werden. Für Frisia könnte dies eine verringerte Transportkapazität der Schiffe zur Folge haben. Da hier aber keinerlei Informationen zur Verfügung stehen, wurde dieser Aspekt in der Simulation nicht sekundär betrachtet.

7.5 Erreichungsgrad der Anforderungen

In diesem Abschnitt werden die in Kapitel 4 gestellten Anforderungen mit dem Ergebnis verglichen. Dazu werden zuerst die funktionalen Anforderungen von JASON, dem Verkehrsmodell und der Verkehrssimulation herangezogen und im Anschluss die nicht funktionalen Anforderungen mit der Implementierung verglichen.

7.5.1 Funktionale Anforderungen

7.5.1.1 JASON

JASON stellt eine Agentenumgebung dar, mit der es möglich ist, agentenbasierte Simulationen durchzuführen. In den Abschnitten 5.3 und 6.1 - Entwurf und Implementierung des Multi-Agenten-Frameworks JASON - wird erläutert, wie es ermöglicht wird, dass Agenten in der Lage sind, gleichzeitig zu agieren, Dienste anzubieten und Dienste von anderen Agenten nutzen zu können. Weiterhin wird dort auch die Nutzung des Schedulers für eine eigene Simulationszeit und eines Kommunikationssubsystem zum Austauschen von Nachrichten dargelegt. Mit diesen Eigenschaften sind alle funktionalen Anforderungen zur Agentensimulation erfüllt.

7.5.1.2 Verkehrsmodell

Das Verkehrsmodell wird in den Abschnitten 5.4 und 6.2 detailliert beschrieben. Es wird dargelegt, dass die Grundlage der Verkehrsinfrastruktur ein Streckennetz darstellt, welches aus Teilstrecken bestehen kann und damit Infrastrukturstandorte verbindet. An solchen Infrastrukturstandorten können Tankstationen mit Tanks und Verladestationen mit Stauraum existieren, die in der Lage sind, Treibstoff und Frachtgut aufzunehmen und abzugeben. Diese Güter haben ein Gewicht und ein Volumen, um Stauraum und Tanks realistisch darstellen zu können.

Weiterhin beinhaltet das Modell Fahrzeuge. Diese haben Stauräume für Ladung und Tanks für Treibstoff. Treibstoffe werden dabei von Motoren verbraucht, die für einen bestimmten Treibstoff ausgelegt sind, um ein Fahrzeug auf einem Streckennetz fortzubewegen. Fahrzeuge haben, gegeben durch die Motoren, bestimmte Leistungsmerkmale, wie aktuelle Leistung, maximale Leistung und Geschwindigkeit. Treibstoffe besitzen bestimmte Merkmale wie Kohlenstoffdioxide, Schwefeloxide und Stickoxide, die bei Verbrauch des Treibstoffs emittiert werden. Der Brennwert des Treibstoffs ermöglicht eine realitätsgetreue Umsetzung des Treibstoffs durch einen Motor zu Energie.

Zusätzlich können Fahrpläne angelegt werden, welche unterschiedliche Aktionen an bestimmten Orten und zu bestimmten Zeiten beinhalten.

7.5.1.3 Verkehrssimulation

In den Abschnitten 5.5 und 6.3 Entwurf und Implementierung der Verkehrssimulation wird deren Aufbau und Funktionalitäten detailliert erläutert. Es wird der Geoagent beschrieben, der die Streckennetze verwaltet und verschiedene Geoservices anbietet. Weiterhin werden der Infrastrukturagent und der Fahrzeugagent beschrieben. Infrastrukturagenten sind in der Lage, Treibstoff und Frachtgut zu beladen und zu entladen und dabei die Verladegeschwindigkeit, die Verladestelle und die Menge zu verhandeln. Fahrzeugagenten sind in der Lage, Fahrzeuge zu steuern und kürzeste Routen zu finden. Um Strecken abzufahren, können Fahrzeugagenten die notwendige Motorleistung ihres Fahrzeugs steuern. Dabei wird entsprechend Treibstoff verbraucht. Die Fahrzeugagenten sind in der Lage, einen Fahrplan abzuarbeiten, Treibstoff aufzunehmen und Frachtgut zu laden und entladen. Mit diesen Funktionalitäten sind alle funktionalen Anforderungen erfüllt worden.

7.5.2 Nicht-funktionale Anforderungen

Im Rahmen der Werkzeugauswahl standen aufgrund der nicht-funktionalen Anforderungen keine Wahlmöglichkeiten zur Verfügung. Die Anforderungen, dass die Implementierung in Java und die Modellierung mit EMF stattfinden sollen, wurden vom Lehrkörper gefordert.

Kommentare wurden zu allen Klassen in englischer Sprache verfasst und stehen im Code zur Verfügung. Die hohe Qualität der Dokumentation erleichtert die Verwendung des Frameworks.

8 Zusammenfassung und Ausblick

Im Folgenden werden die Ergebnisse der Projektgruppe Mobility der Abteilung Business Engineering der Universität Oldenburg zusammengefasst. Anschließend wird ein Ausblick auf mögliche Erweiterungen der erstellten Komponenten gegeben.

8.1 Zusammenfassung

Im Rahmen des Projektes wurde eine agentenbasierte Simulationsumgebung entwickelt mit der Verkehrssysteme analysiert werden können.

Da bekannte Multi-Agenten-Frameworks nicht den gestellten Anforderungen entsprachen, wurde das agentenbasierte Simulationsframework JASON entwickelt. Es kombiniert die Stärken von MASON und JADE. MASON bietet eine Umgebung für diskrete und eventbasierte Agentensimulationen. Die Stärke von JADE liegt in der standardisierten Kommunikation zwischen Agenten. Somit bietet JASON eine abstrakte und performante Umgebung für Agentensimulationen, in der Agenten gleichzeitig agieren können und über ein nachrichtenbasiertes System kommunizieren. Es eignet sich damit für die Analyse von komplexen Sachverhalten.

Um Verkehrssysteme abbilden zu können, wurde ein modellbasiertes Vorgehen zur Erweiterung von JASON zu einem Verkehrssimulationswerkzeug durchgeführt. Das dabei entstandene Verkehrsmodell wurde in EMF entwickelt und bietet die Grundlagen, um Verkehrsumgebungen abbilden zu können. Alle notwendigen Bestandteile eines Verkehrssystems wurden dabei weitestgehend implementiert, ohne Freiheitsgrade für Erweiterungen einzuschränken.

In der Anforderungsanalyse wurde festgestellt, dass die Konfiguration von Verkehrssimulationen ebenfalls eine zentrale Komponente von Simulationsumgebungen darstellt. Daher wurde eine auf dem EMF-Editor basierende Komponente erstellt, über die einerseits die Dateneingabe für eine Verkehrssimulation, andererseits eine Validierung der Simulationsdaten und das Starten der Simulation ermöglicht wird.

Parallel zur Implementierung wurde in Zusammenarbeit mit der Reederei Frisia ein Szenario erarbeitet, aufgrund dessen eine Schiffssimulation erstellt wurde. Mit dieser wurde die Anwendbarkeit der Verkehrssimulation inklusive des Verkehrsmodells für konkrete Verkehrssimulationen evaluiert.

Zu diesem Zweck wurde die Verkehrssimulation um die domänenspezifischen Eigenschaften der Schifffahrt erweitert. Als Grundlage der Erweiterung erfolgte eine Datenerhebung. Fokussiert wurde dabei die Abbildung des Treibstoffverbrauches durch die Motoren der Schiffe. In diesem Bereich konnten zwar Daten erhoben werden, doch an verschiedenen Stellen mussten Annahmen getroffen werden und Schätzdaten in die Simulation übernommen werden.

Die Darstellung der Komponenten eines Schiffsszenarios auf Grundlage des Verkehrsmodells belegt die Nutzbarkeit des erarbeiteten Modells. Sowohl die Darstellung der Schiffe auf der Basis eines Fahrzeugs als auch die Abbildung der Häfen auf Grundlage der Infrastrukturstandorte waren problemlos möglich. Da die Simulation unterschiedliche Tankgeschwindigkeiten ermöglicht, kann eine realitätsgetreue Betankung durch Hafeninfrastrukturen oder Tanklaster dargestellt werden. Aufgrund unzureichender Daten bzgl. der Kosten innerhalb des simulierten Szenarios wurden diese nicht simuliert, sondern im Anschluss in aggregierter Form in die Darstellung der Ergebnisse aufgenommen.

Ein anderer Erweiterungspunkt ist der Editor. Der Editor wurde ebenfalls auf den beschriebenen Anwendungsfall der Schifffahrt spezialisiert. Die Konfiguration der Schiffssimulation wurde hierdurch erleichtert.

Zur Auswertung wurden die durch die Simulation erzeugten Daten, wie z. B. der Verbrauch der Schiffe, in eine Datenbank geschrieben. Dort stehen sie persistent zur weiteren Verarbeitung zur Verfügung. Mithilfe des Business Intelligence and Reporting Tools wurden aus diesen Daten Reports erzeugt, die eine einfache Analyse ermöglichen. Sowohl aggregierte Gesamtdaten als auch detaillierte Objektdaten ermöglichen einen ganzheitlichen Blick auf die Ergebnisse der Simulation.

Mit der Anwendung des Frameworks auf ein konkretes Szenario konnte somit das erarbeite Konzept validiert werden. Alle implementierten Teile des Projektes werden in dem Beispielszenario integriert und stellen zusammen ein Werkzeug dar, das zur Analyse von langjährigen, komplexen und dynamischen Problemszenarien herangezogen werden kann.

8.2 Ausblick

Die in den vorherigen Kapiteln vorgestellten Komponenten können um weitere Funktionalitäten erweitert werden. Mögliche Erweiterungen werden in diesem Abschnitt erläutert.

Eine Erweiterungsmöglichkeit von JASON stellt das Einbinden zusätzlicher Protokolle zur Kommunikation zwischen Agenten dar. So könnten bspw. Auktionen implementiert werden, um eine weitere Verhandlungsart zwischen Agenten zu bieten.

JASON bietet derzeit keine grafischen Darstellungen. Es könnte so erweitert werden, dass es ähnlich wie MASON die Simulationszustände visualisiert. MASON bietet Funktionen zum Pausieren, Wiederaufnehmen und Zwischenspeichern von Simulationen, die ebenfalls in JASON eingefügt werden könnten.

Im Kontext der grafischen Darstellung wäre eine Erweiterung der Verkehrssimulation möglich, sodass Bewegungen von Fahrzeugen innerhalb einer Simulation visuell dargestellt werden können. Auch wäre es denkbar, in weiteren Arbeiten Konzepte zu entwickeln, um die Erstellung von Berichten zur Auswertung einer Verkehrssimulation zu unterstützen. Hierbei muss beachtet werden, dass für verschiedene Domänen unterschiedliche Daten relevant sind und je nach Fragestellung individuelle Berichte erstellt werden müssen.

Im Rahmen der Auswertung von Simulationen wäre es zudem möglich, die Verwendung von Metaheuristiken (Sauer 2011) zu unterstützen, um so z. B. die Eingabeparameter für den Durchlauf einer Simulation anzupassen, sodass diese hinsichtlich einer bestimmten Zielvorgabe optimiert werden.

Das Verkehrssimulationsframework bietet in der aktuellen Umsetzung Agenten für Fahrzeuge und Infrastrukturen. Es wäre denkbar weitere Agenten und Behaviour zur Verfügung zu stellen. Bspw. könnte der GeoAgent Baustellen, Staus oder Straßensperren auf dem Streckennetz erzeugen, die dann von den Fahrzeugagenten beim Fahren berücksichtigt werden. Eine weitere Möglichkeit ist das Einfügen von meteorologischen Daten, wie z. B. Wind oder Strömung, welche das Verhalten der Agenten beeinflussen.

Wie in der Evaluation erläutert, beruhen einige Werte im Frisia-Szenario auf Annahmen, während andere Bereiche, wie bspw. Ladungskosten und -gewicht, gar nicht betrachtet werden. Damit die Ergebnisse für die Frisia-Reederei aussagekräftiger werden, müssten zusätzliche Daten erhoben und in der Simulation berücksichtigt werden. Außerdem wäre es möglich weitere Fragestellungen wie die optimale Platzierung von Bunkern zu untersuchen.

Derzeit wird lediglich der Schiffsverkehr der Frisia-Reederei betrachtet. Ein weiteres Szenario aus dem Verkehrsbereich ist der Umstieg von Fahrzeugen mit Verbrennungsmotor auf Elektromotoren. Dieses Szenario wird z. B. vom Personal Mobility Center (PMC) für die Modellregion Bremen/Oldenburg betrachtet. In diesem Rahmen werden Simulationen

ausgeführt, um u. a. mögliche Standorte für Ladestationen der Fahrzeuge zu bestimmen. Diese Simulationen werden momentan mit MASON ausgeführt und basieren auf einem eigenen Modell. Hier wäre eine Portierung auf das in der Projektgruppe erstellte Verkehrssimulations-Framework möglich, um bspw. nachrichtenbasierte Verhandlungen zwischen Fahrzeugen und Ladestationen zu ermöglichen.

VI. Glossar

Nachfolgend sind wesentliche Begriffe dieses Dokumentes zusammengefasst und erläutert. Eine ausführliche Erklärung findet sich jeweils in den einführenden Abschnitten.

Agenten
Ein Software-Agent bezeichnet ein Computerprogramm, das zu gewissem eigenständigem Verhalten fähig ist (Telecom Italia 2003).

Behaviour
Behaviours beinhalten die Logik der Agenten (Wesche 2004).

Cascading Style Sheet (CSS)
Ist eine deklarative Sprache um die Trennung von Inhalt und Layout bei der Gestaltung von Webseiten zu ermöglichen (Thiele und Thiele 2012).

Eclipse
Eclipse ist als führende Java Integrated Development Environment (IDE) bekannt (Steinberg, Budinsky und Merks 2008).

Eclipse Modeling Framework
Das Eclipse Modeling Framework (EMF) ist ein Modellierungsframework, das die Möglichkeit bietet, effizienten, korrekten und leicht anpassbaren Java Code aus strukturierten Datenmodellen zu generieren (EclipseFoundation 2010).

Emission Controlled Area (ECA)
Sonderzonen der Schifffahrt mit speziellen Umweltrichtlinien in Bezug auf die Emissionen (MAGALOC Projekt 2008).

Extensible Markup Language (XML)
Darstellung strukturierter Daten in Form hierarchisch aufgebauter Dokumente (Vonhoegen 2011).

Framework
Das Programmiergerüst, das noch kein fertiges Programm ist, sondern innerhalb dessen eine Anwendung entwickelt werden kann (DATACOM Buchverlag GmbH 2012).

Hypertext Markup Language (HTML)
Bezeichnet die Auszeichnungssprache im Word Wide Web, mit der Inhalte von Webseiten beschrieben und strukturiert werden (Siepermann und Lackes 2012).

Infrastrukturstandort
Ein Punkt der Infrastruktur, der bestimmte Infrastruktur-Einrichtungen abbildet.

Java Agent Development (JADE)
JADE ist ein Java-basiertes Framework zur Entwicklung von Multi Agenten Systemen (Telecom Italia 2003).

Java (Programmiersprache)
Java ist eine objektorientierte Programmiersprache (DATACOM Buchverlag GmbH 2012).

JavaDoc
JavaDoc ist ein Dokumentationswerkzeug, das bei der Entwicklung von Java-Programmen benutzt wird und zum Java Development Kit (JDK) gehört. Durch JavaDoc können aus speziell gekennzeichneten Kommentaren innerhalb des Quelltextes technische Dokumentationen im HTML-Format erstellt werden (DATACOM Buchverlag GmbH 2012).

Java Persistence API (JPA)
JPA ist eine Schnittstelle für Java-Anwendungen, die die Zuordnung und die Übertragung von Objekten zu Datenbankeinträgen vereinfacht (Oracle 2011).

Liquid Natural Gas (LNG)
Liquid Natural Gas ist Erdgas im flüssigen Zustand, das seine Form durch Herabkühlung auf -164 bis -161 °C erlangt und dabei sein Volumen auf 1/600stel des Gaszustandes reduziert (MAGALOC Projekt 2008).

Marine Pollution (MARPOL)
Marine Pollution beschreibt ein internationales Übereinkommen zur Verhütung der Meeresverschmutzung (IMO 2005).

Multi Agent Simulation of Neighboorhoods (MASON)
MASON ist eine Java-Bibliothek und stellt eine Grundlage zur Erstellung von diskreten und stetigen Simulationen bereit (George Mason University 2012).

Metamodell
Eine abstrakte Darstellung eines Modells (Steinberg, Budinsky und Merks 2008).

Modell

Eine abstrakte Abbildung eines Realitätsausschnitts (Springer Gabler 2012).

Multi-Agenten-Systeme (MAS)

Ein Multi-Agenten-System ist ein System aus mehreren Agenten, die kollektiv ein Problem lösen (Weiss 2000).

MySQL

MySQL ist ein relationales Datenbankverwaltungssystem (Oracle Corporation 2012).

Object-relational Mapping (ORM)

Object-relational Mapping (deutsch: objektrelationale Abbildung) ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann. (Lahres und Rayman 2009).

OSGI

Das OSGi-Framework ist eine offene, modulare und skalierbare Integrationsplattform auf Java-Basis, mittels derer die Vernetzung von Endgeräten, die Auslieferung und Installation von Diensten, Informationen und Inhalten sowie die Fernsteuerung, -diagnose und -wartung von Geräten ermöglicht wird. (Fraunhofer IIS 2012)

Pair Programming

Unter Pair Programming wird eine Vorgehensweise verstanden, bei der zwei Entwickler zusammen an einem Computer den Quellcode erstellen. Dies hat den Vorteil, dass ein Programmierer den Code schreibt, während der anderen diesen zeitgleich kontrolliert (Cohn 2010).

Plug-In

Softwarekomponente zur Erweiterung der Funktionalität über eine definierte Datenschnittstelle in das Hauptprogramm (M. Vogel 2012).

Rich Client Platform (RCP)

Die Rich-Client-Plattform (RCP) ist ein Framework zur Entwicklung von Plug-In-basierten Applikationen (ITWissen 2010).

Structured Query Language (SQL)

Ist eine Datenbanksprache um Daten einzufügen, zu bearbeiten, zu löschen und Abfragen auf diesen Daten durchzuführen (Springer Gabler 2012).

Thread

Die kleinste Verarbeitungseinheit bei einem Betriebssystem (Abeck 2005).

Thread-Pool

Ein Thread-Pool ist eine vorab erzeugte Menge an Threads, die zur Bearbeitung von Aufgaben vorgehalten wird. Neue Aufgaben werden durch einen freien Thread aus dem Pool bearbeitet. Nach der Bearbeitung wird der Thread nicht beendet, sondern bleibt zur Bearbeitung weiterer Aufgaben im Thread-Pool erhalten. Hierdurch werden Ressourcen zur Erzeugung neuer Threads eingespart (Louis und Müller 2007).

Workbench

Die Workbench ist ein Plug-In und enthält die grafische Benutzeroberfläche. Es verwendet die Plug-Ins SWT und JFace, welche Bibliotheken für das Fenstersystem bereitstellen (Steinberg, Budinsky und Merks 2008).

VII. Literaturverzeichnis

Abeck, Sebastian. *Kursbuch Informatik I: formale Grundlagen der Informatik und Programmierkonzepte am Beispiel von Java; eine Einführung in die Informatik auf der Grundlage etablierter Lehrbücher.* Karlsruhe: KIT Scientific Publishing, 2005.

Aktiengesellschaft Norden-Frisia AG. *Reederei - Frisia.* 2012. www.reederei-frisia.de (Zugriff am 02 2012).

Bacvanski, Vladimir, und Petter Graff. *Mastering Eclipse Modeling Framework - EclipseCON 2005.* 2005. http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial28.pdf.

Chart Ferox. *Chart Ferox System.* 2012. http://www.chart-ferox.com/systems/systems-lng-systems-ship-refueling.htm (Zugriff am 2012).

Cohn, Mike. *Agile Softwareentwicklung - Mit Scrum zum Erfolg.* Mübchen: Addison-Wesley, 2010.

DATACOM Buchverlag GmbH. *ITWissen.* 2012. http://www.itwissen.info/definition/lexikon/Java.html (Zugriff am 2012).

—. *ITWissen.* 2012. http://www.itwissen.info/definition/lexikon/Schnittstelle-IF-interface.html (Zugriff am 2012).

—. *ITWissen.* 2012. http://www.itwissen.info/definition/lexikon/Framework-framework.html (Zugriff am 2012).

—. *ITWissen.* 2012. http://www.itwissen.info/definition/lexikon/javadoc.html (Zugriff am 2012).

Dencker, Tobias. *Business Intelligence Reporting Tool (BIRT).* Karlsruhe: Universität Karlsruhe, 2007.

Department of Computer Science George Mason University. „Multiagent Simulation and the MASON Library.“ 08 2011. (Zugriff am 02 2012).

Deutsche UNESCO-Kommision. *UNESCO.* 2012. http://www.unesco.de/ (Zugriff am 2012).

Douvier, Stefan W. *MARPOL.* Hamburg: CT Salzwasser Verlag GmbH & Co. KG, 2008.

Eclipse Project Community. *About the Eclipse Project.* 02 2012. www.eclipse.org (Zugriff am 02 2012).

EclipseFoundation. 2010. http://www.eclipse.org/org/index.php#about (Zugriff am 10. 01 2012).

—. *BIRT.* 2012. http://www.eclipse.org/birt/phoenix/intro (Zugriff am 28. Dezember 2011).

—. *EclipseLink Home.* 09. Dezember 2011. http://www.eclipse.org/eclipselink (Zugriff am 02. Februar 2012).

—. *Teneo EclipseLink.* 9. Dezember 2010. http://wiki.eclipse.org/Teneo/EclipseLink (Zugriff am 02. Februar 2012).

ECOpoint Inc. *dieselnet.com.* 2012. http://www.dieselnet.com/standards/inter/imo.php (Zugriff am 2012).

Fraunhofer IIS. *Fraunhofer - Institut für Integrierte Schaltungen IIS.* 2012. http://www.iis.fraunhofer.de/bf/ec/dm/osgi/ (Zugriff am 2012).

Gate terminal. *GATE - nl.* 2012. http://www.gate.nl/ (Zugriff am 02 2012).

George Mason University. *MASON .* 02 2012. http://cs.gmu.edu/~eclab/projects/mason/ (Zugriff am 03 2012).

Gesellschaft für Informatik e.V. *Multiagentensimulation - Gesellschaft für Informatik e.V.* 2012. http://www.gi.de/no_cache/service/informatiklexikon/informatiklexikon-detailansicht/meldung/multiagentensimulation-149.html (Zugriff am 03 2012).

Gläßer, Lothar. *Open Source Software. Projekte, Geschäftsmodelle, Rechtsfragen, Anwenduingsszenarien - was IT-Entscheider und Anwender wissen müssen.* Wiley-VCH, 2004.

Google. *Google Maps.* 2012. http://maps.google.de/maps?hl=de&tab=wl (Zugriff am 2012).

IEEE. *The Foundation forIntelligent Physical Agents.* 2012. http://www.fipa.org/ (Zugriff am 2012).

IMO. *International Maritime Organization.* 03 2012. http://www.imo.org/Pages/home.aspx (Zugriff am 02 2012).

IMO. „MARPOL Annex VI Prevention of Air Pollution from Ships.“ 2005 .

ITWissen. *Rich-Client-Plattform.* 2010.

Lahres, Bernhard, und Gregor Rayman. *Objektorientierte Programmierung.* 2. Auflage. Galileo Press, 2009.

Larman, Craig. *UML und Patterns angewendet- objektorientierte Softwareentwicklung.* Heidelberg, München, Landsberg, Frechen, Hamburg: Hüthig Jehle Rehm GmbH, 2005.

Leitner, Johannes. „Entwurfsphase.“ 2007.

Louis, Dirk, und Peter Müller. *Das Java 6 Codebook.* München: Addison-Wesley Verlag, 2007.

MAGALOC Projekt. „Report from the MAGALOC project.“ 2008.

Olesen, Helge Rordam, Morten Winther, Thomas Ellermann, Jesper Christensen, und Marlene Plejdrup. „Ship emissions and air pollution in Denmark.“ 2009.

Oracle Corporation. *MySQL.* 2012. http://www.mysql.de/ (Zugriff am 2012).

Oracle. *Java Persistence API.* 2011. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html (Zugriff am 02. Februar 2012).

Reederei Norden Frisia AG. Norderney, 2011.

Rolls-Royce. *The use of LNG as fuel for propulsion on board merchant ships.* Paris, 2011.

Rummler, Andreas, und Konrad Voigt. „Abgelegt - Von Äpfel und Birnen (EMF-Modelle mit Teneo in Datenbanken speichern).“ *Eclipse Magazin*, 2009.

—. „Abgelegt – Von Äpfel und Birnen.“ *Eclipse Magazin*, 03 2009.

Sauer, Jürgen. „Intelligente Systeme.“ *Vorlesung an der Universität Oldenburg.* Oldenburg: Universität Oldenburg, 2011.

Scherf, Helmut. *Modellbildung und Simulation dynamischer Systeme.* München: Oldenbourg Wissenschaftsverlag GmbH, 2010.

seo-united.de. *seo-united.de.* 2012. http://www.seo-united.de/glossar/optimierung/ (Zugriff am 2012).

Siepermann, Markus, und Richard Lackes. *Wirtschaftslexikon.* 2012. http://wirtschaftslexikon.gabler.de/Definition/html.html?referenceKeywordName=HyperText+Markup+Language (Zugriff am 27. März 2012).

Springer Gabler. *Gabler Wirtschaftslexikon.* 2012. http://wirtschaftslexikon.gabler.de/ (Zugriff am 2012).

Statoil Research Center. „LNG used to power the ferry "Glutra" in Norway - The world first ferry to run on LNG.“ Stockholm, 2002.

Steinberg, Dave, Frank Budinsky, und Ed Merks. *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

Sygo, Martin. *Plug-In Development.* Universität Karlsruhe, 2007.

—. *Plug-In Development.* Universität Karlsruhe, 2007.

Telecom Italia. „Java Agent DEvelopment.“ 2003. http://jade.tilab.com/index.html (Zugriff am 2012).

Telecom Italia Lab. *Jade.* 02 2012. http://jade.tilab.com/ (Zugriff am 2012).

Thiele, Oliver, und Markus Thiele. *Cascading Style Sheets.* 2012. http://www.goerlitz-webdesign.de/css-stylesheets.html (Zugriff am 27. März 2012).

Vogel. *Eclipse Modeling Framework (EMF) - Tutorial.* 2011. http://www.vogella.de/articles/EclipseEMF/article.html.

Vogel, Martin. *Martinvogel.de.* 2012. http://lexikon.martinvogel.de/plugin.html (Zugriff am 2012).

Volvo. *Spezifikation Volvo Penta D25A MS.* Göteborg, 2004.

Vonhoegen, Helmut. *Einstieg in XML - Grundlagen, Praxis, Referenz.* Bonn: Galileo Press, 2011.

Wattenmeer, Nationalpark Wattenmeer | Niedersächsisches. *Nationalpark Wattenmeer | Niedersächsisches Wattenmeer.* 2012. http://www.nationalpark-wattenmeer.de/sites/default/files/media/images/karte_nlp_nds_wm.jpg (Zugriff am 2012).

Wattenmeer, Niedersächsisches Wattenmeer | Nationalpark. *Niedersächsisches Wattenmeer | Nationalpark Wattenmeer.* 2012. http://www.nationalpark-wattenmeer.de/nds (Zugriff am 2012).

Weiss, Gerhard. *Multiagent Systems. A Modern Approach to Distributed Artifical Intelligence.* Cambridge MA: MIT Press, 2000.

Wesche, Thomas. *Effiziente Trainingsverfahren für die Multi-Agenten-Systementwicklung.* Magdeburg: Diplomarbeit - Otto-von-Guericke-Universität Magdeburg, 2004.

World Ocean Review. *Das Meer - der weltumspannende Transportweg.* Hamburg: maribus GmbH, 2010.

VIII. Anhang

A. Berichte

Neben dem bereits eingeführten Übersichtsbericht wurden noch ein Schiffbericht, ein Streckenbericht und ein Hafenbericht erstellt. Diese weiteren Berichte wurden in Hinblick auf die Beantwortung der in Abschnitt 7.3.5.4 eingeführten Fragen gestaltet.

Die nachfolgenden Abschnitte gehen auf den Aufbau der entsprechenden Berichte und die darin enthaltenen Elemente in Form von Tabellen, Diagramme etc. ein.

Schiffbericht

Der erste Abschnitt des Schiffberichts enthält die allgemeinen Daten zu den gewählten Schiffen. Diese Daten bieten dem Nutzer die Möglichkeit, in tabellarischer Form grundlegende Informationen bezüglich der aggregierten Fahrtvorgänge einzusehen. Darüber hinaus erlauben diese dem Nutzer bspw. einen Rückschluss auf die verbrauchte Treibstoffmenge pro Schiff und Szenario zu ziehen, welche wiederum für die Berechnung der Emissionen herangezogen werden kann. Des Weiteren ist der Nutzer bei entsprechender Auswahl der Daten auch in der Lage, die verbrauchte Treibstoffmenge eines Schiffes in unterschiedlichen Szenarien anzeigen zu lassen.

Allgemein

Schiff	Szenario	Treibstoff	Treibstoffverbrauch	Zurückgelegte Distanz	Verbrauch pro KM
Frisia 1 I	MDOEvaluation	MDO	488.748,58	28.009,20	17,45
Frisia 2 I	MDOEvaluation	MDO	9.889,11	591,60	16,72
Frisia 4 I	MDOEvaluation	MDO	871.239,73	29.957,40	29,08
Frisia 5 I	MDOEvaluation	MDO	81.064,80	5.865,00	13,82
Frisia 1 I	LNGEvaluation	LNG	548.635,56	27.907,20	19,66
Frisia 2 I	LNGEvaluation	LNG	13.176,39	591,60	22,27
Frisia 4 I	LNGEvaluation	LNG	981.568,38	29.957,40	32,77
Frisia 5 I	LNGEvaluation	LNG	100.478,82	5.865,00	17,13

Abbildung 65 - Allgemeine Daten des Schiffberichts

Im zweiten Teilabschnitt werden die mit den Schiffen verknüpften Kosten dargestellt, die als Grundlage für weitere Analysen genutzt werden könnten. Die Kostentabelle wird im Gegensatz zu allen anderen Tabellen nicht nur mit Daten aus der Simulation gefüllt. Zur Berechnung der Treibstoffkosten wurde eine Annahme bzgl. der Kosten pro Einheit MDO und LNG getroffen. In Abbildung 66 werden diese Ergebnisse ersichtlich.

Kosten

Schiff	Szenario	Treibstoff-kosten
FRISIA 1 I	MDOEvaluation	390.998,86
FRISIA 2 I	MDOEvaluation	7.911,29
FRISIA 4 I	MDOEvaluation	696.987,83
FRISIA 5 I	MDOEvaluation	64.851,84
FRISIA 1 I	LNGEvaluation	164.590,67
FRISIA 2 I	LNGEvaluation	3.952,92
FRISIA 4 I	LNGEvaluation	294.470,51
FRISIA 5 I	LNGEvaluation	30.143,65

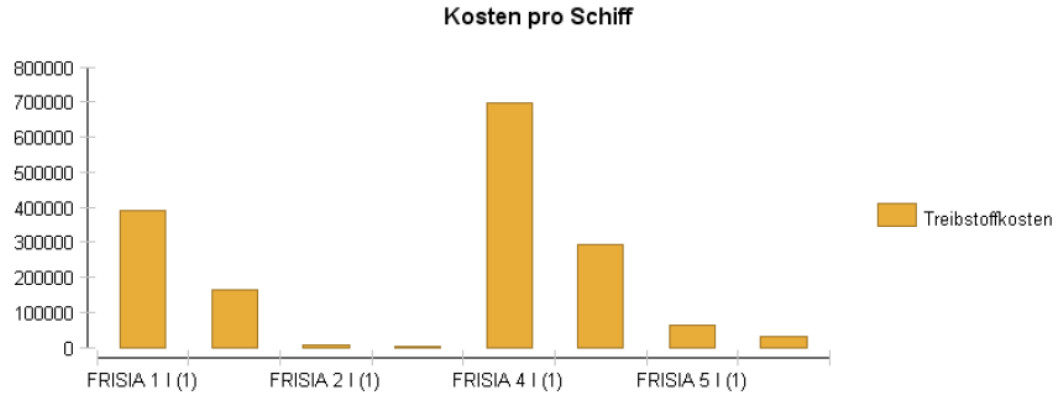

Abbildung 66 - Kostenauflistung des Schiffberichts

Darüber hinaus wird dem Nutzerangeboten, relevante Daten zu den einzelnen Tankereignissen abzulesen. Diese Ergebnisse werden, wie auch schon in den Übersichten zuvor, aggregiert pro Schiff dargestellt. Abbildung 67zeigt bspw. die Tankvorgänge der im Szenario angelegten Schiffe, mit den entsprechenden Tankzeiten in Minuten auf.

Tankvorgänge

Schiffsname	Szenario	Tankzeit	Tankvorgänge
Frisia 1 I	MDOEvaluation	480,05	31
Frisia 2 I	MDOEvaluation	26,32	1
Frisia 4 I	MDOEvaluation	878,98	32
Frisia 5 I	MDOEvaluation	77,89	5
Frisia 1 I	LNGEvaluation	547,36	24
Frisia 2 I	LNGEvaluation	33,41	1
Frisia 4 I	LNGEvaluation	1003,96	25
Frisia 5 I	LNGEvaluation	114,45	5

Abbildung 67 - Tankdaten des Schiffberichts

Im letzten Abschnitt werden die emissionsbezogenen Ergebnisse in Form eines Balkendiagramms dargestellt. Dies ermöglicht dem Nutzer, die von einem Schiff ausgestoßenen Mengen pro Schadstoff miteinander zu vergleichen. Außerdem ist diesbezüglich der Vergleich

zwischen den Schiffen untereinander und bei einer vorherigen Auswahl mehrerer Szenarien sogar übergreifend möglich.

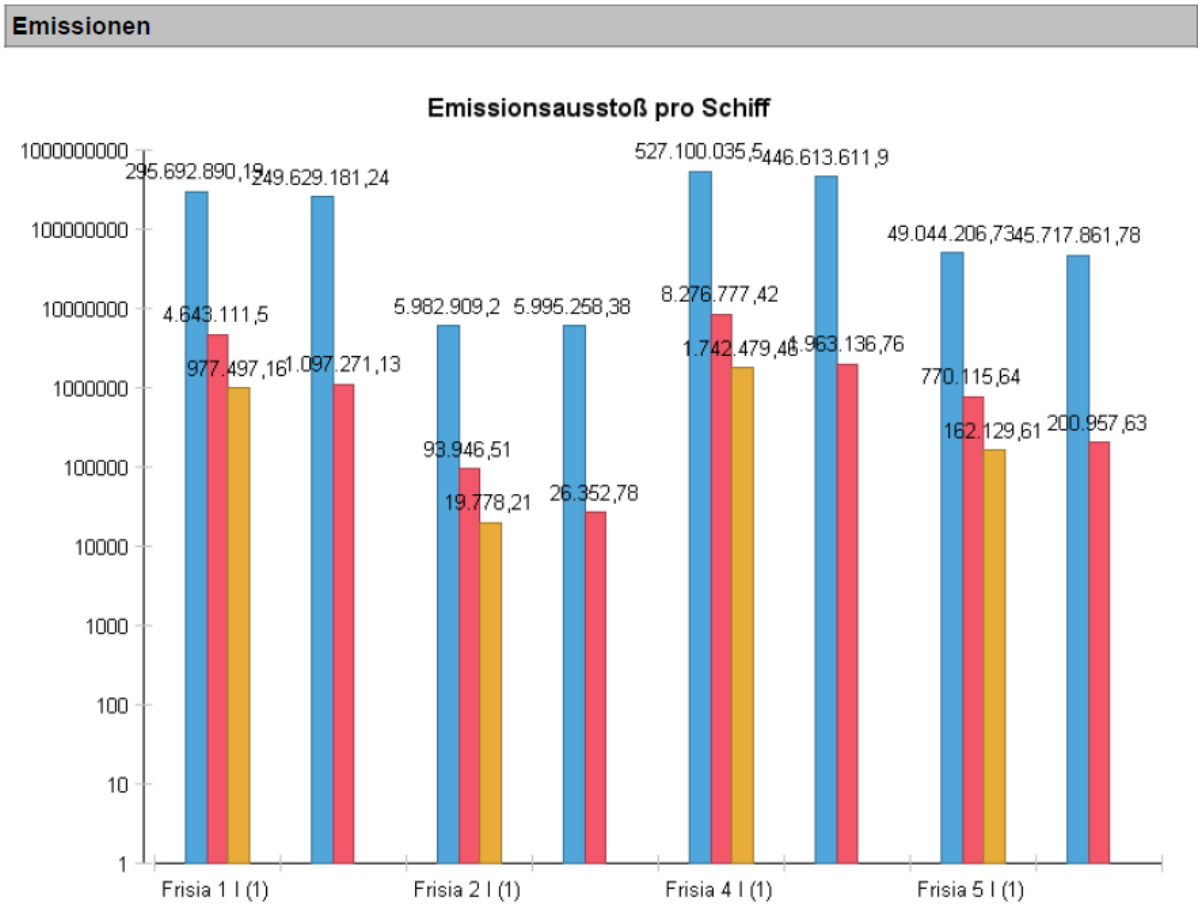

Abbildung 68 - Emissionsdaten des Schiffberichts

Streckenbericht

In dem ersten Abschnitt des Streckenberichts werden die allgemeinen Daten zu den ausgewählten Strecken dargestellt. Diese Daten bieten dem Nutzer die Möglichkeit, in tabellarischer Form grundlegende Daten bezüglich der Strecken einzusehen.

Allgemeine Informationen

Strecken-ID	Streckenname	Szenario	Maximale Geschw.	Distanz	Fahrten	Schiffe
7	NDY-NDD	MDOEvaluation	16,00	9,20	3.153	4
13	Norddeich - Ausfahrt	MDOEvaluation	8,00	0,50	3.155	4
15	Norddeich - Einfahrt	MDOEvaluation	8,00	0,50	3.153	4
16	Norderney - Ausfahrt	MDOEvaluation	8,00	0,50	3.153	4
84615	NDD-NDY	LNGEvaluation	16,00	9,20	3.154	4
84619	Norddeich - Ausfahrt	LNGEvaluation	8,00	0,50	3.154	4
84621	Norddeich - Einfahrt	LNGEvaluation	8,00	0,50	3.152	4
84624	Norderney - Einfahrt	LNGEvaluation	8,00	0,50	3.154	4

Abbildung 69 - Allgemeine Daten des Streckenberichts

Im Abschnitt „Ergebnisse“ werden die erfassten und aggregierten Daten der Streckenbenutzung angezeigt. Dies ermöglicht dem Nutzer, einen Überblick über den streckenbezogenen Treibstoffverbrauch und den daraus resultierenden Emissionsausstößen zu erhalten. Dies geschieht sowohl auf ein als auch übergreifend über mehrere Szenarien bezogen. Letzteres ermöglicht den direkten Vergleich dieser Daten aus unterschiedlichen Szenarien.

Ergebnisse

Strecke	Szenario	Treibstoffname	Treibstoffverbrauch
NDY-NDD	MDOEvaluation	MDO	471.770,34
Norddeich - Ausfahrt	MDOEvaluation	MDO	27.295,10
Norddeich - Einfahrt	MDOEvaluation	MDO	27.277,95
Norderney - Ausfahrt	MDOEvaluation	MDO	27.277,95
NDD-NDY	LNGEvaluation	LNG	581.165,40
Norddeich - Ausfahrt	LNGEvaluation	LNG	34.266,27
Norddeich - Einfahrt	LNGEvaluation	LNG	34.244,83
Norderney - Einfahrt	LNGEvaluation	LNG	34.266,27

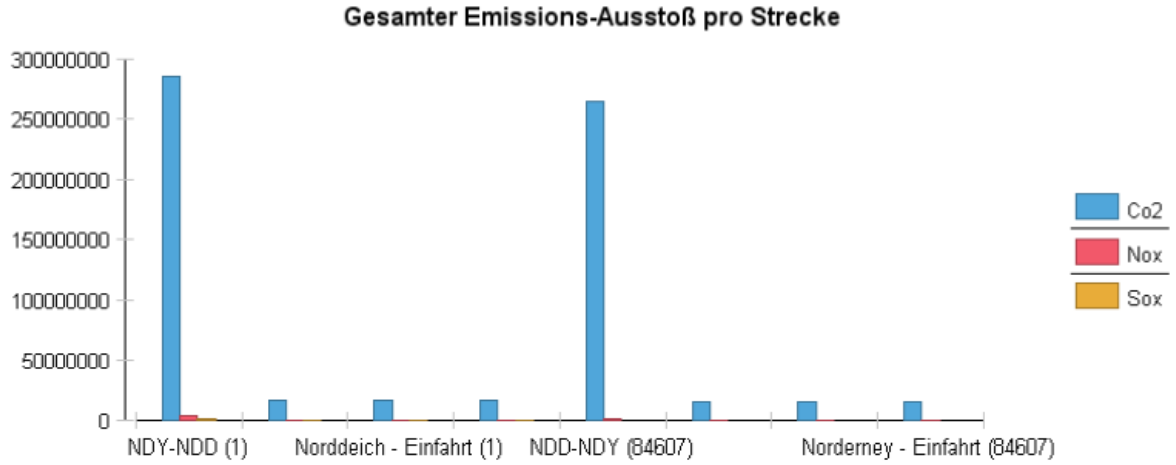

Abbildung 70– Ergebnisse des Streckenberichts

Hafenbericht

Der erste Abschnitt des Hafenberichts enthält sowohl die allgemeinen als auch die Tank- bzw. Treibstoff bezogenen Daten der gewählten Häfen. Diese erlauben dem Nutzer, in tabellarischer und zudem auf Hafenebene aggregierter Form mengen- und zeitbezogene Daten einzusehen. Wie auch beim Schiffbericht können die Preisgrößen aufgrund fehlender Eingangsgrößen nicht mit Daten aus der Simulation gefüllt werden,sondern basieren ebenfalls auf geschätzten Werten.

Infrastrukturen/Häfen

Hafen-ID	Hafen	Szenario	Treibstoff	Menge des Treibstoffs	Einnahmen
21	ip2 at Juist	MDOEvaluation	MDO	363.044,69	1.822.473,00
26	ip2 at Norddeich	MDOEvaluation	MDO	87.527,45	1.822.473,00
84627	ip2 at Juist	LNGEvaluation	LNG	373.265,00	1.822.473,00
84632	ip2 at Norddeich	LNGEvaluation	LNG	187.514,07	1.822.473,00

Tankzeit

Hafen-ID	Hafenname	Tankzeit	Tankauslastung pro Jahr
21	ip2 at Juist	360,26	0,0000069
26	ip2 at Norddeich	86,93	0,0000017
84627	ip2 at Juist	370,75	0,0000071
84632	ip2 at Norddeich	186,13	0,0000035

Abbildung 71 - Allgemeine Daten des Hafenberichts

Den Abschluss des Hafenberichts bildet eine Darstellung der gelieferten Treibstoffmengen der Häfen in Form eines Balkendiagrammes. Damit haben Nutzer die Möglichkeit, die Liefermengen pro Hafen der gewählten Szenarien miteinander zu vergleichen.

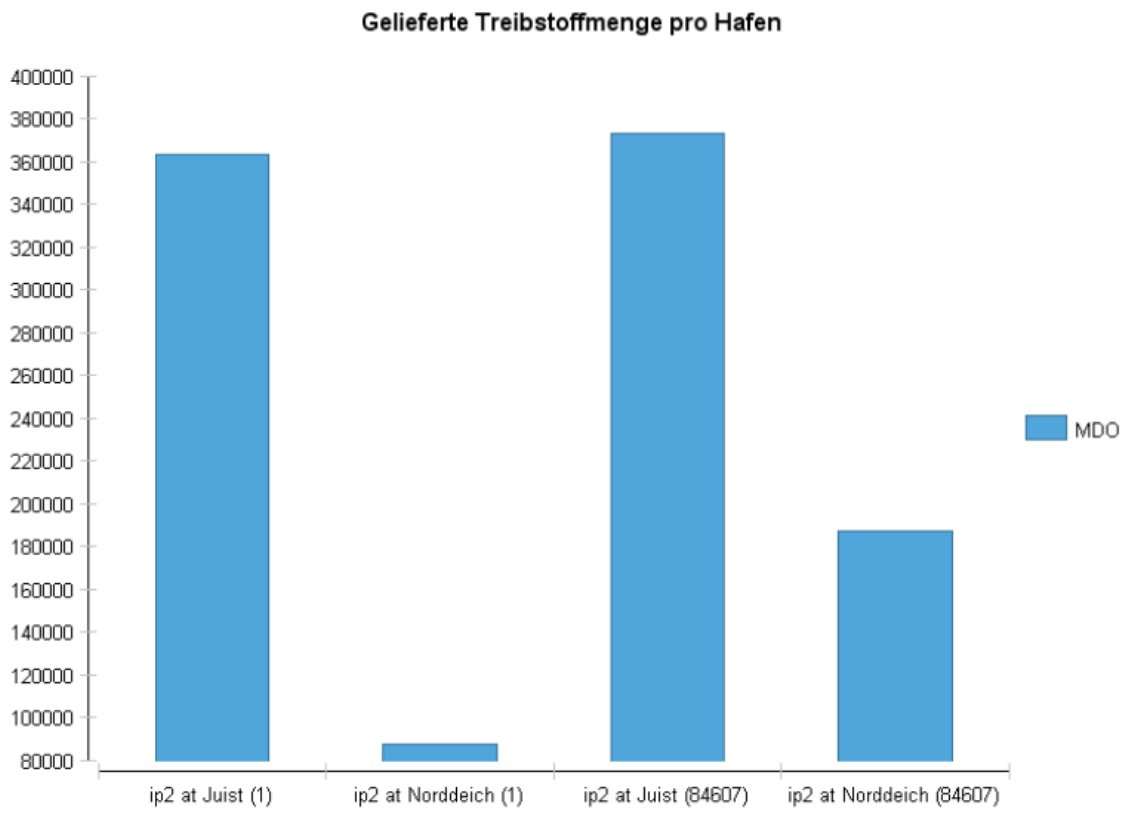

Abbildung 72 - Treibstoffmengen des Hafenberichts

B. Beschreibung der Reporting Oberfläche

Die im Folgenden beschriebene grafische Benutzeroberfläche der Reporting-Komponente (GUI) ist im Editor eingebunden und kann im Anschluss an einen Simulationsdurchlauf aufgerufen werden, um die Daten einer Simulation grafisch darzustellen.

Allgemein

Die GUI ist grundsätzlich in drei verschiedene Bereiche aufgeteilt. Als Erstes besteht die Möglichkeit Elemente auszuwählen, die im Bericht (Report) angezeigt werden sollen. Danach werden die zu erstellenden Berichte vom Nutzer ausgewählt und schließlich erstellt. Abschließend können Einstellungen der Datenbankverbindung gesetzt und modifiziert werden. Diese einzelnen Bereiche werden im folgenden Kapitel „Aufbau der GUI“ ausführlicher vorgestellt. Abbildung 73 stellt die GUI in ihrer Gesamtheit mit allen Teilbereichen dar.

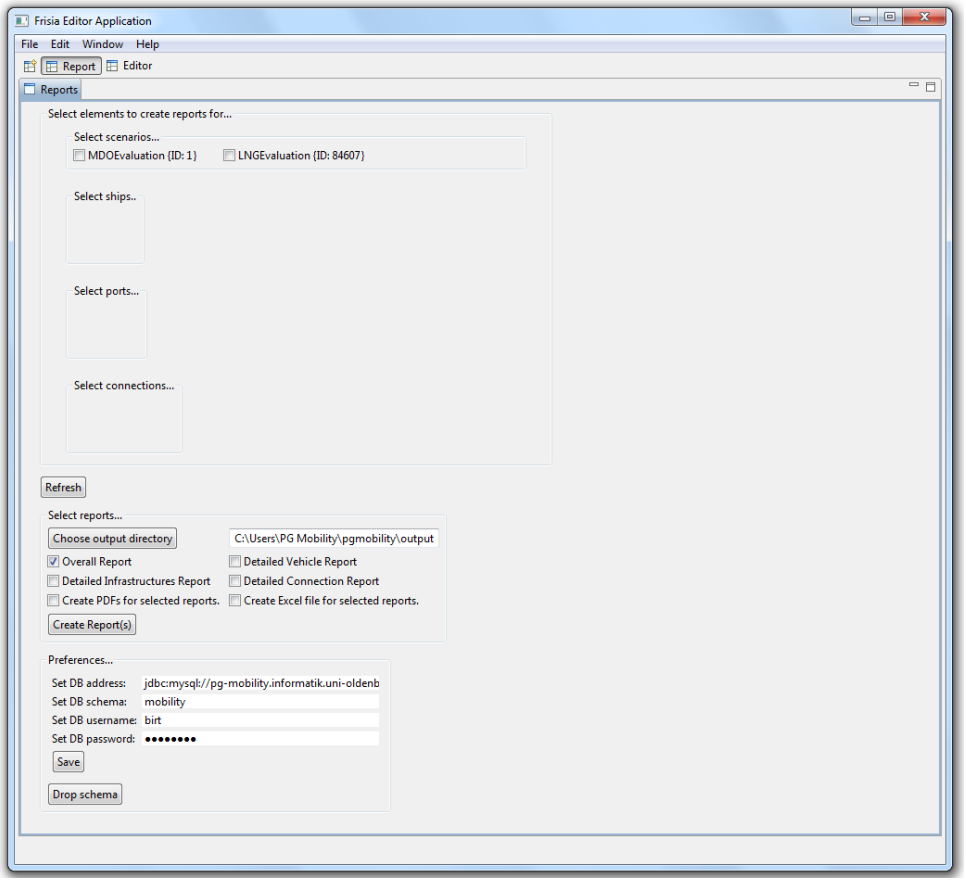

Abbildung 73 - Übersicht der Reporting Oberfläche

Aufbau der GUI

Die Elemente aus Abbildung 74 erlauben eine Auswahl derjenigen Elemente, die in den Berichten angezeigt werden sollen. Zuerst werden hierzu die geforderten Szenarien über Auswahlboxen ausgewählt. Die Elemente Schiffe (*Select ships…*), Häfen (*Select ports…*) und Strecken (*Select connections…*) passen sich hierbei automatisch an das ausgewählte Szenario an, sodass nur Szenariospezifische Elemente angezeigt werden. Die angezeigten Daten werden hierbei, aus der zugrunde liegenden Datenbank, dynamisch geladen.

Über einen Klick auf die Schaltfläche „*Refresh*“ ist es zudem möglich, alle getätigten Eingaben automatisch zurückzusetzen und eventuelle Änderungen der Datensätze aus der Datenbank zu erhalten.

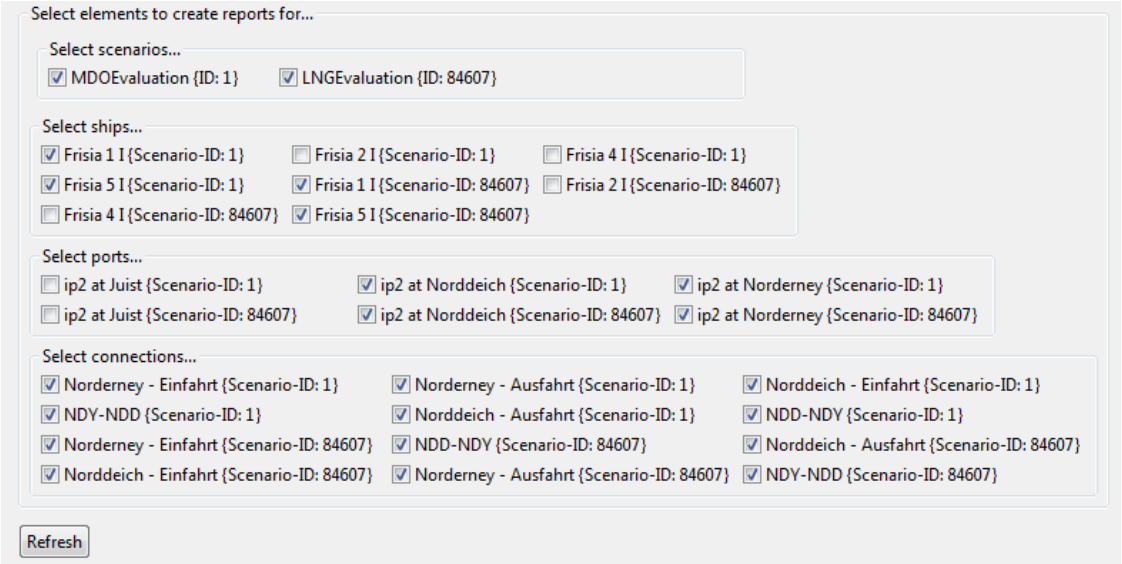

Abbildung 74 - Auswahl der Elemente

Des Weiteren besteht die Möglichkeit zu entscheiden, welche Berichte erstellt werden sollen. Hierbei stehen folgende vier Berichtstypen zur Auswahl, die im Abschnitt 7.3.5.4 und Anhang A näher erläutert werden.

- Übersichtbericht (Overall Report)
- Schiffbericht (Detailed Vehicle Report)
- Hafenbericht (Detailed Infrastructure Report)
- Streckenbericht (Detailed Connection Report)

Standardmäßig werden die Berichte nach einem Klick auf die Schaltfläche „*Create Report(s)*“ in Form von HTML-Dateien generiert. Ist zudem ein plattformunabhängiges Dokumentenformat gewünscht, so können über das Setzen der in Abbildung 75 dargestellten Auswahlbox „*Create PDFs for selected reports*“ zusätzlich PDF-Dateien angelegt werden. Ebenfalls ist es möglich, die Daten als Excel-Datei auszugeben, um bspw. eine Weiterverarbeitung der Ergebnisse zuzulassen. Der Ausgabepfad kann über die Schaltfläche „*Choose output directory*“ gesetzt werden.

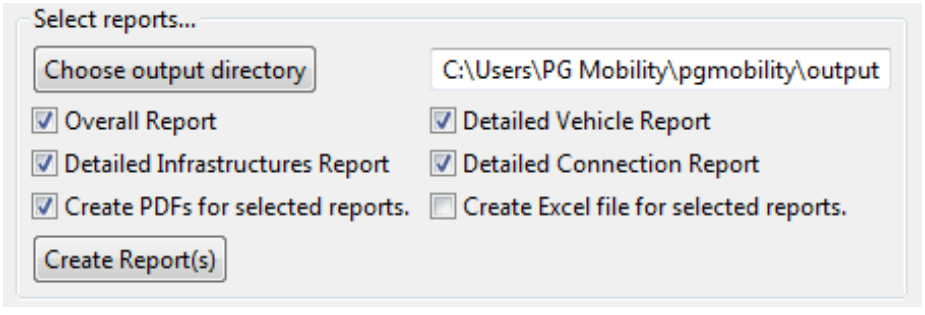

Abbildung 75 - Auswahl der Berichte

Im dritten Teilabschnitt der Reporting-Oberfläche lassen sich spezifische Einstellungen der Datenbankverbindung setzen bzw. verändern. Hierzu zählen neben der Adresse der Datenbank auch das betreffende Datenbankschema, der Benutzername sowie das entsprechende Passwort. Wurden Änderungen dieser Einstellungen vorgenommen, müssen diese über einen Klick auf die Schaltfläche „*Save*“ gespeichert werden. Ebenso ist es möglich, das Schema über die Schaltfläche „*Drop Schema*“ zu leeren, indem es zunächst gelöscht und anschließend wieder neu erstellt wird. Abbildung 76 zeigt den entsprechenden Ausschnitt der Oberfläche.

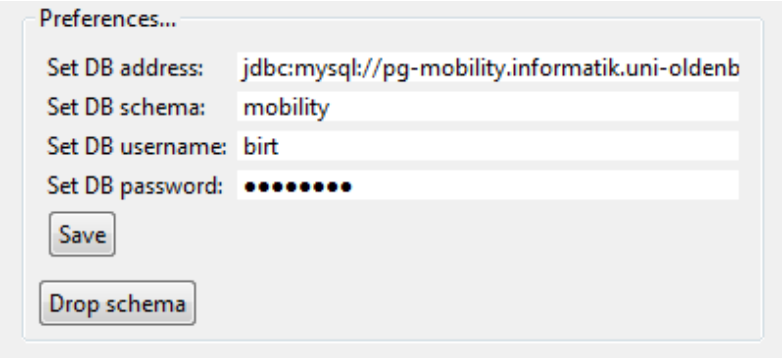

Abbildung 76 - Auswahl der Einstellungen

Wurde die Erstellung der Berichte über die Schaltfläche „*Create Report(s)*“ angestoßen, so öffnet sich das in Abbildung 76 dargestellte Fenster, das den Status der Generierung über eine Fortschrittsanzeige ausgibt.

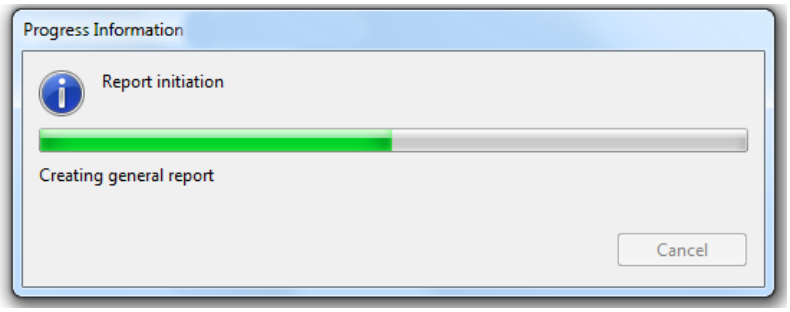

Abbildung 77 - Generierung der Berichte

Nach Erstellung der Berichte werden die HTML-Versionen in einem SWT-Browser-Widget im Editor dargestellt. Daneben erfolgt durch entsprechende Auswahl bspw. auch eine lokale Speicherung als PDF-Datei oder Excel-Output. Abbildung 78 zeigt die GUI, nach dem die Berichte erstellt wurden.

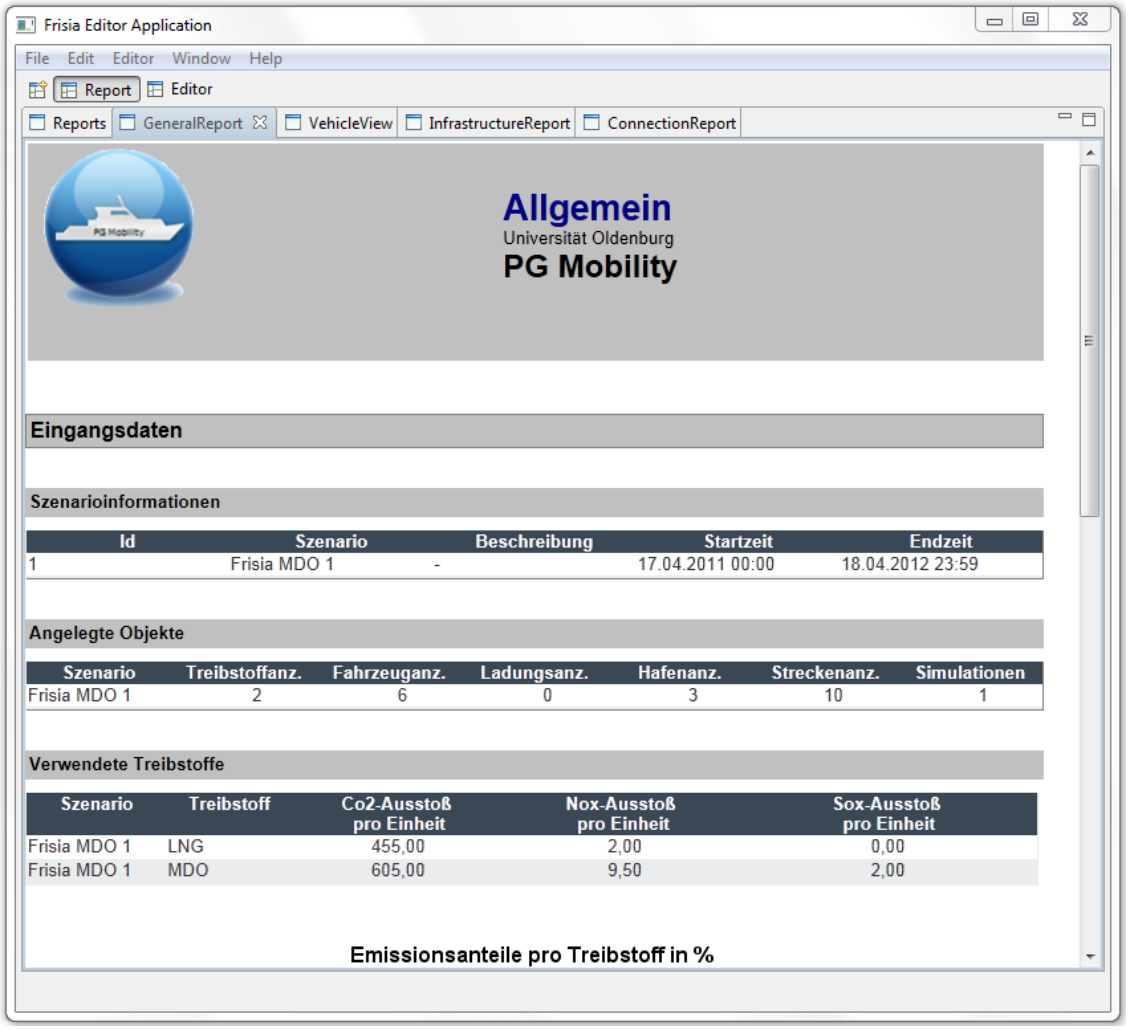

Abbildung 78 - Berichte anzeige